Web Service API Anti-patterns Detection as a Multi-label Learning Problem

Islem Saidani[1], Ali Ouni[1(✉)], and Mohamed Wiem Mkaouer[2]

[1] Ecode de technologie superieure, University of Quebec, Montreal, QC, Canada
islem.saidani@ens.etsmtl.ca, ali.ouni@etsmtl.ca
[2] Rechester Institute of Technology, Rochester, NY, USA
mwmvse@rit.edu

Abstract. Anti-patterns are symptoms of poor design and implementation solutions applied by developers during the development of their software systems. Recent studies have identified a variety of Web service anti-patterns and defined them as sub-optimal solutions that result from bad design choices, time pressure, or lack of developers experience. The existence of anti-patterns often leads to software systems that are hard to understand, reuse, and discover in practice. Indeed, it has been shown that service designers and developers tend to pay little attention to their service interfaces design. Web service antipatterns detection is a non-trivial and error-prone task as different anti-pattern types typically have interleaving symptoms that can be subjectively interpreted and hence detected in different ways. In this paper, we introduce an automated approach that learns from a set of interleaving Web service design symptoms that characterize the existence of anti-pattern instances in a service-based system. We build a multi-label learning model to detect 8 common types of Web service anti-patterns. We use the ensemble classifier chain (ECC) model that transforms multi-label problems into several single-label problems which are solved using genetic programming (GP) to find the optimal detection rules for each anti-pattern type. To evaluate the performance of our approach, we conducted an empirical study on a benchmark of 815 Web services. The statistical tests of our results show that our approach can detect the eight Web service antipattern types with an average F-measure of 93% achieving a better performance compared to different state-of-the-art techniques. Furthermore, we found that the most influential factors that best characterize Web service anti-patterns include the number of declared operations, the number of port types, and the number of simple and complex types in service interfaces.

Keywords: Web service design · Service interface · Service anti-patterns · Genetic programming · Ensemble classifier chain

1 Introduction

Web services have become a popular technology for deploying scale-out application logic and are used in both open source and industry software projects such

W.-S. Ku et al. (Eds.): ICWS 2020, LNCS 12406, pp. 114–132, 2020.
https://doi.org/10.1007/978-3-030-59618-7_8

as Amazon, Yahoo, Fedex, Netflix, and Google. An advantage of using Web services and Service-Based Systems (SBS) is their loose coupling, which leads to agile and rapid evolution, and continuous re-deployment. Typically, SBSs use of a collection Web services that communicate by messages through declared operations in the services interfaces (API).

Being the most used implementation of the Service Oriented Architecture (SOA), Web services are based on a number of widely acknowledged design principles, qualities and structural features that are different from traditional systems [2,20,29,30]. While there is no generalized recipe for what is considered to be a good service design, there exists guidelines about how to develop service-oriented designs while following a set of quality principles like service reusability, flexibility, and loose coupling principles [10,20,29]. However, like any software system, Web service must evolve to add new user requirements, fix defects or adapt to new environment changes. Such frequent changes, as well as other business factors, developers expertise, and deadline pressure may, in turn, lead to the violation of design quality principles. The existence of bad programming practices, inducing poor design, also called "*anti-patterns*" or "*design defects*", are an indication of such violations [20,21,25]. Such antipatterns include the *God Object Web service* which typically refers to a Web service with large interface implementing a multitude of methods related to different technical and business abstractions. The *God Object Web Service* is not easy to discover and reuse and often unavailable to end users because it is overloaded [12]. Moreover, when many clients are utilizing one interface, and several developers work on one underlying implementation, there are bound to be issues of breakage in clients and developer contention for changes to server-side implementation artifacts [12]. To this end, such anti-patterns should be detected, prevented and fixed in real world SBS to adequately fit in the required system's design with high QoS [12,29].

While recent works attempted to detect and fix Web service antipatterns [18–21,25,33], the detection of such antipatterns is still a challenging and difficult task. Indeed, there is no consensual way to translate formal definition and symptoms into actionable detection rules. Some efforts attempted to manually define detection rules [18,19,25]. However, such manual rules are applied, in general, to a limited scope and require a non-trivial manual effort and human expertise to calibrate a set of detection rules to match the symptoms of each antipattern instance with the actual characteristics of a given Web service. Other approaches attempted to use machine learning to better automate the detection of antipatterns [20,21,32]. However, these detection approaches formulated to the detection as a single-label learning problem, *i.e.*, dealing with antipatterns independently and thus ignoring the innate relationships between the different antipattern symptoms and characteristics. As a result, existing machine learning-based approaches lead to several false positives and true negatives reducing the detection accuracy. Indeed, recent studies showed that different types of Web service antipatterns may exhibit similar symptoms and can thus co-exist in the same Web service [12,20,25]. That is, similar symptoms can be used to characterize multiple antipattern types making their identification even harder and

error-prone [12,20]. For example, the *God Object Web service* (GOWS) antipattern is typically associated with the *Chatty Web service* (CWS) antipattern which manifests in the form of a Web service with a high number of operations that are required to complete abstraction. Consequently, the GOWS and the CWS typically co-occur in Web services. Inversely, the GOWS antipattern has different symptoms than the fine-grained Web service (FGWS) antipattern which typically refers to a small Web service with few operations implementing only apart of an abstraction. Hence, knowing that a Web service is detected as a FGWS antipattern, it cannot be a GOWS or a CWS as they have different innate characteristics/symptoms.

In this paper, our aim is to provide an automated and accurate technique to detect Web service anti-patterns. We formulate the Web services antipatterns detection problem as a multi-label learning (MLL) problem to deal with the interleaving symptoms of existing Web service antipatterns by generating multiple detection rules that can detect various antipattern types. We use the ensemble classifier chain (ECC) technique [28] that converts the detection task of multiple antipattern types into several binary classification problems for each individual antipattern type. ECC involves the training of n single-label binary classifiers, where each one is solely responsible for detecting a specific label, *i.e.*, antipattern type. These n classifiers are linked in a chain, such that each binary classifier is able to consider the labels identified by the previous ones as additional information at the classification time. For the binary classification, we exploit the effectiveness of genetic programming (GP) [13,14,20] to find the optimal detection rules for each antipattern. The goal of GP is to learn detection rules from a set of real-world instances of Web service antipatterns. In fact, we use GP to translate regularities and symptoms that can be found in real-world Web service antipattern examples into actionable detection rules. A detection rule is a combination of Web service interface quality metrics with their appropriate threshold values to detect various types of antipatterns.

We implemented and evaluated our approach on a benchmark of 815 Web services from different application domains and sizes and eight common Web service antipattern types. To evaluate the performance of our GP-ECC approach, and the statistical analysis of our results show that the generated detection rules can identify the eight considered antipattern types with an average precision of 89%, and recall of 93% and outperforms state-of-the-art techniques [20,25]. Moreover, we conducted a deep analysis to investigate the symptoms, *i.e.*, features, that are the best indicators of antipatterns. We found that the most influential factors that best characterize Web service anti-patterns include the number of declared operations, the number of port types, and the number of simple and complex types in service interfaces.

This paper is structured as follows: the paper's background is detailed in Sect. 2. Section 3 summarizes the related studies. In Sect. 4, we describe our GP-ECC approach for Web service antipatterns detection. Section 5 presents our experimental evaluation, and discusses the obtained results. Section 6 discusses potential threats to the validity our our approach. Finally, Sect. 7 concludes and outlines our future work.

2 Background

This section describes the basic concepts used in this paper.

2.1 Web Service Anti-Patterns

Anti-patterns are symptoms of bad programming practices and poor design choices when structuring the web interfaces.They typically engender web interfaces to become harder to maintain and understand [17].

Various types of antipatterns, characterized by how they hinder the quality of service design, have been recently introduced with the purpose of identifying them, in order to suggest their removal through necessary refactorings [12,16,25]. Typical web service antipatterns are described in Table 1:

Table 1. The list of considered Web service antipatterns.

Antipatterns definitions
Chatty Web service (CWS): is a service where a high number of operations, typically attribute-level setters or getters, are required to complete one abstraction. This antipattern may have many fine-grained operations, which degrades the overall performance with higher response time [12]
Fine grained Web service (FGWS): is a too fine-grained service whose overhead (communications, maintenance, and so on) outweighs its utility. This defect refers to a small Web service with few operations implementing only a part of an abstraction. It often requires several coupled Web services to complete an abstraction, resulting in higher development complexity, reduced usability [12]
God object Web service (GOWS): implements a multitude of methods related to different business and technical abstractions in a single service. It is not easily reusable because of the low cohesion of its methods and is often unavailable to end users because it is overloaded [12]
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). Ambiguous names are not semantically and syntactically sound and affect the service discoverability and reusability [19]
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. A DWS usually deals with very small messages of primitive types and may have high data cohesion [25]
CRUDy Interface (CI): is a service with RPC-like behavior by declaring create, read, update, and delete (CRUD) operations, e.g., `createX()`, `readY()`, etc. Interfaces designed in that way might be chatty because multiple operations need to be invoked to achieve one goal. In general, CRUD operations should not be exposed via interfaces [12]
Redundant PortTypes (RPT): is a service where multiple portTypes are duplicated with the similar set of operations. Very often, such portTypes deal with the same messages. RPT antipattern may negatively impact the ranking of the Web Services [12]
Maybe It is Not RPC (MNR): is an antipattern where the Web service mainly provides CRUD- type operations for significant business entities. These operations will likely need to specify a significant number of parameters and/or complexity in those parameters. This antipattern causes poor system performance because the clients often wait for the synchronous responses [12]

We focus our study on these eight antipattern types as they are the most common ones in SBSs based on recent studies [15,16,21,22,25,30,33].

2.2 Multi-label Learning

Multi-label learning (MLL) is the machine learning task of automatically assigning an object into multiple categories based on its characteristics [5,28,31]. Single-label learning is limited by one instance with only one label. MLL is a non-trivial generalization by removing the restriction and it has been a hot topic in machine learning [5]. MLL has been explored in many areas in machine learning and data mining fields through classification techniques. There exists different MLL techniques [3,5,28,31,35] including (1) problem transformation methods and algorithms, *e.g.*, the classifier chain (CC) algorithm, the binary relevance (BR) algorithm, label powerset (LP) algorithm, and (2) algorithm adaptation methods such as the K-Nearest Neighbors (ML.KNN), as well as (3) ensemble methods such as the ensemble classifier chain (ECC), and random k-labelset (RAKEL).

The Classifier Chain (CC) Model. The CC model combines the computational efficiency of the BR method while still being able to take the label dependencies into account for classification. With BR, the classifier chains method involves the training of q single-label binary classifiers and each one will be solely responsible for classifying a specific label $l_1, l_2, ..., l_q$. The difference is that, in CC, these q classifiers are linked in a chain $\{h_1 \rightarrow h_2 \rightarrow ... \rightarrow h_q\}$ through the feature space. That is, during the learning time, each binary classifier h_j incorporates the labels predicted by the previous $h_1, ..., h_{j-1}$ classifiers as additional information. This is accomplished using a simple trick: in the training phase, the feature vector x for each classifier h_j is extended with the binary values of the labels $l_1, ..., l_{j-1}$.

The Ensemble Classifier Chain (ECC) Model. One of the limitation of the CC model is that the order of the labels is random. This can lead may lead to a single standalone CC model be poorly ordered. Moreover, there is the possible effect of error propagation along the chain at classification time, when one (or more) of the first classifiers predict poorly [28]. Using an ensemble of chains, each with a random label order, greatly reduces the risk of these events having an overall negative effect on classification accuracy. A majority voting method is used to select the best model. Moreover, a common advantage of ensembles is their performance in increasing overall predictive performance [3,28].

2.3 Genetic Programming

Genetic Programming (GP) [13], a sub-family of Genetic Algorithms (GA), is a computational paradigm that were inspired by the mechanics of natural evolution, including survival of the fittest, reproduction, and mutation. GP begins with a set of random population of candidate solutions, also called individuals or chromosomes. Each individual of the population, is represented in the form of a

computer program or tree and evaluated by a fitness function to quantitatively measure its ability to solve the target problem.

In this paper, we apply GP to the problem of Web service antipatterns detection. Hence, we show how GP can effectively explore a large space of solutions, and provide intelligible detection rules with ECC. Also, we bridge the gap between MLL and GP based on the ECC method to solve the problem of antipatterns detection, where each Web service may contain different interleaving antipatterns, *e.g.*, GOWS, CWS and CI. For the binary labels, our ECC model adopts GP to learn detection rules for each antipattern type.

3 Related Work

Detecting and specifying antipatterns in SOA and Web services is a relatively new field. The first book in the literature was written by Dudney et al. [12] 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 [30]. Furthermore, Král et al. [16] 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, Palma et al. [25] have proposed a rule-based approach called SODA-W that 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. [29] and Mateos et al. [19] 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. Mateos et al. [18] have proposed an interesting approach towards generating WSDL documents with less antipatterns using text mining techniques. Ouni et al. [20,22] proposed a search-based approach based on evolutionary computation to find regularities, from examples of Web service antipatterns, to be translated into detection rules. However, detections rules based approaches tend to have a higher number of false positives. Ouni et al. [21] introduced a machine learning based approach to build detection models for different Web service antipattern types. However, the major limitation of the current approaches is that deal with the Web service antipatterns problem as a single label learning problem ignoring the valuable information related to the shared symptoms between different antipattern types. As a consequence, they suffer from reduced accuracy related to several false positives and true negatives.

To fix such antipatterns, Daagi et al. [9] proposed an automated approach based on formal concept analysis to fix the GOWS antipattern. Ouni et al. [23,24] introduced a hybrid approach based on graph partitioning and search based optimization to improve the design quality of web service interfaces to reduce coupling and increase cohesion. Later, Wang et al. [33] have formulated an interactive approach to find the optimal design of Web service and reduce the number of antipatterns.

4 Approach

In this section, we provide the problem formulation for Web service antipatterns detection as a MLL problem. Then, we describe the details of our approach.

4.1 Problem Formulation

We define the Web services antipatterns detection problem as a multi-label learning problem. Each antipattern type is denoted by a label l_i. A MLL problem can be formulated as follows. Let $X = R^d$ denote the input feature space. $L = \{l_1, l_2, ...l_q\}$ denote the finite set of q possible labels, *i.e.*, antipattern types. Given a multi-label training set $D = \{(x_1, y_1), (x_2, y_2), ...(x_N, y_N)\}(x_i \in X, y_i \subseteq L)$, the goal of the multi-label learning system is to learn a function $h : X \rightarrow 2^L$ from D which predicts a set of labels for each unseen instance based on a set of known data.

4.2 Approach Overview

The main goal of our approach is to generate a set of detection rules for each antipattern type while taking into consideration the dependencies between the different antipatterns and their interleaving symptoms. Figure 1 presents an overview of our approach to generate service antipatterns detection rules using the GP-ECC model. Our approach consists of two phases: *training* phase and *detection* phase. In the training phase, our goal is to build an ensemble classifier chain (ECC) model learned from real-world antipattern instances identified from existing Web services based on several GP models for each individual antipattern. In the detection phase, we apply this model to detect the proper set of labels (*i.e.*, antipattern type) for a new unlabeled data (*i.e.*, a new Web service).

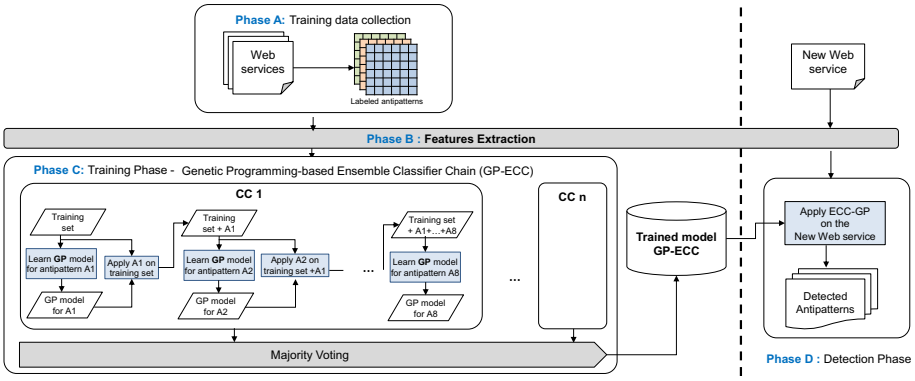

Fig. 1. The Web service antipatterns detection framework using GP-ECC.

Our approach takes as inputs a set of Web services with known labels, *i.e.*, antipatterns (phase A). Then, extracts a set of features characterizing the considered antipattern types from which a GP algorithm will learn (phase B). Next, an ECC algorithm will be built (phase C). The ECC algorithm consists of a set of classifier chain models (CC), each with a random label order. Each CC model, learns eight individual GP models for each of the eight considered antipattern types. The i^{th} binary GP detector will learn from the training data while considering the existing i already detected antipatterns by the $i-1$ detected antipatterns to generate the optimal detection rule that can detect the current i^{th} antipattern. In total, the ECC trains n multi-label CC classifiers $CC_1, ..., CC_n$; each classifier is given a random chain ordering; each CC builds 8 binary GP models for each antipattern type. Each binary model uses the previously predicted binary labels into its feature space. Then, our framework searches for the near optimal GP-ECC model from these n multi-label chain classifiers using an ensemble majority voting schema based on each label confidence [28].

In the detection phase, the returned GP-ECC model is a machine learning classifier that assigns multiple labels, *i.e.*, antipattern types, to a new Web service based on its current features, *i.e.*, its symptoms (phase D). In the next subsections, we provide the details of each phase.

4.3 Phase A : Training Data Collection

Our proposed technique leverages knowledge from a set of examples containing real world instances of web service antipatterns. The base of examples contains different web service antipatterns from different application domains (*e.g.*, social media, weather, online shopping, etc.), which were gathered from various Web service online repositories and search engines, like ProgrammableWeb, and ServiceXplorer, etc. To ensure the correctness of our dataset, the studied antipattern instances were manually inspected and verified according to existing guidelines from the literature [12,16,20,25]. Our dataset is publicly available [1].

4.4 Phase B : Features Extraction Module

The proposed techniques leverages a set of popular and widely used metrics related to web services [20–22,25,27,32,34]. As shown in Table 2, our technique develops its detection rules using suite of over 42 quality metrics including (*i*) code level metrics, (*ii*) WSDL interface metrics, and (*iii*) measurements of performance. Code metrics are calculating using service Java artefacts, being mined using the Java™ API for XML Web Services (JAX-WS)[1] [8] as well as the *ckjm* tool[2] (Chidamber & Kemerer Java Metrics) [6]. WSDL metrics capture any design properties of Web services, in the structure of the WSDL interface level. Furthermore, a set of dynamic metrics is also captured, using web service invocations, *e.g.*, availability, and response time.

[1] http://docs.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html.
[2] http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/.

4.5 Phase C : Multi-label Learning Using GP-ECC

As outlined in the previous Sect. 4.2, our solution uses the ECC classifier [28] to model the multi-label learning task into multiple single-label learning tasks. Our multi-label ECC is a detection model for the identification of web service antipatterns. Each classifier chain (CC) represents a GP model (binary decision tree) for each smell type while considering the previously detected smells (if any), *i.e.*, each binary GP model uses the previously predicted binary labels into its feature space. The motivation behind the choice of GP-based models is driven by its efficiency in the resolution of similar software engineering problems such as, design defects, and code smells identification [3,14,20,22].

In our approach, we adopted the Non-dominated Sorting Genetic Algorithm (NSGA-II) [11] as a search algorithm to generate antipatterns detection rules. NSGA-II is a powerful and widely-used evolutionary algorithm which extends the generic model of GP learning to the space of programs. Unlike other evolutionary search algorithms, in our NSGA-II adaptation, solutions are themselves programs following a tree-like representation instead of fixed length linear string formed from a limited alphabet of symbols [13]. More details about NSGA-II can be found in Deb et al. [11].

We describe in the following subsections the three main adaptation steps: (*i*) solution representation, (*ii*) the generation of the initial generation (*iii*) fitness function, and (*iv*) change operators.

(*i*) Solution Representation. A solution consists of a rule that can detect a specific type of anti-pattern in the form of:

IF (Combination of metrics and their thresholds) **THEN** antipattern type.

In MOGP, the solution representation is a tree-based structure of functions and terminals. Terminals represent various structural, dynamic, and service oriented metrics, extracted from Table 2. Functions are logic operators such as OR (union), AND (intersection), or XOR (eXclusive OR). Thus, each solution is encoded as a binary tree with leafnodes (terminals) correspond to one of metrics described in Table 2 and their associated threshold values randomly generated. Internal-nodes (functions) connect sub-tress and leaves using the operators set $C = \{AND, OR, XOR\}$. Figure 2 is a simplified illustration of a given solution.

(*ii*) Generation of the Initial Population. The initial population of solutions is generated randomly by assigning a variety of metrics and their thresholds to the set of different nodes of the tree. The size of a solution, *i.e.*, the tree's length, is randomly chosen between lower and upper bound values. These two bounds have determined and called the problem of bloat control in GP, where the goal is to identify the tree size limits. Thus, we applied several trial and error experiments using the HyperVolume (HP) performance indicator [13] to determine the upper bound after which, the sign remains invariant.

(*iii*) Fitness Function. The fitness function evaluates how good is a candidate solution in detecting web service antipatterns. Thus, to evaluate the fitness

Table 2. List of Web service quality metrics used.

Metric	Description	Metric level
Service interface metrics		
NPT	Number of port types	Port type
NOD	Number of operations declared	Port type
NCO	Number of CRUD operations	Port type
NOPT	Average number of operations in port types	Port type
NPO	Average number of parameters in operations	Operation
NCT	Number of complex types	Type
NAOD	Number of accessor operations declared	Port type
NCTP	Number of complex type parameters	Type
COUP	Coupling	Port type
COH	Cohesion	Port type
NOM	Number of messages	Message
NST	Number of primitive types	Type
ALOS	Average length of operations signature	Operation
ALPS	Average length of port types signature	Port type
ALMS	Average length of message signature	Message
RPT	Ratio of primitive types over all defined types	Type
RAOD	Ratio of accessor operations declared	Port type
ANIPO	Average number of input parameters in operations	Operation
ANOPO	Average number of output parameters in operations	Operation
NPM	Average number of parts per message	Message
AMTO	Average number of meaningful terms in operation names	Operation
AMTM	Average number of meaningful terms in message names	Message
AMTP	Average number of meaningful terms in port type names	Type
Service code metrics		
WMC	Weighted methods per class	Class
DIT	Depth of Inheritance Tree	Class
NOC	Number of Children	Class
CBO	Coupling between object classes	Class
RFC	Response for a Class	Class
LCOM	Lack of cohesion in methods	Class
Ca	Afferent couplings	Class
Ce	Efferent couplings	Class
NPM	Number of Public Methods	Class
LCOM3	Lack of cohesion in methods	Class
LOC	Lines of Code	Class
DAM	Data Access Metric	Class
MOA	Measure of Aggregation	Class
MFA	Measure of Functional Abstraction	Class
CAM	Cohesion Among Methods of Class	Class
AMC	Average Method Complexity	Method
CC	The McCabe's cyclomatic complexity	Method
Service Performance Metrics		
RT	Response Time	Method
AVL	Availability	Service

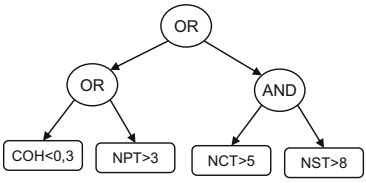

Fig. 2. A simplified example of a solution for GOWS antipattern.

of each solution, we use two objective functions, based on two well-known metrics [14,20], to be optimized, *i.e.*, precision and recall. The precision objective function aims at maximizing the detection of correct antipatterns over the list of detected ones. The recall objective function aims at maximizing the coverage of expected antipatterns from the base of examples over the actual list of detected instances. Precision and recall of a solution S are defined as follows.

$$Precision(S) = \frac{|\{\text{Detected antipatterns}\} \cap \{\text{Expected antipatterns}\}|}{|\{\text{Detected antipatterns}\}|} \quad (1)$$

$$Recall(S) = \frac{|\{\text{Detected antipatterns}\} \cap \{\text{Expected antipatterns}\}|}{|\{\text{Expected antipatterns}\}|} \quad (2)$$

(*iv*) **Change Operators.** Crossover and mutation are used as change operators to evolve candidate solutions towards optimality.

Crossover. We adopt the "standard" random, single-point crossover. It selects two parent solutions at random, then picks a sub-tree on each one. Then, the crossover operator swaps the nodes and their relative subtrees from one parent to the other. Each child thus combines information from both parents.

Mutation. The mutation operator aims at slightly changing a solution, without losing its main properties. It can be applied for both function or terminal nodes. For a given solution to mutated, one or multiple nodes are randomly selected, then changed according to their type. For functions, a logic operator can be replaced with any randomly selected logic operator, while for terminals, metrics may be swapped with another metric, or a metric threshold can be changed.

ECC Majority Voting. As shown in Fig. 1, for each CC, MOGP will generate an optimal rule for each type of web service antipattern, *i.e.*, binary detection. Then, ECC allows to find the best CC that provides the best MLL from all the trained binary models. Each CC_i model is likely to be unique and able to achieve different multi-label classifications. These classifications are summed by label so that each label receives a number of votes. A threshold is used to select the most popular labels which form the final predicted multi-label set. This is a generic voting scheme used in MLL transformation methods [28].

4.6 Phase D: Detection Phase

After constructing the GP-ECC model in the training phase, it will be then used to detect a set of labels for a new Web service. It takes as input the set of features

extracted from a given Web service using the feature extraction module. As output, it returns the detection results for each individual label, *i.e.*, antipattern.

5 Empirical Study

In this section, we describe our empirical study to evaluate our GP-ECC approach. We report the research questions, the experimental setup, and results.

5.1 Research Questions

We designed our empirical study to answer the three following research questions.

- **RQ1: (Performance)** How accurately can our GP-ECC approach detect Web service anipatterns?
- **RQ2: (Sensitivity)** What types of anipatterns does our GP-ECC approach detect correctly?
- **RQ3: (Features influence)** What are the most influential features that can indicate the presence of anipatterns?

5.2 Experimental Setup

We evaluate our approach on a benchmark of 815 Web services [1]. Table 3 summarizes the experimental dataset. Furthermore, as a sanity check, all antipatterns were manually inspected and validated based on literature guidelines [12,30] as discussed in Sect. 4.3. Web services were collected from different Web service search engines including eil.cs.txstate.edu/ServiceXplorer, programmableweb.com, biocatalogue.org, webservices.seekda.com, taverna.org.uk and myexperiment.org. Furthermore, for better generalisability, our empirical study, our collected Web services are drawn from 9 different application domains, *e.g.*, financial, science, search, shipping, etc.

We considered eight common types of Web service antipatterns, *i.e.*, *god object Web service* (GOWS), *fine-grained Web service* (FGWS), *chatty Web service* (CWS), *data Web service* (DWS), *ambiguous Web service* (AWS), *redundant port types* (RPT), *CRUDy interface* (CI), and *Maybe It is Not RPC* (MNR), (cf. Sect. 2). In our experiments, we conducted a 10-fold cross-validation procedure to split our data into training data and evaluation data.

To answer **RQ1**, we conduct experiments to justify our GP-ECC approach.

Baseline Learning Methods. We first compare the performance of our meta-algorithm ECC. We used GP, *decision tree* (J48) and *random forest* (RF) as corresponding basic classification algorithms. We also compared with the widely used MLL algorithm adaptation method, *K-Nearest Neighbors* (ML.KNN). Thus, in total, we have 4 MLL algorithms to be compared. One fold is used for the test and 9 folds for the training.

Table 3. The list of Web services used in our evaluation.

Category	# of services	# of antipatterns
Financial	185	115
Science	52	18
Search	75	33
Shipping	58	23
Travel	105	49
Weather	65	21
Media	82	19
Education	55	28
Messaging	63	43
Location	75	39
All	**815**	**388**

State-of-the-Art Detection Methods. Moreover, we compare the performance of our approach with two state-of-the-art approaches, SODA-W [25] and P-EA [20] for Web service antipattern detection. The SODA-W approach of Palma et al. [25] manually translates antipattern symptoms into detection rules and algorithms based on a literature review of Web service design. P-EA [20] adopts parallel GP technique to detect Web service antipatterns based on a set of Web service interface metrics. Both approaches detect antipattern types in an independent manner.

To compare the performance of each method, we use common performance metrics, *i.e.*, precision, recall, and F-measure [14,20,28]. Let l a label in the label set L. For each instance i in the antipatterns learning dataset, there are four outcomes, True Positive (TP_l) when i is detected as label l and it correctly belongs to l; False Positive (FP_l) when i is detected as label l and it actually does not belong to l; False Negative (FN_l) when i is not detected as label l when it actually belongs to l; or True Negative (TN_l) when i is not detected as label l and it actually does not belong to l. Based on these possible outcomes, precision (P_l), recall (R_l) and F-measure (F_l) for label l are defined as follows:

$$P_l = \frac{TP_l}{TP_l + FP_l} \quad ; \quad R_l = \frac{TP_l}{TP_l + FN_l} \quad ; \quad F_l = \frac{2 \times P_l \times R_l}{P_l + R_l}$$

Then, the average precision, recall, and F-measure of the $\mid L \mid$ labels are calculated as follows:

$$Precision = \frac{1}{\mid L \mid} \sum_{l \in L} P_l \quad ; \quad Recall = \frac{1}{\mid L \mid} \sum_{l \in L} R_l \quad ; \quad F1 = \frac{1}{\mid L \mid} \sum_{l \in L} F_l$$

Statistical Test Methods. To compare the performance of each method, we perform Wilcoxon pairwise comparisons [7] at 99% significance level (*i.e.*, $\alpha =$

0.01) to compare GP-ECC with each of the 9 other methods. We also used the non-parametric effect Cliff's delta (d) [7] to compute the effect size. The effect size d is interpreted as Negligible if $\mid d \mid < 0.147$, Small if $0.147 \leq \mid d \mid < 0.33$, Medium if $0.33 \leq \mid d \mid < 0.474$, or High if $\mid d \mid \geq 0.474$.

To answer **RQ2**, we investigated the antipattern types that were detected to find out whether there is a bias towards the detection of specific types.

To answer **RQ3**, we aim at identifying the features that are the most important indicators of whether a Web service has a given antipattern or not. For each antipattern type, we count the percentage of rules in which the feature appears across all obtained optimal rules by GP. The more a feature appears in the set of optimal trees, the more the feature is relevant to characterize that antipattern.

5.3 Results

Results for RQ1 (Performance). Table 4 reports the average precision, recall and F-measure scores for the compared methods. We observe that we see that ECC performs well with GP as a base method as compared to J48 and RF. We used GP-ECC as the base for determining statistical significance. In particular, the GP-ECC method achieves the highest F-measure with 0.91 compared to the J48 and RF methods achieving an F-measure of 0l9 and 0.89, respectively, with medium and large effect sizes. The same performance was achieved by GP-ECC in terms of precision and recall, with 0.89 and 0.93, respectively. The statistical analysis of the obtained results confirms thus the suitability of the GP formulation compared to decision tree and random forest algorithms. We can also see overall superiority for of the ECC and in particular the GP-ECC compared to the transformation method ML.KNN in terms of precision, recall and F-measure with large effect size. One of the reasons that ML.KNN does not perform well is that it ignores the label correlation, while ECC consider the label correlation by using an ensemble of classifiers. Moreover, among the 3 base learning algorithms, GP performs the best, followed by J48 and RF.

Moreover, we observe from Table 4 that GP-ECC achieved a higher superiority than both state-of-the-art approaches, P-EA and SODA-W. While P-EA achieves promising results with an average F-measure of 83%, it is still less than GP-ECC. Moreover, SODA-W achives an F-measure of 72% which lower than other approaches. We conjecture that a key problem with P-EA and SODA-W is that they detect separately possible antipatterns without looking at the relationship between them. Through a closer manual inspection of the false positive and true negative instances by P-EA and SODA-W, we found a number of Web services that are detected at the same time as god object Web services (GOWS) and fine-grained Web services (FGWS) which would reduce the overall accuracy as GOWS and FGWS cannot co-occur in the same Web service. Other missing chatty Web service (CWS) instances were identified in Web services that are detected as GOWS. Indeed, GP-ECC makes the hidden relationship between antipatterns more explicit which has shown higher accuracy.

Table 4. The achieved results by ECC with the base algorithms GP, J48, and RF; ML.KNN; and existing approaches SODA-W and P-EA.

Approach	Precision		Recall		F1	
	Score	p-value (d)*	Score	p-value (d)*	Score	p-value (d)*
GP-ECC	**0.89**	-	**0.93**	-	**0.91**	-
J48-ECC	0.87	<0.01 (M)	0.9	<0.01 (M)	0.88	<0.01 (M)
RF-ECC	0.86	<0.01 (M)	0.89	<0.01 (M)	0.87	<0.01 (M)
ML.KNN	0.83	<0.01 (L)	0.84	<0.01 (L)	0.83	<0.01 (L)
P-EA	0.82	<0.01 (L)	0.85	<0.01 (L)	0.83	<0.01 (L)
SODA-W	0.7	<0.01 (L)	0.74	<0.01 (L)	0.72	<0.01 (L)

*p-value(d) reports the statistical difference (p-value) and effect-size (d) between GP-ECC and the algorithm/approach in the current row.
The effect-size (d) is N : Negligible − S : Small − M : Medium − L : Large

Results for RQ2 (Sensitivity). Figure 3 reports the sensitivity analysis of each specific antipattern type. We observe that GP-ECC does not have a bias towards the detection of any specific antipattern type. As shown in the figure, GP-ECC achieved good performance and low variability in terms of the median F-measure, ranging from 87% to 93%, across the 8 considered antipattern types. The highest F-measure was obtained for the god object (GOWS) and fine-grained (FGWS) antipatterns (93%) which heavily relies on the notion of size. This higher performance is reasonable since the existing guidelines [12,30] rely heavily on the notion of size in terms of declared operations, port types, and simple/complex data types used. But for antipatterns such as the ambiguous Web service (AWS), the notion of size is less important, it rather relies an the meaningfulness and length of operations and messages identifiers. This aspect makes this type of antipatterns hard to detect using such information as it often depends on human interpretations.

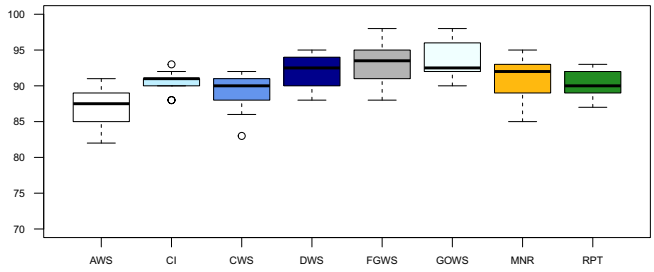

Fig. 3. F-measure achieved by GP-ECC for each antipattern across all categories.

Results for RQ3 (Features Influence). To better understand what features, *i.e.*, metrics, are most used by our GP-ECC model to generate detection rules

among all the generated rules, we count the percentage of rules in which the feature appears. Table 5 shows the statistics for each smell type with the top-10 features (cf. Table 2), from which the three most influencing features values are in bold. We observe that the number of operations declared (NOD), the number of messages (NOM), the number of simple and complex types (NST and NCT), and the cohesion (COH) are the most influencing parameters. Other features such as the number of input parameters in operations (NIPO) and the coupling (COUP) are also influencing the existence of antipatterns. We also found that some features such the average length of operation signatures (ALOS) are specific to the ambiguous Web service (AWS) antipattern and do not participate to characterize any of the other considered antipattern types.

Table 5. The most influential features for each antipattern.

	GOWS	FGWS	CWS	DWS	AWS	RPT	CI	MNR
NOD	**98**	**100**	91	86	52	**96**	**93**	83
NOM	**92**	**90**	**100**	**92**	55	**91**	89	**93**
COH	**89**	84	89	87	23	**92**	**92**	85
WMC	82	82	81	65	45	81	72	83
NIPO	79	75	89	84	**92**	61	**93**	**98**
NCT	81	85	91	93	41	54	86	91
NST	**89**	**86**	**96**	**96**	**82**	32	80	93
ALOS	34	32	41	18	**100**	9	39	23
COUP	76	69	**93**	82	71	52	89	**100**
LCOM	88	77	88	85	46	81	86	79

We thus observe that different interface service level measures play a crucial role in the emergence of antipatterns, while those related to the source code are less influencial. These findings suggest that more attention has to be paid to the design of their service interface to avoid the presence of antipatterns and their impact on the software quality. This finding aligns also with previous research advocating the importance of service interface design [4,12,18,24,26,33]

6 Threats to Validity

Threats to construct validity could be related to the performance measures. We basically used standard performance metrics such as precision, recall and F-measure that are widely accepted in MLL and software engineering [20,25]. Another potential threat could be related to the selection of learning techniques. Although we use the GP, J48 and RF which are known to have high performance, we plan to compare with other ML techniques in our future work.

Threats to internal validity relate to errors in our experiments. Our approach relies on the used metrics to characterize antipatterns. We mitigated this issues by using popular and well-accepted metrics and tools to neasure our metrics.

Threats to external validity relate to the generalizability of our results. Our approach relies on learning from existing services, and so, their diversity is critical for our learning process. We mitigated this threat by choosing independent services, issued from different providers, and they were also developed in multiple application domains. Also, our training set was manually validated, however, such human activity is prone to error sand personal bias. The reduction of such bias can be achieved by following existing literature gidelines [12,16] randomly choosing a statistically significant sample that is reclassified by the three authors. Then, the kappa agreement is calculated and its corresponding score is 0.83, which is considered a high score for inter-rater agreement [7].

7 Conclusion and Future Work

Web service antipatterns are symptoms of potential problems threatening the longevity of services. Although such antipatterns can facilitate the coding the quick delivery of services, their long-term impact hinders the maintainability and evolvability of services. This paper developed a novel technique, leveraging an existing set of manually verified antipatterns, to develop a metric-based detection rules using ensemble classifier chain. We transform multi-label problems into several single-label problems that are solved using the genetic programming. Our experiments show the effectiveness of our detection strategy by achieving an F-Measure of 93%, when analyzing a large set of 815 web services.

As part of our future investigations, we plan on extending the set of metrics we used as well as other RESTFul Web services, in order to explore potential features, which may further improve the accuracy of our detection strategy.

References

1. Replication package (2020). https://github.com/WS-antipatterns/dataset
2. Almarimi, N., Ouni, A., Bouktif, S., Mkaouer, M.W., Kula, R.G., Saied, M.A.: Web service api recommendation for automated mashup creation using multi-objective evolutionary search. Appl. Soft Comput. **85**, 105830 (2019)
3. Almarimi, N., Ouni, A., Chouchen, M., Saidani, Islem, M.M.W.: On the detection of community smells using genetic programming-based ensemble classifier chain. In: International Conference on Global Software Engineering, pp. 1–12 (2020)
4. Boukharata, S., Ouni, A., Kessentini, M., Bouktif, S., Wang, H.: Improving web service interfaces modularity using multi-objective optimization. Automated Softw. Eng. **26**(2), 275–312 (2019). https://doi.org/10.1007/s10515-019-00256-4
5. de Carvalho, A.C.P.L.F., Freitas, A.A.: A Tutorial on Multi-label Classification Techniques, pp. 177–195 (2009)
6. Chidamber, S.R., Kemerer, C.F.: A metrics suite for object oriented design. IEEE Trans. Softw. Eng. **20**(6), 476–493 (1994)

7. Cohen, J.: Statistical power analysis for the behavioral sciences. Academic Press (1988)
8. Coscia, J.L.O., Crasso, M., Mateos, C., Zunino, A.: Estimating web service interface quality through conventional object-oriented metrics. CLEI E. **16**(1) 2056–2101 (2013)
9. Daagi, M., Ouni, A., Kessentini, M., Gammoudi, M.M., Bouktif, S.: Web service interface decomposition using formal concept analysis. In: IEEE International Conference on Web Services (ICWS), pp. 172–179 (2017)
10. Daigneau, R.: Service Design Patterns: fundamental design solutions for SOAP/WSDL and restful Web Services. Addison-Wesley (2011)
11. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. Evol. Comput. **6**(2), 182–197 (2002)
12. Dudney, B., Krozak, J., Wittkopf, K., Asbury, S., Osborne, D.: J2EE Antipatterns. Wiley, Hoboken (2003)
13. John, R., Koza, M.: Genetic programming: On programming computers by means of natural selection and genetics. In: Association for Computing Machinery, MIT Press, Cambridge (1992)
14. Kessentini, M., Ouni, A.: Detecting android smells using multi-objective genetic programming. In: IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 122–132 (2017)
15. Král, J., Žemlička, M.: Crucial service-oriented antipatterns. Int. J. Adv. Softw. **2**(1), 160–171 (2009)
16. Král, J., Zemlicka, M.: Popular SOA Antipatterns. In: Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, pp. 271–276 (2009)
17. Marinescu, R.: Detection strategies: metrics-based rules for detecting design flaws. In: 2013 IEEE International Conference on Software Maintenance, pp. 350–359 (2004)
18. Mateos, C., Rodriguez, J.M., Zunino, A.: A tool to improve code-first web services discoverability through text mining techniques. Softw. Pract. Experience **45**(7), 925–948 (2015)
19. Mateos, C., Zunino, A., Coscia, J.L.O.: Avoiding WSDL bad practices in code-first web services. SADIO Electron. J. Inform. Oper. Res. **11**(1), 31–48 (2012)
20. Ouni, A., Kessentini, M., Inoue, K., Cinneide, M.O.: Search-based web service antipatterns detection. IEEE Trans. Serv. Comput. **10**(4), 603–617 (2017)
21. Ouni, A., Daagi, M., Kessentini, M., Bouktif, S., Gammoudi, M.M.: A machine learning-based approach to detect web service design defects. In: IEEE International Conference on Web Services (ICWS). pp. 532–539 (2017)
22. Ouni, A., Gaikovina Kula, R., Kessentini, M., Inoue, K.: Web service antipatterns detection using genetic programming. In: Annual Conference on Genetic and Evolutionary Computation (GECCO), pp. 1351–1358 (2015)
23. Ouni, A., Salem, Z., Inoue, K., Soui, M.: SIM: an automated approach to improve web service interface modularization. In: IEEE International Conference on Web Services (ICWS), pp. 91–98 (2016)
24. Ouni, A., Wang, H., Kessentini, M., Bouktif, S., Inoue, K.: A hybrid approach for improving the design quality of web service interfaces. ACM Trans. Internet Technol. (TOIT) **19**(1), 1–24 (2018)
25. Palma, F., Moha, N., Tremblay, G., Gueheneuc, Y.G.: Specification and detection of SOA antipatterns in web services. In: Software Architecture, pp. 58–73 (2014)
26. Perepletchikov, M., Ryan, C., Frampton, K., Schmidt, H.: Formalising service-oriented design. J. Softw. **3**(2), 1–14 (2008)

27. Perepletchikov, M., Ryan, C., Tari, Z.: The impact of service cohesion on the analyzability of service-oriented software. IEEE Trans. Serv. Comput. **3**(2), 89–103 (2010)
28. Read, J., Pfahringer, B., Holmes, G., Frank, E.: Classifier chains for multi-label classification. Mach. Learn. **85**(3), 333 (2011)
29. Rodriguez, J.M., Crasso, M., Mateos, C., Zunino, A.: Best practices for describing, consuming, and discovering web services: a comprehensive toolset. Softw. Pract. Experience **43**(6), 613–639 (2013)
30. Rotem-Gal-Oz, A.: SOA Patterns. Manning Publications (2012)
31. Tsoumakas, G., Katakis, I.: Multi-label classification: an overview. Int. J. Data Warehous. Min. **3**(3), 1–13 (2007)
32. Wang, H., Kessentini, M., Ouni, A.: Bi-level identification of web service defects. In: International Conference on Service-Oriented Computing, pp. 352–368 (2016)
33. Wang, H., Kessentini, M., Ouni, A.: Interactive refactoring of web service interfaces using computational search. IEEE Trans. Serv. Comput. **3** 6–12 (2017)
34. Wang, H., Ouni, A., Kessentini, M., Maxim, B., Grosky, W.I.: Identification of web service refactoring opportunities as a multi-objective problem. In: IEEE International Conference on Web Services (ICWS), pp. 586–593 (2016)
35. Zhang, M.L., Zhou, Z.H.: Ml-knn: a lazy learning approach to multi-label learning. Pattern Recogn. **40**(7), 2038–2048 (2007)