A Bi-Level Multi-Objective Approach for Web Service Design Defects Detection

Soumaya Rebai[a], Marouane Kessentini[a], Hanzhang Wang[b], Bruce Maxim[a]

[a] *University of Michigan, Dearborn, Michigan, USA*
[b] *eBay, San Jose, California, USA*

Abstract

Context: Web services frequently evolve to integrate new features, update existing operations and fix errors to meet the new requirements of subscribers. While this evolution is critical, it may have a negative impact on the quality of services (QoS) such as reduced cohesion, increased coupling, poor response time and availability, etc. Thus, the design of services could become hard to maintain and extend in future releases. Recent studies addressed the problem of web service design antipatterns detection, also called design defects, by either manually defining detection rules, as combination of quality metrics, or generating them automatically from a set of defect examples. The manual definition of these rules is time-consuming and difficult due to the subjective nature of design issues, especially to find the right thresholds value. The efficiency of the generated rules, using automated approaches, will depend on the quality of the training set since examples of web services antipatterns are limited. Furthermore, the majority of existing studies for design defects detection for web services are limited to structural information (interface/code static metrics) and they ignore the use of quality of services (QoS) or performance metrics, such as response time and availability, for this detection process or understanding the

Email addresses: srebal@umich.edu (Soumaya Rebai), marouane@umich.edu (Marouane Kessentini), hanzwang@ebay.com (Hanzhang Wang), bmaxim@umich.edu (Bruce Maxim)

impact of antipatterns on these QoS attributes.**Objective:** To address these challenges, we designed a bi-level multi-objective optimization approach to enable the generation of antipattern examples that can improve the efficiency of detection rules. **Method:** The upper-level generates a set of detection rules as a combination of quality metrics with their threshold values maximizing the coverage of defect examples extracted from several existing web services and artificial ones generated by a lower level. The lower level maximizes the number of generated artificial defects that cannot be detected by the rules of the upper level and minimizes the similarity to well-designed web service examples. The generated detection rules, by our approach, are based on a combination of dynamic QoS attributes and structural information of web service (static interface/code metrics). **Results:** The statistical analysis of our results, based on a data-set of 662 web services, confirms the efficiency of our approach in detecting web service antipatterns comparing to the current state of the art in terms of precision and recall. **Conclusion:** The multi-objective search formulation at both levels helped to diversify the generated artificial web service defects which produced better quality of detection rules. Furthermore, the combination of dynamic QoS attributes and structural information of web services improved the efficiency of the generated detection rules.
Keywords: Search based software engineering, quality of services, services design.

---

1. Introduction

Service-Oriented Computing (SOC) has emerged as an evolutionary paradigm that is changing the way software applications are implemented, deployed, and delivered to help industry meet their ever-more-complex challenges [1]. Nowa-

days, SOC is becoming widely accepted in industry such as FedEx [1], Dropbox [2], Google Maps [3], eBay[4], etc. The massive adoption of this paradigm and its popularity are mainly due to the offered reusability, modularity, flexibility, and scalability [2]. SOC utilizes services which are independent and portable program units as fundamental elements to support rapid, low cost development of heterogeneous and distributed systems [3].

Any successful deployed web services evolve over time to meet the new changes in the requirements and/or to fix bugs. The continuous changes and evolution of web services may create poor and bad design practices which are generally called "antipatterns" that can impact the performance and usability of the web service [4]. Maintaining a good design quality is critical but it is excessively expensive both in time and resources for the service providers.

To detect web service antipatterns, most of the existing studies consider only the interface or code-level metrics of bad-designed web services [5, 6, 7, 8]. Therefore, they enable developers to evaluate the quality of their service using mainly static information extracted from the implementation details of the interface and the services, such as coupling, cohesion, and number of operations. However, it is widely known that the quality of service metrics such as the response time and availability play a significant role in evaluating the overall performance of a service-based system. Furthermore, most of these studies [5, 6, 7, 8] are based on declarative rule specification. The 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

[1] http://www.fedex.com/ca_english/businesstools/webservices
[2] https://www.dropbox.com/developers/core
[3] developers.google.com/maps/documentation/webservices
[4] https://developer.ebay.com/docs

high calibration efforts to find the right threshold value for each metric. In addition, the translation of the symptoms into rules is not obvious because several symptoms can described using multiple metrics and thresholds.

To address these challenges, few heuristic-based approaches are proposed to generate design defects detection rules from defect examples [9, 10]. However, such studies require a high number of interface design defect examples (data) to provide efficient detection rules solutions. In fact, design defects are rarely documented by developers which explains the need for an approach that is able to generate artificial defects examples in order to improve the efficiency of detection rules. In addition, it is challenging to ensure the diversity of the examples to cover most of the possible bad-practices. In addition, these heuristic-based studies are still also limited to the use of structural metrics and did not consider the impact of antipatterns on the performance of the services.

In this work, we start from the hypothesis that the generation of efficient web service defect 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 and 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 [11]. 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 addition, we assume that an effective web service antipatterns detection process should

be based on a combination of *dynamic QoS attributes* and the *structural information* of web service (static interface/code metrics). Several of Web services antipatterns can negatively impact the QoS such as availability and response time. For instance, a GOWS antipattern typically can include a large number of operations which can reduce the response time dramatically. A GOWS web service suffers, in general, from a low cohesion which may lead to a high response time and a low availability due to the large number of calls between operations at multiple web services.The use of response time quality attribute may help to find the right threshold in terms of number of operations and cohesion level that can truly impact the web service performance. Thus, the generated detection rules can be more accurate.

In our approach, the upper level generates a set of detection rules, a combination of static and dynamic metrics and QoS attributes, which maximizes the coverage of the base of defect examples and the coverage of artificial defects which are generated by the lower level and minimizes the size of a generated rule. The lower level maximizes the number of generated artificial defects that cannot be detected by the rules produced by the upper level and minimizes the distance between the artificial defects and the base of bad-designed web services examples. The advantage of our bi-level approach is that the generation of detection rules is not limited to some defect examples that are hard to collect. However, this approach allows the prediction of new defects that are different from those in the base of examples. Furthermore, our problem requires a search in a large space for a solution which balances different conflicting objectives to generate rules suitable for different scenarios. Therefore, it would be appropriate to consider a multi-objective search-based approach that finds a trade-off between conflicting objectives in each level.

We applied and validated these rules on a benchmark of 662 real-world web

services from different application domains and five common web service antipatterns.However, our proposed approach can be used in a generic way for any other type of defect as long as a number of examples are available. Statistical analysis of our experiments shows the efficiency of our bi-level multi-objective approach in detecting web service antipatterns, with a precision of 84% and recall of 91%. The results confirm the outperformance of our bi-level proposal compared to state-of-art web service design defects detection techniques [9, 10, 12] and our previous work limited to mono-objective bi-level approach using only structural metrics [8]. Thus, the validation confirmed our hypothesis that the detection of antipatterns require a combination of structural and performance metrics.

The remainder of this paper is organized as follows. Section 2 presents the relevant background related to this research, the problem statement, a motivation example, and the challenges of the presented work. Section 3 describes our approach overview and the problem adaptation. Empirical study and results are provided in Section 4 while threats to validity are discussed in Section 5. Section 6 is dedicated to related work. Finally, we conclude and provide our future research directions in Section 7.

2. Background and Problem Statement

In this section, we present the most frequent types of design defects for web services and the different types of metrics that can be used to evaluate the quality of a services. Then, we will describe a motivating example for our proposed approach.

2.1. Web Services Design 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,

comprehensibility and discoverability [13] which may impact the usability and popularity of services [14]. 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 [15]. 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, some studies defined different types of web services design defects [16, 15]. In our experiments, we focus on the eight following web service defect types since they are the most frequent and severest ones [17], and also to be able to compare our detection approach to the state of the art:

- *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.
- *Redundant PortTypes (RPT):* is an antipattern where multiple portTypes are duplicated with the similar set of operations. In fact, one of the potential sources of RPT defects is the use of defective WSDL generation

tools as pointed out in [13]. Another source of RPT is when developers are adding new features in a rush without considering the reusability of their implementation and architecture design (similar to code clones).

The web service antipatterns detection mechanism involves finding the fragments of the design which violate some quality indicators. Table 1 describes all the metrics that are used in this paper to cover bad quality symptoms. These metrics are a combination of static, dynamic and performance metrics related to the following abstraction levels of web services applications :

- **Web service interface-level (WSDL) metrics:** are mainly related to the interface, message, operation and Port type. The list of WSDL metrics are described in Table1 from ALPS until RPT. In our approach, we are considering the two WSDL versions 1.0 and 2.0 since they are both supported by our parser in extracting the metrics.

- **Web service code-level metrics:** are the static information that we can extract from the services code skeletons. The most widely-used code-level metrics are those defined by Chidamber and Kemerer [18] as described in Table 1 (from Ca until CC). For all code-level metrics, we calculate the average value for all the classes that implement the specific web service. For instance , the depth of inheritance (DIT) represents the depth of inheritance of a class and it is defined as the depth of the class in the inheritance tree and the depth of a node of a tree refers to the length of the maximal path from the node to the root of the tree. Thus, we parsed the code to extract the calls by static analysis and also used relevant keywords such as “extends” to confirm the nature of these calls. The QoS metrics are more related to the execution of web services to calculate response time, availability, etc.

- **Quality of Service (QoS) metrics:** we selected 9 popular metrics (From Response until Documentation in Table 1), namely response, availability, throughput, successability, reliability, and latency are dynamic metrics which measure the web service overall performance. Documentation and compliance are static metrics to measure the usability of the web service interface. In our work, We extracted all these metrics from the QWS dataset [19].

We selected these defect types in our experiments because they are the most frequent, the hardest to detect [20, 9], and cover different maintainability factors. We have also several examples of these defects and we are able to compare the performance of our detection technique to existing studies [8, 9, 12]. However, the proposed approach in this paper is generic and can be extended to any type of defect.

2.2. Motivating Example and Challenges

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. Thus, it may not be accurate to identify a GOWS defect based on structural information such as the number of operations. Both structural and non-structural (QoS attributes) factors are complementary when detecting a GOWS antipattern. The impact

Category	Metric Name	Definitions
QoS Metrics	Response	Time to send a request and receive a response (ms)
	Availability	Number of successful invocation/total invocation (%)
	Throughput	Number of invocations for a given time (invokes/sec)
	Successability	Number of response/number of request messages (%)
	Reliability	Ratio of number of error messages to total messages (%)
	Compliance	The extent to which a WSDL follows specification (%)
	Best practices	The extent to which a service follows WS-I BAsic (%)
	Latency	Time taken for the server to process a given request (ms)
	Documentation	Measure of documentation (i.e. description tags) in WSDL
Interface Metrics	ALPS	Average length of port types signature
	COH	Cohesion
	COUP	Coupling
	NAOD	Number of accessor operations declared
	NCO	Number of CRUD operations
	NOD	Number of operations declared
	NOPT	Average number of operations in port types
	NPT	Number of port types
	RAOD	Ratio of accessor operations declared
	ALOS	Average length of operations signature
	AMTO	Average number of meaningful terms in operations names
	ANIPO	Average number of input parameters in operations
	ANOPO	Average number of output parameters in operations
	NPO	Average number of parameters in operations
	ALMS	Average number of message signature
	AMTM	Average number of meaningful terms in message names
	NOM	Number of messages
	NPM	Average number of parts per message
	AMTP	Average number of meaningful terms in port type names
	NCT	Number of complex types
	NCTP	Number of complex types parameters
	NST	Number of primitive types
	RPT	Ratio of primitive types over all defined types
Code Metrics	Ca	Afferent couplings
	CAM	Cohesion Among Methods of Class
	CBO	Coupling Between Object Classes
	Ce	Efferent couplings
	DAM	Data Access Metric
	DIT	Depth of Inheritance Tree
	LCOM	Lack of cohesion in methods
	LCOM3	Lack of cohesion in methods
	LOC	Lines of Code
	MFA	Measure of Functional Abstraction
	MOA	Measure of Aggregation
	NOC	Number of Children
	NPM	Number of Public Methods
	RFC	Response for a Class
	WMC	Weighted methods per class
	AMC	Average Method Complexity
	CC	The McCabe's cyclomatic complexity

Table 1: List of Web services metrics used

of the appearance of GOWS can be seen on the performance of services such as response time and availability. Thus, these attributes can confirm a GOWS antipattern rather than just relying on number of operations. In fact, it is always challenging to define a threshold for the number of operations but a combination of both low QoS attributes and high number of operations will definitely improve the accuracy of the GOWS detection rules. Programmers are mainly interested to fix design defects impacting the quality of services and not those who just violate some metrics such as coupling, cohesion and number of operations. However, existing studies are limited to the use of structural information when detecting design defects.

Our GOWS motivating example was not only related to the size of the interface but also other metrics such as low cohesion. The used Amazon service's interface suffers from low-cohesion and it is already classified in our dataset as a GOWS antipattern, its high response time and low availability can be explained by the low cohesion of operations and not only the size of the interface. If the number of operations becomes high (like in most GOWS antipatterns) then the response time and availability will be dramatically impacted. In practice, one of the main reasons of services low availability is the high number of calls that make some servers inaccessible/down.

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 which results in lack of defects examples. Thus, it is time-consuming and difficult to collect defects and inspect manually large web services. In fact, unlike the bugs localization problem where bug reports data are available to train the model, detecting web services antipatterns suffers from the lack of documented defect examples which affects the efficiency of the generated detection rules. In addition, it is challenging to ensure the diversity of

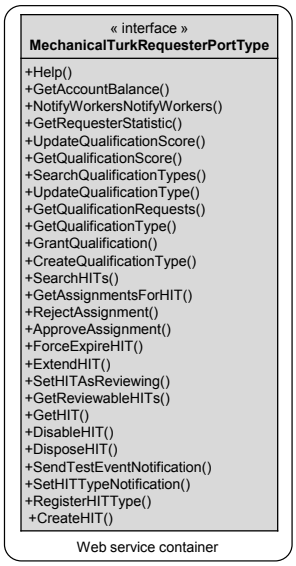


Figure 1: God object Web service (GOWS) example.

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 multi-objective optimization problem.

3. Bi-level Multi-objective Optimization for Web services Defects Detection

3.1. Bi-level Multi-objective Optimization Technique

In this study, we considered the web services defect detection problem as a bi-level multi-objective optimization problem where the optimal solution of the lower level problem determines the feasible space of the upper level optimization problem [11, 21]. In our adaptation, the upper level problem is the generation of detection rules and the lower level problem is the generation of design defects that may not be detected using the rules of the upper level solutions.

We start by describing the basic concepts of bi-level optimization, then we introduce the multi-objective optimization technique.

3.1.1. 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. Bi-level optimization problem (BLOP) also called two-level optimization, is a specific type of optimization where one problem is nested within another [11, 21]. 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. 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 as described in Figure 3.

The problem contains two types of variables: (1) the upper-level variables x_u and (2) the lower-level variables x_l. Formally, BLOP is defined as follows:

Definition1. For the upper-level objective function F: $\mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$ and lower-level objective function $f : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}$, the bi-level problem is given by:

$$\begin{aligned} \min_{x_u \in X_U, x_l \in X_L} \quad & F(x_u, x_l) \\ \text{subject to} \quad & x_l \in argmin\{f(x_u, x_l), g_j(x_u, x_l) \leq 0, j = 1, ..., J\} \\ & G_k(x_u, x_l) \leq 0, k = 1, ..., K \end{aligned}$$

where $G_k : X_U \times X_L \to \mathbb{R}$ and $g_j : X_U \times X_L \to \mathbb{R}$ denote respectively the

upper and the lower level constraints. J is the population size at the upper level, K is the population size at the lower level and n is the number of fitness functions,

The study involved multiple objectives at the upper lever, and multiple objectives at the lower level. Thus, the next section presents the Multi-objective optimization technique.

Existing methods to solve BLOPs could be classified into two main families: (1) classical methods and (2) evolutionary methods. The first family includes extreme point-based approaches [22], penalty function methods [23] and trust region methods [24]. The main shortcoming of these methods is that they heavily depend on the mathematical characteristics of the BLOP at hand. The second family includes meta-heuristic algorithms that are mainly Evolutionary Algorithms (EAs). Recently, several EAs have demonstrated their effectiveness in tackling such type of problems thanks to their insensibility to the mathematical features of the problem in addition to their ability to tackle large-size problem instances by delivering satisfactory solutions in a reasonable time [25, 26, 27].

In our adaptation, each level is formulated as a multi-objective problem. The next sub-section will give details about multi-objective optimization.

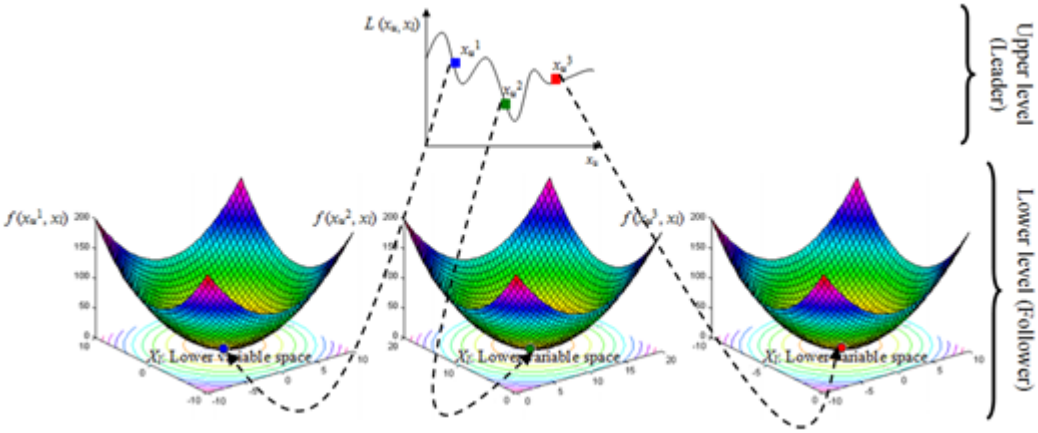


Figure 2: The upper and lower levels of the Bi-Level process

3.1.2. Multi-Objective Optimization

Multi-Objective search considers more than one objective function to be optimized simultaneously. Objective functions are used to evaluate the generated solutions. It is hard to find an optimal solution that solves such problem because the objectives to be optimized are conflicting. For this reason, a multi-objective search-based algorithm could be suitable to solve this problem because it finds a set of alternative solutions, rather than a single solution as result. One of the widely used multi-objective search techniques is NSGA-II [28] that has shown good performance in solving several software engineering problems [29].

A high-level view of NSGA-II is depicted in Algorithm 3. 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* [28] 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 :

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

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 PF_0 get assigned dominance level of 0 Then, after taking these solutions out, *fast-non-dominated-sort* calculates the Pareto-front PF_1 of the remaining

Algorithm 1 High level pseudo code for NSGA-II

```
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
```

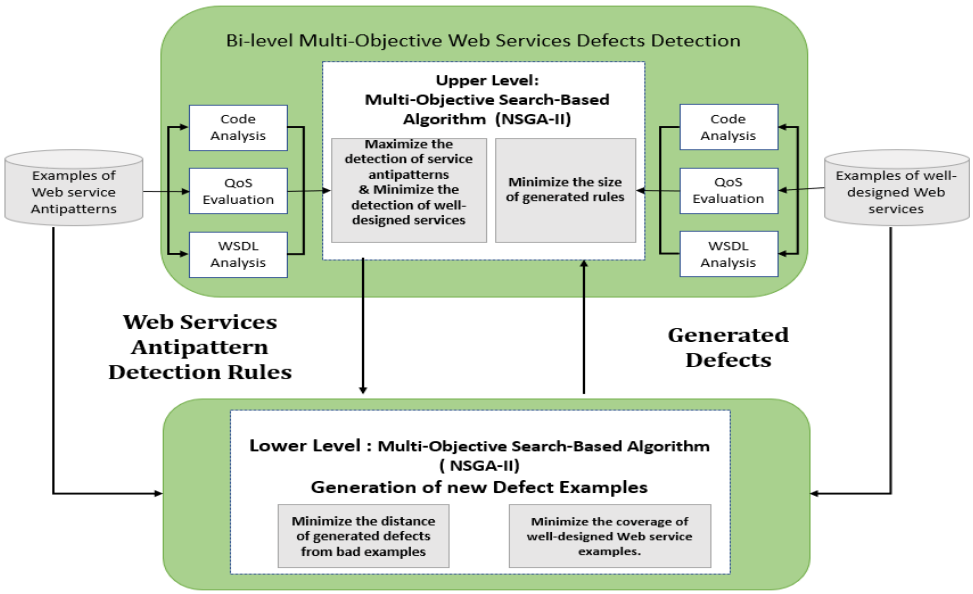
population; solutions on this second front get assigned dominance level of 1, and so on. The dominance level becomes the basis of selection of individual solutions for the next generation. Fronts are added successively until the parent population P_{t+1} is filled with N solutions (line 8). When NSGA-II has to cut off a front PF_i and select a subset of individual solutions with the same dominance level, it relies on the crowding distance [28] to make the selection (line 9). This parameter is used to promote diversity within the population. This front PF_i to be split, is sorted in descending order (line 13), and the first (N- $|P_{t+1}|$) elements of PF_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).

Therefore, a bi-level multi-objective optimization problem involves two levels of multi-objective optimization problems, each one implements an NSGA-II algorithm with different set of objectives as described in the next subsection.

3.2. Approach Overview

As Figure 3 shows, our Bi-Level Multi-Objective (BLMO) approach includes two levels where both the leader and the follower have two objectives.

Figure 3: Bi-level Multi-Objective Web service defects detection overview



As described in Figure 3, the proposed approach takes as inputs two sets of web service examples: (1) one set contains service antipattern examples and (2) another has well-designed service examples. It extracts the metrics, previously described, of each web service in the sets. Then, the upper level generates a set of detection rules per solution. The detection rule generation process selects randomly, from the list of possible metrics, a combination of quality metrics and their threshold values to detect a specific antipattern type. Therefore, the optimal solution is a set of detection rules that best detect the antipatterns of the base of examples and those generated by the lower level while minimizing the number of generated rules.

The follower (lower level) uses well-designed web service examples to generate "artificial" design defects based on the notion of deviation from a reference (well-designed) set of web services. The generation process of web services defect examples is performed using a multi-objective heuristic search that maximizes on one hand, the distance between generated web service defect examples and reference examples and, on the other hand, maximizes the number of generated examples that are not detected by the leader (detection rules).

In our bi-level multi-objective approach, the two levels are dependent and therefore there is no parallelism. The upper level is executed for a number of 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 detection rules, and then this process is repeated several times until reaching a termination criterion such as the number of iterations. For each level, we selected the ideal point from the Pareto front of solutions which corresponds to the closest solution to the best possible values of the fitness functions.

3.3. Problem Formulation

3.3.1. Solution Representation

Each candidate solution in the upper level is a sequence of detection rules where each rule is represented by a binary tree such that:

- The Root and each internal node represent a logic operator either "AND" or "OR" to connect other nodes.
- Each leaf node represents a quality metric and its corresponding threshold.

For example, the following rule of Fig. 4 states that a web service s satisfying the following combination of metrics is considered as a GOWS defect:

As described in Figure 5, the generated structure of defects, in the lower level, is represented as a vector where each element is a (metric, threshold)

Algorithm 2 Upper level algorithm

Inputs: Quality of web service metrics M, web services defect examples base B, Well-designed web services base D, Number of best upper solutions that are considered for lower level optimization nbs, Upper population size N1, Lower population size N_2, Upper number of generations G1, Lower number of generations G_2
Output: Best detection rule BDR
Begin
$P_0 \leftarrow Initialization(N, M)$
for each DB_e in P_0 **do**
  $BCS_0 \leftarrow NSGA-IIWSDefectsGeneration(\mathrm{DR}_0, D, N_2, G_2)$;
  $BR_0 \leftarrow Evaluations(\mathrm{DR}_0, B, BCS_0)$;
end for
$t \leftarrow 1$
while t<G1 **do**
  $Q_t \leftarrow Variation(\mathrm{P}_{t-1})$
  for each DR_t in Q_t **do**
    DR_t = UpperEvaluations(DR_t,B);
  end for
  for each of the best nbs **rules** DR_t in Q_t **do**
    $BCS_t \leftarrow NSGA-IIWSDefectsGeneration(\mathrm{DR}_t, \mathrm{D}, N_2, G_2)$;
    $DR_t \leftarrow EvaluationsUpdate(\mathrm{DR}_t, BCS_t)$;
  end for
  $U_t \leftarrow \mathrm{P}_t \cup \mathrm{Q}_t$;
  $P_{t+1} \leftarrow EnvironmentalSelection(\mathrm{N}_1, U_t)$;
  $t \leftarrow$t+1;
end while
BDR $\leftarrow IdealPointSelection(\mathrm{P}_t)$;
END

element that characterizes the generated artificial web service defect.

3.3.2. Fitness Functions

At the upper level, we aim to optimize two fitness functions. The first one is formulated to maximize the coverage of web services defect examples (input) and the coverage of the generated artificial web service defects by the lower level. The second fitness function is formulated to minimize the size of the generated rule. Thus, the fitness functions at the upper level are defined as follows:

$$\begin{aligned} Maximize f_{upper,1} &= \frac{Precision\ (SR, WSDE{+}AWSD){+}\ Recall(SR, WSDE{+}AWSD)}{2} \\ Minimize f_{upper,2} &= size(DetectionRules) \end{aligned} \tag{1}$$

Algorithm 3 Lower level algorithm: NSGA-IIWSDefectsGeneration

Inputs: Upper level detection rule UDR, Well-desiged web service examples base D, Population size N, number of generations G
Output: Best artificial web service defects BCS
Begin
$P_0 \leftarrow Initialization(N, D)$;
$P_0 \leftarrow Evaluation$(P$_0$,D,UDR);
$t \leftarrow 1$;
while t<G **do**
  $Q_t \leftarrow Variation$(P$_{t-1}$)
  $Q_t =$ Evaluation(Q_t,D,UDR);
  $U_t \leftarrow$P$_t \cup$Q$_t$;
  $P_{t+1} \leftarrow EnvironmentalSelection$($N$,U$_t$);
  $t \leftarrow$t+1;
end while
BCS $\leftarrow IdealPointSelection$(P$_t$);
END

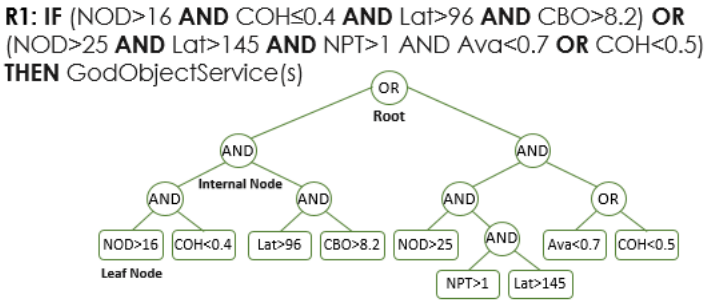


Figure 4: Solution Representation at the Upper Level.

where: $WSDE$ is the abbreviation for WS Defect Examples, $AWSD$ is the abbreviation for Artificial WS Defects and SR is the set of generated detection rules (solution).

At the lower level, for each solution of the upper level, an NSGA-II is executed to generate the best set of artificial defects that cannot be detected by the detection rules of the upper level. Two objective functions are formulated at the lower level to maximize the number of un-detected artificial defects that are generated and minimize the distance with web services antipatterns. More

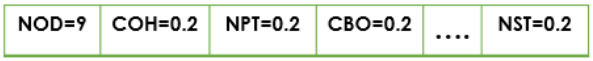


Figure 5: Solution Representation at the Lower Level.

formally, the two objectives are expressed as follows:

$$Minimize f_{lower,1} = \sum_{i=1}^{M} (ArtificialDefects(i) - Average(RAE(i)))$$
$$Minimize f_{lower,2} = countdefects(DR, AD) \quad (2)$$

where RAE is the abbreviation for References Antipatterns Examples, DR is the detection rules defined at the upper level, AD is the generated artificial defects and M is the number of metrics used to compare between artificial defects and the poor Web services examples. The first fitness function calculates the distance between the artificial defects and the ones in our base of examples to make sure that they are different.Thus, M is not restricted based on the type of antipatterns because we do not want our artifical examples to be restricted to limited behavior of antipatterns.

In our proposed approach, we are not generating detection rules only based on the QoS properties but we are including code-level metrics and interface metrics as well. Our benchmark/dataset contains web services along with their code-level , interface and QoS metrics and anti-patterns. The training data/examples guided the bi-level algorithm, via the fitness functions. to identify/generate the patterns and relationships between the different metrics and the anti pattern type. Since the fitness functions are mainly based on coverage criteria thus we can confirm that the best detection rules can be generalized on a large number of web services.

3.3.3. Change Operators

For the upper level, the mutation operator can be applied to a leaf node (metric), or to an internal node (logical operator) in our tree representation. It starts by randomly selecting a node in the tree. Then, if the selected node is a leaf (metric), it is replaced by another metric or another threshold value. Each metric has a maximum and minimum values which represent the range from where the change operator selects a threshold value. However, if it is an internal node (AND-OR), it is replaced by a new function. For the lower-level, the mutation operator consists of randomly changing a metric in one of the vector dimension.

Regarding the crossover at the upper level, two parents are selected, and a sub-tree is picked on each one. Then, the crossover operator swaps the nodes and their relative sub-trees from one parent to the other. The crossover operator can be applied to only parents having the antipatterns type to detect. Each child thus combines information from both parents.

The crossover operator allows creating two offspring Child1 and Child2 from the two selected parents Parent1 and Parent2, where the first x elements of Parent1 become the first x elements of Child2. Similarly, the first x elements of p2 become the first x elements of Child1.

4. Validation

In order to evaluate our approach for detecting antipatterns using the proposed bi-level multi-objective (BLMPO) approach, we conducted a set of experiments based on an existing benchmark[19]. Each experiment is repeated 30 times, and the obtained results are subsequently statistically analyzed with the aim to compare our multi-objective bi-level proposal with a variety of existing web service antipatterns detection approaches. In this section, we first present

our research questions and then describe and discuss the obtained results.

4.1. Research Questions

We defined the following research questions for our empirical study:

- **RQ1.** To what extent does the proposed approach detect various types of web service antipatterns based on a combination of structural and dynamic (QoS) metrics? It is important to quantitatively assess the completeness and correctness of our BLMO detection approach based on QoS and multi-objective search.

- **RQ2.** How does BLOP perform compared to existing web service antipatterns detection algorithms not using QoS metrics and multi-objective search? This research question is helpful to evaluate the benefits of the use of a multi-objective algorithm at both levels since we will compare our approach to our previous work based on a bi-level mono-objective algorithm (BLOP)[8]. Furthermore, we compared our approach with another mono-level search based approach [9] and an existing deterministic approach, SODA-W [12] which is not based on heuristic search. SODA-W is based on manually defined rules(including threshold values) to detect web service antipatterns. Both approaches are limited to the use of structural metrics thus they are useful to evaluate the benefits of considering dynamic quality of services attributes.

- **RQ3.** To what extent the detection of Web service antipatterns based on a combination of QoS and structural metrics can be useful and relevant for practitioners? We collected the opinions of developers about our tool and their perception of the importance of several of detected web service antipattern types.

4.2. Experimental Settings

To evaluate the performance of the proposed approach, we used existing benchmarcks of referencen number [19] to build our final dataset which consists of 662 good and bad web services desing. These web services (1) have different sizes, (2) originate from various application categories such as financial, science, travel, weather, etc, (3) have available source code, and (4) belonging to different development teams. These web services are retrieved from the QWS dataset then filtered to eliminate the ones which are not running anymore to be used by subscribers. We extract their interface file and code skeleton. Then, we manually inspected and validated the antipatterns of these services.

We considered the different antipattern types described in Section 2. Table 2 shows the distribution of these antipatterns in the 662 web services. We used a 10-fold cross validation procedure. In fact, the 10-fold cross validation means that we split our data into 10 training data sets and 1 evaluation data set. For each fold, one category of services (evaluation data) is evaluated using the remaining nine categories (training data) as training examples. Then, we repeated the process ten times. We use the two measures of precision and recall to evaluate the accuracy of our approach and to compare it with existing techniques. 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.

Antipatterns types	**# services**	**Distribution**
GOWS	237	36%
FGWS	179	27%
CWS	39	5%
DWS	119	18%
RPT	113	17%

Table 2: Anti-pattern occurrences within the 662 Web Services.

To answer RQ1, we use both recall and precision to evaluate the efficiency of our approach in identifying antipatterns. We 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 evaluated on the effectiveness of BLMO compared to existing approaches using the precision (PR) and recall (RC) measures. All three approaches are tested on the same benchmark described in Table 2 to ensure a fair comparison.The distribution of antipatterns type means the number of services containing at least one instance of that type of antipattern divided by the total number of analyzed services. We have also evaluated the execution time (T) required by the different approaches.

To answer RQ3, we conducted a post-study survey with developers to understand what types of web services antipatterns are important for them in practice and how useful our detection tool. To this end, we asked 48 software developers, including 29 professional developers working on the development of services-based application and 19 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.

An often-omitted aspect in metaheuristic search is the tuning of algorithm parameters. In fact, parameter setting influences significantly the performance of a search algorithm on a particular problem. The stopping criterion was set to 100,000 fitness evaluations for all search algorithms in order to ensure fairness of comparison. 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. The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used to compare two related samples, matched samples, or repeated measurements on a single sample to assess whether their population mean ranks differ (i.e. it is a paired difference test). In our case, it was used due to the randomness of the meta-heuristic algorithms and to ensure that their out-performance is not random but consistent on 30 independent runs. Additionally, the other parameters value 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. For our bi-level approach, both lower-level and upper-level are run each with a population of 20 individuals and 30 generations. It should be noted that the lower-level routine is not called for all upper-level population members. To control, the high computational cost of our bi-level approach, only ns% of the best upper-level population members are allowed to call the lower-level optimization algorithm. Based on a parametric study, the value of 5% for ns is found to be adequate empirically in our experiments. For our experiment, we generated up to 100 artificial web service antipatterns from deviation with the best of examples.

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 Wilcoxon rank sum test [30] with a 95% confidence level ($\alpha = 5\%$). The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used when comparing two related samples to verify whether their population mean-ranks differ or not. The latter verifies the null hypothesis H0 that the obtained results of two algorithms are samples from continuous distri-

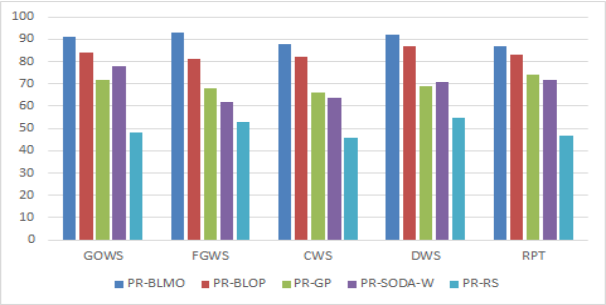


Figure 6: Median precision on 30 runs for the 10-folds of the 662 web services using the different detection techniques with a 95% confidence level.

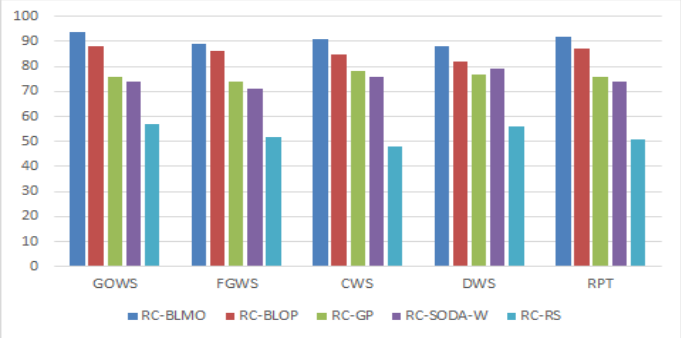


Figure 7: Median recall on 30 runs for the 10-folds of the 662 web services using the different detection techniques with a 95% confidence level.

butions with equal medians, as against the alternative that they are not, H1. In this way, we could decide whether the outperformance of BLMO over one of each of the other detection algorithms (or the opposite) is statistically significant or just a random result.

4.3. Results and Discussions

4.3.1. Results for RQ1

The results for the first research question RQ1 are presented in Figures 6 and 7. The obtained results show that our BLMO approach is able to detect most of the expected antipatterns in the different web services with a median precision higher than 90%. Thus, our technique does not have a bias towards

Table 3: Web services used in the empirical study.

Category	**# services**	**# antipatterns**	**average NOD**	**average NOM**	**average NCT**
Financial	121	52	31.73	57.31	22.14
Science	58	19	12.49	17.14	98.72
Search	49	21	9.66	18.94	28.43
Shipping	72	17	17.28	27.76	23.42
Travel	81	22	21.07	33.13	131.12
Weather	73	18	11.63	17.16	8.24
Media	33	19	11.8	16.4	32.29
Education	52	15	12.73	16.23	32.46
Messaging	63	20	9.18	13.36	18.25
Location	83	22	6.89	29.73	11.15
All	662	139	14.18	27.3	48.6

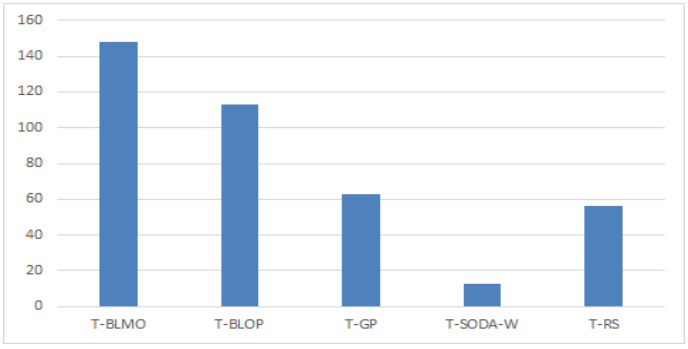


Figure 8: Median execution time on 30 runs for the 10-folds of the 662 web services using the different detection techniques.

the detection of specific web service antipattern types. As described Figures 6 and 7, we had an almost equal accuracy distribution of each Wev service antipattern types. Having a relatively good distribution of antipattern is useful for developers to make the right decisions about the quality of services. Overall, all the five web service antipatterns types are detected with good precision and recall scores in the different systems (an average of 91%). This ability to identify different types of antipatterns underlines a key strength to our approach. Most other existing tools and techniques rely heavily on the notion of size and static information to detect antipatterns. This is reasonable considering that some antipatterns like the GOWS are associated with the notion of size (number of operations). For web service antipatterns like RPT, however, the notion of size is less important, and this makes this type of anomaly hard to detect using structural information. This also confirms that the use of the dynamic quality of service attributes helped to achieve good results in detecting antipatterns.

The highest precision value for *GOWS* (93%) can be explained by the fact that these web service antipatterns are the easiest to detect due to their structure. For the web services antipattern type *DWS*, the precision is the lowest one (88%), but is still an acceptable score. These antipatterns are likely to be difficult to detect using metrics alone and may require interactions with the user. Sometimes developers have a reason why a Web service is too small such as they wanted to make sure that specific operations are loosely coupled to other services for security reasons. Thus, it is difficult to consider the context of specific requirements in static and dynamic rules. Similar observations are valid for the recall. The obtained results indicate that our approach is able to achieve an average recall of 89%. Thus, the quality of the detection rules are good for almost all the web service defect types considered in our experiments. Thus, we can conclude that our BLMO multi-objective approach detects well all

the types of considered antipatterns based on a combination QoS and structural metrics(RQ1).

4.3.2. Results for RQ2

The goal of the second research question is to investigate how well BLMO performs against random search (RS), our previous mono-objective bi-level work [8], an existing mono-level and single-objective approach (GP) [9] where only defect examples are used (without the consideration of the lower-level algorithm), and an existing detection tool (SODA-W) [31] not based on computational search. All these existing work did not consider the use of dynamic quality of service metrics and they are limited mainly to the interface level static metrics. The Random Search is implemented as a sanity check to justify the need for intelligent search. It has the same structure of our BLOP approach but without the selection and change operators and it is mainly based on random generation of solutions at both levels.

Figures 6 and 7 report the average comparative results on 30 runs with 95% as confidence level using the Wilcoxon rank sum test.The confidence level is the threshold to determine if the results are statistically significant or not. RS (random search at both levels using the same fitness functions) did not perform well when compared to BLMO both in terms of precision and recall achieving average less than 50% on the majority of different web service antipattern types. 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 BLMO are also better than the mono-objective bi-level and mono-level approaches [9, 8] in terms of both precision and recall. In fact, the mono-objective genetic programming technique have an average between 74% and 79% of precision and recall however BLOP (mono-objective bi-level) has better scores with an average of

more than 84% of precision and recall on most of the different web services. Thus, both techniques have lower precision and recall than BLMO. These results confirm that an intelligent search is required to explore the search space and that the use of the mutli-objective search at two levels along with the QoS attributes improved the obtained detection results.

While SODA-W shows promising results with an average precision of 76% and recall of 73%, it is still less than BLMO in all the five considered defect types. We conjecture that a key challenge 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 hard to generalize, 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.

Since our proposal is based on bi-level optimization, it is important to evaluate the execution time (T). It is evident that BLMO requires higher execution time than RS, BLOP, GP,and SODA-W since BLMO has an optimization algorithm to be executed at the lower level. To reduce the computational complexity of our BLOP adaptation, we selected only best solutions (10%) at the upper level to update their fitness evaluations based on the coverage of artificial web service antipatterns that are generated by the optimization algorithms executed at the lower level for every selected solution. All the search-based algorithms under comparison were executed on machines with Intel Xeon 3 GHz processors and 8 GB RAM. As described in Figure 8, all the existing studies were faster than BLMO. However, the execution for BLMO is reasonable because the al-

gorithm is executed only once then the generated rules will be used to detect antipatterns. There is no need to execute BLMO again except in the case that the base of examples (training set) will be updated with a high number of new web service antipattern examples.

One of the advantages of using our BLMO 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. The existing mono-level work of Ouni et al. [9] (GP) require a higher number of defect examples than BLMO to generate good quality of detection rules. We can conclude, based on the obtained results that our BLMO approach outperforms, in average, the state of the art web service antipatterns detection techniques that are not using multi-objective search and QoS metrics (response to RQ2).

4.3.3. Results for RQ3

Subjects were first asked to fill out a 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. The first part of the questionnaire includes questions to evaluate the relevance of some detected web service antipatterns by BLMO using the following scale: 1. Not at all relevant; 2. Slightly relevant; 3. Moderately relevant; and 4.Extremely relevant. If a detected antipattern is considered relevant then this means that the developer considers that it is important to fix it. The second part of the questionnaire includes questions for those antipatterns that are considered at least "moderately relevant". We asked the subjects to specify their usefulness based on the following options: 1. web services selection; 2. Quality assurance; 3. Bug prediction; 4. Effort prediction; and 5. Refactoring opportunities. The questionnaire is completed anonymously thus ensuring

confidentiality and this study were approved by the IRB at the University of Michigan: "Research involving the collection or study of existing data, documents, records, pathological specimens, or diagnostic specimens, if these sources are publicly available or if the information is recorded by the investigator in such a manner that participants cannot be identified, directly or through identifiers linked to the participants".

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 (one graduate student and one faculty) and not only answering the questions. A brief tutorial session was organized for every participant around web services antipatterns and quality of services to make sure that all of them have a minimum background to participate in the study. The instructions indicate also that the developers need to inspect the source code and the interfaces to evaluate the detected web service antipatterns and their relevance and not by evaluating the quality metric values. In addition, all the developers performed the experiments in a similar environment: similar configuration of the computers, tools and facilitators of the study. These sessions were also recorded as audio and the average time required to finish all the questions was 2 hours.

We evaluated, first, the relevance of a set of 30 detected web service antipatterns (6 instances from each type of antipattern) by the participants. Figure 9 illustrates that only less than 17% of detected antipatterns are considered not at all relevant by the developers. Around 68% of the antipatterns are considered as moderately or extremely relevant by the different participants, and this confirms the importance of the detected web service antipatterns for developers.

To better evaluate the relevance of the detected web service antipatterns, we investigated the types of antipatterns that developers perhaps consider them more or less important than others (e.g. RPT, GOWS, etc.). Figure 10 sum-

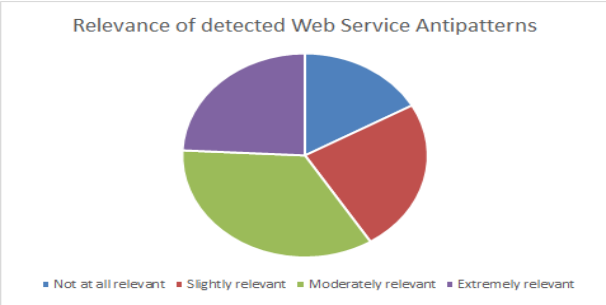


Figure 9: The relevance of detected web service antipatterns.

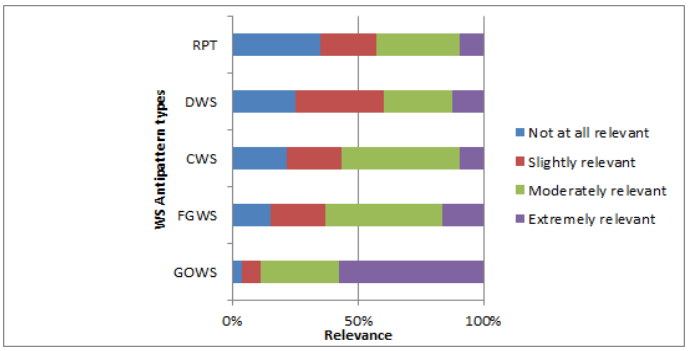


Figure 10: Relevance of different types of web service antipattern.

marizes our findings. It is clear that the detected GOWS are considered very relevant for developers. One of the reasons can be the impact of large number of operations on the performance of the services (response time, availability, etc.). In addition, it is very difficult for users to select a relevant operation when the interface contains a very large number of operations. Another interesting observation is that RPT antipatterns are not considered very relevant by developers. It is hard for developers to decide about the relevance of some types of antipattern without checking manually some of the detect ones and understand their possible impact. Thus, we did this post-study questionnaire to ask the developers after using the tool and checking some of the results. It may inform future research about which antipattern types to prioritize.

It is also important to evaluate the usefulness of the detected web service

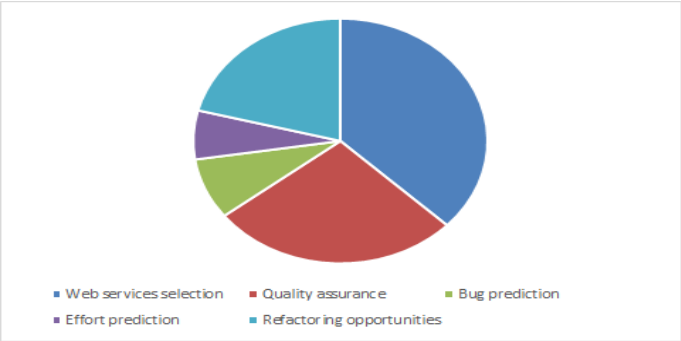


Figure 11: The usefulness of web service antipatterns for developers

antipatterns from the developers perspective. Thus, we asked the participants to justify the usefulness of the code-smells ranked as moderately or extremely relevant. Figure 11 describes the obtained results. The main usefulness is related to web services selection, refactoring guidance and quality assurance. In fact, most of the participants we interviewed found that the detected antipatterns give relevant advices about where refactorings should be applied to fix operations and portTypes. In addition, they found that the web service antipatterns detection process is much more helpful than the traditional analysis of quality metrics to find refactoring opportunities. They consider the use of traditional quality metrics for Quality Assurance as a time consuming process, and it is easier to interpret the results of detected antipatterns and apply the appropriate refactorings to improve the overall quality of the web services.

We summarize briefly in the following the feed-back of the participants during the think aloud sessions. Most of the participants mention that the detection rules generated by our bi-level multiobjective approach represents a faster solution than manual assessment of the quality of web services. The manual techniques represent a time consuming process to calibrate the metrics threshold or the combination of metrics to identify a maintainability issue manually. The participants found the detection rules useful to maintain a good quality of the

design of web services. In addition, the developers liked the flexibility to modify the rules (metrics or thresholds) if required. Some possible improvements for our detection techniques were also suggested by the participants. Some participants believe that it will be very helpful to extend the tool by adding a new feature to rank the detected web service antipatterns based on several criteria such as risk, cost and benefits. They believe that current web service quality assessment tools do not provide any support to estimate the risk, cost and benefits of fixing some maintainability issues.

To conclude, the developers found the use of QoS and multi-objective search efficient to detect web service antipatterns and found most of the detected types relevant (answer to RQ3)

5. Threats to Validity

Several factors may bias our empirical study. These factors can be classified in three categories: construct, internal and external validity. Construct validity concerns the relation between the theory and the observation. Internal validity concerns possible bias with the results obtained by our proposal. Finally, external validity is related to the generalization of observed results outside the sample instances used in the experiment.

One possible construct validity threat arises because although we considered several well-known web services design defect types, we must further evaluate the performance and ability of our bi-level technique to detect other defect types. A construct threat can also be related to the corpus of manually detected web service design defects since developers do not all agree if a candidate is a defect or not. We will ask some new experts to extend the existing corpus and provide additional feedback regarding the detected antipatterns. In addition, the recall score is challenging to calculate by the developers of our experiments

and requires additional participants to check its accuracy. A limitation related to our experiments is the difficulty to set the thresholds for some of existing state of the art techniques. In fact, we used the default thresholds used by these techniques that can have an impact on the quality of the results. The evaluation of detected web service defects for some participants is mainly based on the definitions of the antipatterns and the examples that we provided during the pilot study. However, the definition of antipatterns is subjective and depends on the programming behavior of the participants thus this can affect the accuracy of the detection results.

A construct threat is related to the fact that our detection results depend on the examples of antipatterns and well-designed web services. In addition, the generation of artificial web services can lead to several non-useful examples (generated by the lower-level). Additional constraints should be defined to better guide the search at a lower level to refine the generation of artificial web service examples. The same observation is valid for the used change operators at both the upper and lower levels that can generate invalid rules and antipattern examples (e.g. redundancy) may be avoided by the definition of additional constraints.

We take into consideration the internal threats to validity in the use of stochastic algorithms since 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 Wilcoxon rank sum test with a 95% confidence level. The parameter tuning of the different optimization algorithms used in our experiments creates another internal threat that we need to evaluate in our future work by additional experiments to evaluate the impact of upper and lower levels' parameters on the quality of the results.

For the selection threat, the subject diversity in terms of profile and ex-

perience could affect our study. We mitigated the selection threat by giving written guidelines and examples of antipatterns already evaluated with arguments and justification. Additionally, each group of subjects evaluated different antipatterns from different systems using different techniques/algorithms. Randomization also helps to prevent the learning and fatigue threats. Only few antipatterns per system were randomly picked for the evaluation. Diffusion threat is related to the fact that most of the subjects are from the same university, and the majority know each other. However, they were instructed not to share information about the experience before a certain date.

External validity refers to the generalization of our findings. In our study, external threat to validity concerns mainly the employed defect types and the studied web services. We considered five types of web service antipatterns which constitute a wide representative set of standard and most frequent defects. Likewise, we have selected 662 real world web services belong to different application domains, offer diverse functionalities, have different sizes and were developed by different companies.

6. Related Work

Web service antipatterns detection is a newly emerging area, therefore, few works addressed this problem using different techniques [32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44]. These techniques range from guidelines to fully automatic detection.

In 2003, Dudney et al. [45] have written the first book related to this topic. The catalog consists of 52 antipatterns in SOA, and especially in the area of Web-services. The main problem of the web service antipatterns definitions in this catalog is its informal definition.

More recently, Rotem-Gal-Oz et al. [15] have provided the symptoms of

some additional SOA antipatterns also informally. Furthermore, Král et al. [16] have described seven "popular" SOA antipatterns which violate the SOA principles but they did not discuss their detection. In another study, M. Hirsch et al. [46] have introduced a catalog of WSDL-based web services discoverability antipatterns.

Likewise, Torkmani et al. [47] have presented a repository of 45 general antipatterns in SOA. They have introduced a new method based on check lists to identify and detect antipatterns of SOA in service-oriented development. The goal of this work is a comprehensive review of these antipatterns that will help developers to work with a clear understanding of patterns in phases of software development and so avoid many potential problems. Other similar works are proposed by Rodriguez et al. [48, 49] and J. L. Ordiales Coscia et al. [50] who 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. The heuristics are simple rules based on pattern matching. Mateos et al. [51] have proposed an approach to prevent antipattern during the phase of WSDL documentation generation.

Moha et al. [17] have proposed a rule-based approach called SODA for the specification and detection of antipatterns on a service-based SCA system (Service Component Architecture). Each antipattern is specified with rule cards, which are sets of rules that use specific metrics. However, the proposed approach is restricted to SCA. Later, Palma et al. [31] extended this work by proposing the SODA-W approach inspired from SODA to specify and detect SOA antipatterns in web services. Indeed, authors used the relevant properties of web service-specific antipatterns to extend their previous domain-specific language (DSL). They performed detection for ten antipatterns. However, SODA-W is limited to a smaller set of WSDL interface metrics.

Search-Based Software Engineering (SBSE) uses a computational search approach to solve optimization problems in software engineering [52]. After the formulation of software engineering task as a search problem, by defining it in terms of solution representation, fitness function, and solution change operators, many search algorithms can be applied to solve that problem. The use of bi-level search technique was only used in [21]. However, this study is limited to detecting code smells in JAVA programs using static metrics and did not consider the use of multi-objective algorithms.

Recently, some search-based approaches to address web service antipatterns detection have been proposed [7, 9, 8, 10]. Ouni et al. [7] introduced a novel search-based approach to detect web service anti patterns including the god object web service, fine-grained web service, ambiguous web service, Data service and Chatty service. Using genetic programming approach, authors generate detection rules based on a combination of metrics and threshold values. However, the proposed approach is limited to only web service interface-level metrics (WSDL) and cannot consider all web service antipattern symptoms. The same authors [9] extended their previous work in different ways. Firstly, they proposed a novel automated approach for web service antipattern detection as a cooperative parallel optimization problem. Secondly, they extended their initial metric suite by including web service code-level metrics and web service dynamic metrics to better uncover potential antipattern symptoms. Thirdly, they extend their base of web service antipattern examples by incorporating three more antipatterns namely Redundant PortTypes, CRUDy Interface and Maybe It is NotRPC and finally, they extended the evaluation of the approach. As result, the experimental evaluation performed better than the previous work.

The main novelty of our approach is related to the consideration of both static and dynamic metrics to detect antipatterns unlike existing work limited

to static ones. Furthermore, our work considered the generation of defects while existing work are limited to the manual definition of rules or the generation of rules from a small set of examples.

7. Conclusions and Future Work

We proposed, in this paper, a bi-level multi-objective approach for the web service antipatterns detection problem. In our approach adaptation, the upper level generates a set of detection rules which are a combination of QoS, Interface, and code level metrics, using two conflicting fitness functions. The first objective is to maximize the coverage of both the base of defect examples and artificial defects generated by the lower level and to minimize the coverage of well-designed web service examples. The second objective is to minimize the size of a detection rule. The lower level generates artificial defects that cannot be generated by the upper-level detection rules which will help to generate fitter rules.

We implemented our proposed approach and evaluated it on a benchmark of 662 web services and several common web service antipattern types. The empirical study shows that proposed bi-level multi-objective optimization approach outperforms our previous multi-objective approach, bi-level approach and other state-of-the-art approaches. As part of our future work, we are planning to explore the use of bi-level for the automated repair of detected antipatterns.Additionally, we may consider other techniques such as the concept of generative adversarial networks [53] in generating the artificial defects. We will also work on the prioritization of detected defects due to the large number of potential issues that need to be fixed when improving the quality of Web services. For example, we can evaluate the impact of detected defects on the overall quality of service as a way to rank the identified antipatterns. Addi-

tionally, an empirical study about the impact of different antipatterns on QoS (maintainability, changeability, comprehensibility, discoverability) is considered as part of our future work.

References

References

[1] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, Service-oriented computing: a research roadmap, International Journal of Cooperative Information Systems 17 (02) (2008) 223–255.

[2] E. Newcomer, G. Lomow, Understanding SOA with Web services, Addison-Wesley, 2005.

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

[4] J. Král, M. Zemlicka, Popular SOA Antipatterns, in: Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009, pp. 271–276.

[5] D. Athanasopoulos, A. V. Zarras, G. Miskos, V. Issarny, Cohesion-Driven Decomposition of Service Interfaces Without Access to Source Code, IEEE Transactions on Services Computing 8 (JUNE) (2015) 1–18.

[6] D. Romano, M. Pinzger, Analyzing the evolution of web services using fine-grained changes, in: IEEE International Conference on Web Services (ICWS), 2012, pp. 392–399. doi:10.1109/ICWS.2012.29.

[7] A. Ouni, R. Gaikovina Kula, M. Kessentini, K. Inoue, Web service antipatterns detection using genetic programming, in: Proceedings of the 2015 on Genetic and Evolutionary Computation Conference, GECCO'15, ACM, 2015, pp. 1351–1358.

[8] H. Wang, M. Kessentini, A. Ouni, Bi-level identification of web service defects, in: International Conference on Service-Oriented Computing, Springer, 2016, pp. 352–368.

[9] A. Ouni, M. Kessentini, K. Inoue, M. O Cinneide, Search-based web service antipatterns detection, IEEE Transactions on Services Computing PP (99). doi:10.1109/TSC.2015.2502595.

[10] H. Wang, M. Kessentini, T. Hassouna, A. Ouni, On the value of quality of service attributes for detecting bad design practices, in: Web Services (ICWS), 2017 IEEE International Conference on, IEEE, 2017, pp. 341–348.

[11] J. F. Bard, Practical bilevel optimization: algorithms and applications, Vol. 30, Springer Science & Business Media, 2013.

[12] F. Palma, N. Moha, G. Tremblay, Y.-G. Guéhéneuc, Specification and detection of soa antipatterns in web services, in: European Conference on Software Architecture, Springer, 2014, pp. 58–73.

[13] J. L. O. Coscia, C. Mateos, M. Crasso, A. Zunino, Refactoring code-first web services for early avoiding wsdl anti-patterns: Approach and comprehensive assessment, Science of Computer Programming 89 (2014) 374–407.

[14] M. P. Singh, M. N. Huhns, Service-oriented computing - semantics, processes, agents, Wiley, 2005.

[15] A. Rotem-Gal-Oz, SOA Patterns, Manning Publications, 2012.

[16] J. Král, M. Žemlička, Crucial service-oriented antipatterns, International Journal On Advances in Software 2 (1) (2009) 160–171.

[17] N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, Y.-G. Guéhéneuc, B. Baudry, J.-M. Jézéquel, Specification and detection of soa antipatterns, in: Service-Oriented Computing, Springer, 2012, pp. 1–16.

[18] S. R. Chidamber, C. F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on software engineering 20 (6) (1994) 476–493.

[19] E. Al-Masri, Q. H. Mahmoud, The qws dataset (2008).

[20] D. Athanasopoulos, A. Zarras, Fine-grained metrics of cohesion lack for service interfaces, in: IEEE International Conference on Web Services (ICWS), 2011, pp. 588–595. doi:10.1109/ICWS.2011.27.

[21] D. Sahin, M. Kessentini, S. Bechikh, K. Deb, Code-smell detection as a bilevel problem, ACM Trans. Softw. Eng. Methodol. 24 (1) (2014) 6:1–6:44. doi:10.1145/2675067.
URL https://doi.org/10.1145/2675067

[22] W. Candler, R. Townsley, Linear two-level programming problem., COMP. & OPER. RES. 9 (1) (1982) 59–76.

[23] E. Aiyoshi, K. Shimizu, Hierarchical decentralized systems and its new solution by a barrier method., IEEE Transactions on Systems, Man and Cybernetics (6) (1981) 444–449.

[24] B. Colson, P. Marcotte, G. Savard, An overview of bilevel optimization, Annals of operations research 153 (1) (2007) 235–256.

[25] K. Deb, A. Sinha, An efficient and accurate solution methodology for bilevel multi-objective programming problems using a hybrid evolutionary-local-search algorithm, Evolutionary computation 18 (3) (2010) 403–449.

[26] F. Legillon, A. Liefooghe, E.-G. Talbi, Cobra: A cooperative coevolutionary algorithm for bi-level optimization, in: Evolutionary Computation (CEC), 2012 IEEE Congress on, IEEE, 2012, pp. 1–8.

[27] A. Koh, A metaheuristic framework for bi-level programming problems with multi-disciplinary applications, in: Metaheuristics for Bi-level Optimization, Springer, 2013, pp. 153–187.

[28] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: Nsga-ii, Evolutionary Computation, IEEE Transactions on 6 (2) (2002) 182–197.

[29] M. Harman, S. A. Mansouri, Y. Zhang, Search-based software engineering: Trends, techniques and applications, ACM Computing Surveys (CSUR) 45 (1) (2012) 11.

[30] F. Wilcoxon, S. Katti, R. A. Wilcox, Critical values and probability levels for the wilcoxon rank sum test and the wilcoxon signed rank test, Selected tables in mathematical statistics 1 (1970) 171–259.

[31] F. Palma, N. Moha, G. Tremblay, Y.-G. Guéhéneuc, Specification and detection of soa antipatterns in web services, in: Software Architecture, Springer, 2014, pp. 58–73.

[32] M. Kessentini, H. Sahraoui, M. Boukadoum, Example-based model-transformation testing, Automated Software Engineering 18 (2) (2011) 199–224.

[33] M. Kessentini, A. Bouchoucha, H. Sahraoui, M. Boukadoum, Example-based sequence diagrams to colored petri nets transformation using heuristic search, in: European Conference on Modelling Foundations and Applications, Springer, Berlin, Heidelberg, 2010, pp. 156–172.

[34] M. Kessentini, H. Sahraoui, M. Boukadoum, M. Wimmer, Search-based design defects detection by example, in: International Conference on Fundamental Approaches to Software Engineering, Springer, Berlin, Heidelberg, 2011, pp. 401–415.

[35] A. ben Fadhel, M. Kessentini, P. Langer, M. Wimmer, Search-based detection of high-level model changes, in: 2012 28th IEEE International Conference on Software Maintenance (ICSM), IEEE, 2012, pp. 212–221.

[36] M. Kessentini, M. Wimmer, H. Sahraoui, M. Boukadoum, Generating transformation rules from examples for behavioral models, in: Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications, ACM, 2010, p. 2.

[37] S. Kalboussi, S. Bechikh, M. Kessentini, L. B. Said, Preference-based many-objective evolutionary testing generates harder test cases for autonomous agents, in: International Symposium on Search Based Software Engineering, Springer, Berlin, Heidelberg, 2013, pp. 245–250.

[38] M. Kessentini, R. Mahaouachi, K. Ghedira, What you like in design use to correct bad-smells, Software Quality Journal 21 (4) (2013) 551–571.

[39] A. Ouni, R. Gaikovina Kula, M. Kessentini, K. Inoue, Web service antipatterns detection using genetic programming, in: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, ACM, 2015, pp. 1351–1358.

[40] U. Mansoor, M. Kessentini, B. R. Maxim, K. Deb, Multi-objective code-smells detection using good and bad design examples, Software Quality Journal 25 (2) (2017) 529–552.

[41] U. Mansoor, M. Kessentini, M. Wimmer, K. Deb, Multi-view refactoring of

class and activity diagrams using a multi-objective evolutionary algorithm, Software Quality Journal 25 (2) (2017) 473–501.

[42] M. W. Mkaouer, M. Kessentini, M. Ó. Cinnéide, S. Hayashi, K. Deb, A robust multi-objective approach to balance severity and importance of refactoring opportunities, Empirical Software Engineering 22 (2) (2017) 894–927.

[43] A. Ouni, M. Kessentini, K. Inoue, M. O. Cinnéide, Search-based web service antipatterns detection, IEEE Transactions on Services Computing 10 (4) (2017) 603–617.

[44] M. Fleck, J. Troya, M. Kessentini, M. Wimmer, B. Alkhazi, Model transformation modularization as a many-objective optimization problem, IEEE Transactions on Software Engineering 43 (11) (2017) 1009–1032.

[45] B. Dudney, J. Krozak, K. Wittkopf, S. Asbury, D. Osborne, J2EE Antipatterns, John Wiley; Sons, Inc., 2003.

[46] M. Hirsch, A. Rodriguez, J. M. Rodriguez, C. Mateos, A. Zunino, Spotting and removing wsdl anti-pattern root causes in code-first web services using nlp techniques: A thorough validation of impact on service discoverability, Computer Standards & Interfaces 56 (2018) 116–133.

[47] M. A. Torkamani, H. Bagheri, A Systematic Method for Identification of Anti-patterns in Service Oriented System Development, International Journal of Electrical and Computer Engineering 4 (1) (2014) 16–23.

[48] J. M. Rodriguez, M. Crasso, C. Mateos, A. Zunino, Best practices for describing, consuming, and discovering web services: a comprehensive toolset, Software: Practice and Experience 43 (6) (2013) 613–639.

[49] J. M. Rodriguez, M. Crasso, A. Zunino, M. Campo, Automatically detecting opportunities for web service descriptions improvement, in: Software Services for e-World, Springer, 2010, pp. 139–150.

[50] J. L. Ordiales Coscia, C. M. Mateos Diaz, M. P. Crasso, A. O. Zunino Suarez, Anti-pattern free code-first web services for state-of-the-art java wsdl generation tools.

[51] C. Mateos, J. M. Rodriguez, A. Zunino, A tool to improve code-first web services discoverability through text mining techniques, Software: Practice and Experience 45 (7) (2015) 925–948.

[52] M. Harman, S. A. Mansouri, Y. Zhang, Search-based software engineering: Trends, techniques and applications, ACM Computing Surveys 45 (1) (2012) 1–61.

[53] Gan: A beginner's guide to generative adversarial networks.
URL https://skymind.ai/wiki/generative-adversarial-network-gan