Web Service Antipatterns Detection Using Genetic Programming

Ali Ouni[1], Raula Gaikovina Kula[1], Marouane Kessentini[2], Katsuro Inoue[1]

[1]Graduate School of Information Science and Technology, Osaka University, Japan
{ali, raula-k, inoue}@ist.osaka-u.ac.jp

[2]Department of Computer and Information Science, University of Michigan, USA
marouane@umich.edu

ABSTRACT

Service-Oriented Architecture (SOA) is an emerging paradigm that has radically changed the way software applications are architected, designed and implemented. SOA allows developers to structure their systems as a set of ready-made, reusable and compostable services. The leading technology used today for implementing SOA is Web Services. Indeed, like all software, Web services are prone to change constantly to add new user requirements or to adapt to environment changes. Poorly planned changes may risk introducing antipatterns into the system. Consequently, this may ultimately leads to a degradation of software quality, evident by poor quality of service (QoS). In this paper, we introduce an automated approach to detect Web service antipatterns using genetic programming. Our approach consists of using knowledge from real-world examples of Web service antipatterns to generate detection rules based on combinations of metrics and threshold values. We evaluate our approach on a benchmark of 310 Web services and a variety of five types of Web service antipatterns. The statistical analysis of the obtained results provides evidence that our approach is efficient to detect most of the existing antipatterns with a score of 85% of precision and 87% of recall.

Categories and Subject Descriptors

D.2.7 **[Distribution, Maintenance, and Enhancement]:** *Restructuring, reverse engineering, and reengineering*

Keywords

Web services, antipatterns, search-based software engineering.

1. INTRODUCTION

Service Oriented Architecture (SOA) has emerged as the next generation of software systems. As part of the service-oriented computing paradigm, SOA revolutionizes the process of developing and deploying distributed software applications as a set of reusable composable services [1]. SOA provides many architectural benefits including reusability, flexibility, adaptability, and maintainability [1]. This architectural style can be implemented utilizing a variety of SOA technologies, such as Web Services, OSGi, SCA, and REST. Today, Service-Based Systems (SBS) have become prevalent and omnipresent in our everyday life such as Facebook, Dropbox, Google Maps, PayPal, FedEx, and so on.

Web services must be carefully designed and implemented to adequately fit in the required system's design with high QoS [2]. Indeed, there is no generalized recipe for proper service design. A set of guiding quality principles for service-oriented design exist such as service flexibility, operability, composability, and loose coupling principles [1]. However, the design of services is influenced mostly on context and usage [3]. Even though the programmers are familiar with these principles, business factors such as deadline pressures may lead to violations of quality principles. The presence of programming patterns associated with bad design and bad programming practices, known as "antipatterns", are an indication of such violations [4] [5].

Common Web service antipatterns include the nanoservice, and multiservice. Nanoservice is an antipattern where a service is too fine-grained characterized with few low cohesive operations, and whose overhead (communications, maintenance, and so on) outweighs its utility [3]. In contrast, the multiservice antipattern describes about the other extreme, i.e., the largest service. Multiservice corresponds to a god service that contains a large number of very low cohesive operations related to different business logics. Nanoservice and multiservice antipatterns can cause many maintenance and evolution problems such as poor performance, fragmented logic, overhead, client breakages and unavailability.

Consequently, there is a high need for efficient techniques that both Web service users and providers can use to detect and prevent antipatterns in their SBSs. Although there are several tools and techniques to detect antipatterns and code-smells in object-oriented (OO) systems [6] [7] [8], Web service antipatterns detection is not mature enough to provide efficient detection techniques [9] [10]. Indeed, despite the importance and extensive usage of Web services in last years, no automated approach for the detection of such antipatterns in Web services has been proposed.

In this paper, we introduce a novel automated approach for detecting Web service antipatterns. We propose a search-based approach to automatically infer antipattern detection rules from a base of real-world examples of Web service antipatterns. The problem is to find, from a large list of Web service metrics, the best combination of metrics and their appropriate threshold values, for each antipattern type. We thus express antipatterns detection as an optimization problem, using genetic programming (GP) to generate detection rules. A candidate detection rule is expressed as a combination of metrics and their appropriate threshold values; and should detect as much as possible the number of antipatterns from the base of examples. We present an empirical study to evaluate our approach on a benchmark composed of 310 Web services from six different application domains including 136 antipattern instances. We compare our approach to two other popular algorithms and random search. The statistical results reveal that our approach was significantly better than particle swarm optimization, simulated

Table 1. Web service antipattern definitions.

Antipattern	Definition
Multiservice	Also called god object Web service, represents a service implementing a multitude of methods related to different business and technical abstractions. This service aggregates too many methods into a single service, and it is not easily reusable because of the low cohesion of its methods and is often unavailable to end-users because it is overloaded [16].
Nanoservice	is a too fine-grained service whose overhead (communications, maintenance, and so on) outweighs its utility. This antipattern 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 [16]
Chatty Service	represents an antipattern 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 [9].
Data service	an antipattern that 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 Data Web Service usually deals with very small messages of primitive types and may have high data cohesion [10].
Ambiguous Service	is an antipattern where developers use ambiguous or meaningless names for denoting the main elements of interface elements (e.g., port-types, operations, and messages). Ambiguous names are not semantically and syntactically sound and affect the discoverability and the reusability of a Web service [17].

annealing as well as random search with more than 85% of precision and 87% of recall.

The remainder of this paper is organized as follows. Section 2 describes the background and motivation challenges. Section 3 introduces our search-based approach. In section 4, we present and discuss the validation results. Section 5 discusses the different threats to validity. Section 6 surveys the existing work. Finally, we conclude and outline our future research directions in section 7.

2. BACKGROUND

In this section, we provide a brief overview of SOA, Web services and Web service antipatterns. Then, we outline the different problems and challenges that motivate our approach.

2.1 Definitions

SOA is a logical way for designing complex distributed software systems using functionality implemented by third-party providers. In a SOA, the service requester satisfies its specific needs by using services offered by service providers, through published and discoverable interfaces.

Web services is nowadays the fittest and popular technology to implement SOA [11]. According to the W3C (World Wide Web Consortium), a Web Service is defined as "a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artefacts" [12]. Its interface is described as a WSDL (Web service Description Language) document that contains structured information about the Web service's location, its offered operations and the input/output parameters, and so on. The aim of the Web services platform is to provide the required level of interoperability among different applications using predefined web standards.

Antipatterns are symptoms of poor design and implementation practices that describe a bad solution to a recurring design problem. They often lead to negative effects on software quality for which maintenance and evolution become harder [13]. Software engineers often introduce antipatterns unintentionally during the initial design or during software development due to bad design decisions, ignorance or time pressure. Therefore, antipatterns should be detected and removed from the software design as early as possible. Table 1 summaries the common Web service antipatterns including multiservice, nanoservice, chatty service, data service, and ambiguous service. In this paper, We focus mainly on these five antipattern types in our experiments as they are the most frequently occurring ones in SBSs based on recent studies [10] [14] [15].

2.2 Web service antipatterns detection challenges

The most challenging issues when detecting Web service antipatterns is how to find the best metrics that characterize such antipattern, how to find the appropriate threshold values for these metrics; and most importantly how to find the best combination of these metrics. Indeed, most of the existing works are limited to provide definitions to Web service antipatterns and/or characterize their common symptoms towards an antipattern catalog. However, automating the detection of such antipatterns is still a very challenging task.

In recent approaches [10] [18] [19], Web service antipatterns detection relies on declarative rule specification using domain-specific language (DSL). In these settings, rules are manually defined to identify the key symptoms that characterize a Web service antipattern using combinations of mainly quantitative (metrics), structural, and/or lexical information. However, in an exhaustive scenario, the number of possible antipatterns to manually characterize and formulate with rules can be very large. Unfortunately, it is very difficult to find a consensus to characterize and formulate such symptoms. Moreover, even when consensus is met, the same symptom could be associated to many antipattern types, which may compromise the precise identification of antipattern instances. Indeed, translating antipattern definitions from the natural language to metrics is still mainly a subjective task. That is, different antipatterns are characterized by the same metrics.

Another inherent problem is related to the definition of threshold values when dealing with quantitative information. Indeed, there is no general agreement on extreme manifestations of Web service antipatterns [16]. That is, for each antipattern, rules that are expressed in terms of metrics need substantial calibration efforts to find the right threshold value for each metric, above which an antipattern is said to be detected. Since there is no consensus in defining SOA antipatterns, different threshold values should be tested to find the best one. For instance, the multiservice detection involves information such as service size, number of operations, number of port types, and cohesion. Although we can measure the number of operations of a service, an appropriate threshold value is not trivial to define. A service considered large in a given context could be considered as normal in another.

Furthermore, detecting Web service antipatterns is more complicated than OO ones. That is, Web service source code is located in the provider side, and clients could only access and invoke services through their interfaces described in WSDL documents. This makes the situation more difficult to assess, detect, and prevent badly designed Web service, i. e., antipatterns.

2.3 Motivating examples

The WSDL fragment below illustrates the salient aspects of a multiservice antipattern, in the form of a service interface. The core identifying aspect of a multiservice antipattern is that it implements multiple core business and/or technical abstractions with low operations cohesion. This is manifested at the service interface as different public methods that involve different entities or abstractions. In this example, it can be seen that there are methods that operate on different core functionalities. For instance, the bookFlight() method used to book a flight trip, while the reserveHotel() method attempts to reserve the specified hotel room. Overall, this multiservice supports the functionalities flight, car and hotel booking, payment, invoice services, and so on. Each of these is a significant core business abstraction, and typically will have many associated methods. Therefore, while this example is simplified and is merely illustrative, in reality, a typical multiservice will include many methods related to each abstraction, resulting in a service with huge number of methods.

On the other extreme, i. e., nanoservice, we consider the example of a Calculator service taken from real-world Web service provided by Apache Geronimo[1]. A basic calculator service would not be complicated; it supports several simple operations such as add, subtract, multiply, divide and other operations. The example of Apache Geronimo shows the WSDL file from the Apache Calculator service, which performs addition of two integers. This is a very fine-grained service as all it can do is accept two numbers and return the sum. However, there is a lot of code (and overhead) for this simple operation. As services are consumed over network (Internet, LAN), they might be bound by the limitations and costs incurred by communications over those networks (e.g., the time needed to send/receive messages) [3]. The problem becomes more disturbing when considering this level of granularity in other more complicated real-life services.

For non-expert clients the line between nanoservices, multiservices and appropriately sized services is not obvious. In addition, even for service providers, service logics may look

```
<wsdl:definitions>
    <wsdl:types>
     ...
    </wsdl:types>
     ...
    <wsdl:portType name="FlightPortType">
        <wsdl:operation name="bookFlight">
            ...
        </wsdl:operation>
        <wsdl:operation name="reserveFlight">
            ...
        </wsdl:operation>
        <wsdl:operation name="cancelFlight">
         ...
        </wsdl:operation>
   </wsdl:portType>
   <wsdl:portType name="OtherServicePortType">
        <wsdl:operation name="reserveCar">
            ...
        </wsdl:operation>
        <wsdl:operation name="cancelCar">
            ...
        </wsdl:operation>
        <wsdl:operation name="reserveHotel">
            ...
        </wsdl:operation>
        <wsdl:operation name="checkDates">
            ...
        </wsdl:operation>
        <wsdl:operation name="modifyBooking">
            ...
        </wsdl:operation>
        <wsdl:operation name="acceptPayment">
            ...
        </wsdl:operation>
        <wsdl:operation name="validateCredit">
            ...
        </wsdl:operation>
        <wsdl:operation name="generateInvoice">
            ...
        </wsdl:operation>
              ...
    </wsdl:portType>
          ...

</wsdl:definitions>
```

[1]https://cwiki.apache.org/confluence/display/GMOxDOC21/jaxws-calculator+-+Simple+Web+Service+with+JAX-WS

Table 2. List of used metrics.

Metric	Description
NPT	Number of port-types
NOD	Number of operations declared
NOPT	Average number of operations in port-types
NPO	Average number of parameters in operations
NCT	Number of complex types
NAOD	Number of accessor operations declared
NCTP	Number of complex type parameters
COUP	Coupling
COH	Cohesion
NOM	Number of messages
NST	Number of primitive types
ALOS	Average length of operations signature
ALPS	Average length of port-types signature
ALMS	Average length of message signature
RPT	Ratio of primitive types over all defined types
RAOD	Ratio of accessor operations declared
ANIPO	Average number of input parameters in operations
ANOPO	Average number of output parameters in operations
NPM	Average number of parts per message
AMTO	Average number of meaningful terms in operation names
AMTM	Average number of meaningful terms in message names
AMTP	Average number of meaningful terms in port-type names

promising at design level, but can prove to be antipatterns when they are implemented. To make the situation worst, a comprehensive service contract does not guarantee that a service is not an antipattern. Thus, it is very important to provide efficient techniques to support both Web service clients and providers.

To address or circumvent the above mentioned issues and challenges, we introduce a search-based approach to automatically derive Web service antipattern detection rules.

3. APPROACH

In this section, we describe our approach for Web service antipatt-erns detection. The key idea is to see the detection problem as a search based combinatorial optimization problem to find the sought detection rules from a large list of possible metrics and thresholds.

3.1 Approach overview

Figure 1 provides a high-level overview of the approach proposed in this paper. Our approach uses knowledge from a base of examples that contains real instances of Web service antipatterns. These examples will serve to generate new Web service antipattern detection rules based on combinations of Web service metrics and threshold values. The detection rules are automatically derived by an optimization process that learns from the available examples.

As shows in Figure 1, our approach takes as inputs a base (*i.e.*, a set) of Web service antipattern examples and a set of Web service metrics. As output, our approach derives a set of detection rules. Using GP [20], our rules' derivation process generates randomly, from a given list of metrics, a combination of metric/threshold for each antipattern type. Thus, the generation process can be viewed as a search-based combinatorial optimization to find the suitable combination of metrics/thresholds that best detect the antipattern instances in the base of examples. In other words, the best set of rules is the one that detects the maximum number of antipatterns in terms of precision and recall.

The base of examples contains different Web service antipatterns from different application domains (e.g., weather, finance, shipping, etc.) that can be collected from different Web service search engines, such as eil.cs.txstate.edu/ServiceXplorer, and programmableweb.com, etc. These antipatterns were manually

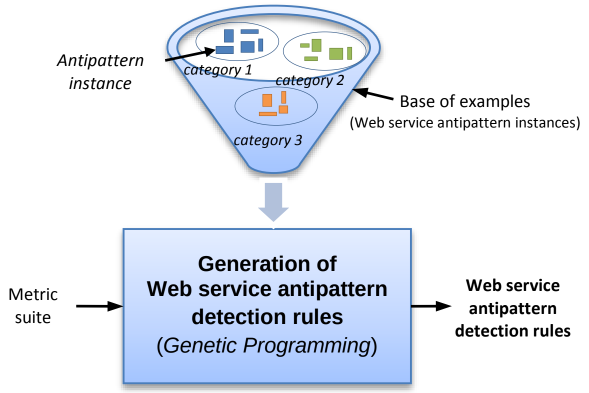
Antipattern instance
category 1
category 2
category 3
Base of examples
(Web service antipattern instances)
Metric suite
Generation of Web service antipattern detection rules
(Genetic Programming)
Web service antipattern detection rules

Figure 1. Approach overview

inspected and validated based on existing guidelines from the literature [3] [16]. During a training stage, these antipatterns are iteratively evaluated using rules generated by GP [20]. The process is driven by a fitness function that calculates the quality of each candidate solution (detection rule) by comparing the list of detected antipatterns with the expected ones from the base of examples.

Our metric suite is based on a set of Web service metrics. Table 1 summarizes the used metrics. The first fourteen metrics (NPT-ALMS) are defined in the literature [5] [10] [21] [19]. We also adapted and defined eight other metrics (RPT-AMTP). The last three metrics, AMTO, AMTM, and AMTP, are implemented based on WordNet[2], a widely used lexical database. Each operation, port-type and message identifier is tokenized based on camel case splitter. Then, we assume that the more the extracted tokens exist in WordNet database, the more the identifier is meaningful, i.e., semantically and syntactically sound.

As many metrics combinations are possible, the detection rules generation process is, by nature, a combinatorial optimization problem. The number of possible solutions quickly becomes huge as the number of metrics and possible threshold values increases. A deterministic search is not practical in such cases, and the use of heuristic search is warranted. The dimensions of the solution space are set by the metrics, their threshold values, and logical operations between them: union (metric1 OR metric2) and intersection (metric1 AND metric2). A solution is determined by assigning a threshold value to each metric. The search is guided by the quality of the solution according to the number of detected antipatterns in comparison to the expected ones form the base of examples.

3.2 SBSE formulation

Complex decision problems with multiple variables and large search spaces such as this are well-matched to search based software engineering (SBSE), which has proven good performance in providing decision support in several software engineering problems [22]. Our approach uses SBSE [22] [23], as it provides best practice to define a heuristic search algorithm, solution representation, fitness function, change operators, and so on [23]. In this section we describe our SBSE approach.

3.2.1 Search algorithm

As a search method, we employed a widely used computational search technique, GP [20], which have shown good performance in solving many software engineering problems [22]. GP takes as input a set of SOAP metrics and a set of Web service antipattern examples, and finds as output the optimal solution that corresponds to a set of rules that should detect the antipattern instances in the base of examples. For more details about GP, interested readers can refer to [20]. In the following, we need to define problem-specific solution encoding, genetic operators and fitness function to ensure best performance.

3.2.2 Solution representation

Candidate solutions to the problem are antipattern detection rules. A solution is represented as a set of IF – THEN rules. A detection rule has the following structure:

IF "Combination of metrics with their threshold values" **THEN** "antipattern type"

The IF clause describes the conditions or situations under which an antipattern type is detected. These conditions correspond to logical expressions that combine some metrics and their threshold values using logic operators (AND, OR). If some of these conditions are satisfied by a Web service, then it is detected as the antipattern type figuring in the THEN clause of the rule. We will have as many rules as types of antipatterns to be detected. In our case, mainly for illustrative reasons, and without loss of generality, we focus on the detection of five common types, namely multiservice, nanoservice, dataservice, chatty service, and ambiguous service (cf. Table 1). For instance, let us consider the following detection rules, in the iteration i, and its interpretation:

R1: **IF** (NOD(s)≥17 AND COH(s)≤0.43 AND NOPT(s)≥7.8) OR (NOD(s)≥24 AND COH(s)≤0.39 AND NPT(s)≥2 AND NST(s)≥41 OR NCT(s)≥32) **THEN** MultiService(s)

R2: **IF** (NCT(s)≤5 OR NST(s)≤8 AND NPT(s)≤2 AND NOD(s)≤5 AND COH(s)≥0.42) OR (NOPT(s)≤4.2 AND COUP(s)≥0.36 AND COH(s)≥0.39 AND NOD(s)≤6 OR NPT(s)≤2) **THEN** NanoService(s)

R3: **IF** ((ANIPO(s)≥4 OR ANOPO(s)≥4) AND (NCT(s)≥31 OR NOM(s)>=79) AND COH(s)≥0.31 AND NAOD(s)>=13) **THEN** DataService(s)

R4: **IF** (NPT(s)≤3 AND NOD(s)≥10 AND RAOD(s)≥0.38 AND (NCT(s)≥15 OR ANOPO(s)≥8.1) AND (NOM(s)>=38 OR NPM(s)>=2.2) AND COH(s)≤0.42) **THEN** ChattyService(s)

R5: **IF** (ALOS(s)≤1.6 OR ALOS(s)≥4.9 AND AMTO(s)≤0.6 AND NIOP(s)≥4 OR AMTM(s)≤0.52) **THEN** AmbiguousService(s)

We encoded a solution as tree where each subtree represents a detection rules for a particular antipattern type. Figure 2 represents the correspondent tree for the multiservice antipattern, i.e., R1.

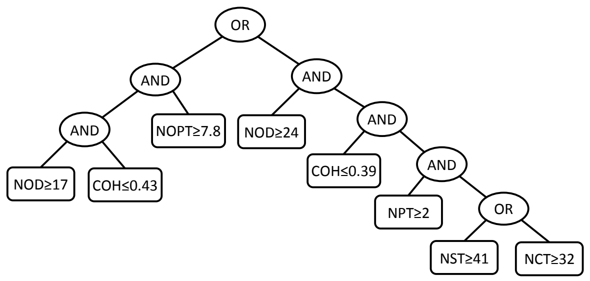

Figure 2. Solution encoding for the multiservice antipattern.

The initial population, composed by n solutions, was randomly obtained by assigning to each subtree m metrics ranging from 1 to nbMetrics (the number of considered metrics). For each metric we randomly assign a threshold value as defined in Section 3.1.

3.2.3 Fitness function

To evaluate the fitness of each solution we employed a fitness function that maximizes the number of detected antipatterns in comparison to the expected ones in the base of examples. In this

[2] wordnet.princeton.edu

Table 3. Web services used in the empirical study.

Category	# services	# antipatterns	Average # of operations	Average # of messages	Average # of complex types
Financial	94	67	29.52	57.31	19.01
Science	34	3	8.47	17.14	96.73
Search	37	13	8.35	18.94	26.13
Shipping	38	10	13.36	27.76	20.21
Travel	65	28	16.09	33.13	121.13
Weather	42	15	8.54	17.16	9.14
All	310	136	17.08	34.2	48.6

Table 4. Algorithms parameters.

Algorithm	Parameters	Values
GP	Population size	100
	Crossover probability	0.9
	Mutation probability	0.1
	Number of crossing points	1
	Selection	Roulette-wheel selection
SA	Initial temperature	100
	Final temperature	0.0232
	Cooling coefficient	0.99
	Number of iterations	30
PSO	Number of particles in a swarm	200
	Acceleration coefficient c_1	2
	Acceleration coefficient c_2	2

context, we define the fitness function of a candidate solution, as the average of both precision and recall as follows:

$$fitness = \frac{\frac{\sum_{i=1}^{p} a_i}{t} + \frac{\sum_{i=1}^{p} a_i}{p}}{2} \in [0,1]$$

where t is the number of antipatterns in the base of examples, p is the number of detected antipatterns, and a_i has value 1 if the ith detected service exists in the base of examples with the same antipattern type, i.e., true positive, and value 0 otherwise.

3.2.4 Genetic operators

Crossover: We use a random, single point crossover operator. Two parent solutions 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 only on parents having the same type of antipattern. Each child thus combines information from both parents.

Mutation: The mutation operator can be applied either to a function node or a terminal node. This operator can modify one or many nodes. For a selected individual, the mutation operator first randomly selects a node in the tree. Then, if the selected node is a terminal (quality metric), it is replaced by another terminal (metric or another threshold value); if the selected node is a function (AND-OR operators), it is replaced by a new function (e.g., AND becomes OR). If a tree mutation is to be carried out, the node and its subtree are replaced by a new randomly generated subtree.

4. VALIDATION

This section explains the design of our empirical study; the research questions we set out to answer, the methods and statistical tests we used to answer these questions. The experimental material is available for replication purposes[3].

4.1 Research questions

We designed our experiments to answer the following research questions:

RQ1 (SBSE Validation): How does the proposed GP-based approach performs compared to random search and other existing search-based algorithms?

RQ2 (Efficiency): To which extent can the proposed approach detect Web service antipatterns?

RQ3 (Sensitivity): What types of Web service antipatterns does it detect correctly?

4.2 Experimental setting

4.2.1 Analysis method

To answer RQ1, we compared our GP formulation with random search (RS) [24] to make sure that there is a need for an intelligent method to explore the search space. In addition, to justify the adoption of GP, we compared our approach to two other popular search algorithms namely particle swarm optimization (PSO) [25] and simulated annealing (SA) [26]. RQ1 is a standard 'baseline' question asked in any attempt at an SBSE formulation [23]. To evaluate the efficiency of each algorithm in detecting Web service antipatterns in comparison to RS, PSO and SA, we use precision and recall metrics, which are defined as follows:

$$Recal = \frac{true\ positives}{total\ number\ of\ antipatterns\ in\ the\ base\ of\ examples}$$

$$Precision = \frac{true\ positives}{number\ of\ detected\ antipatterns}$$

To answer RQ2, we also use both recall and precision criteria to evaluate the efficiency of our approach in identifying antipatterns. We considered five common Web service antipattern types, namely multiservice, nanoservice, chatty service, data service, and ambiguous service (see Section 2.1).

To answer RQ3, we investigated the antipattern types that were detected to find out whether there is a bias towards the detection of specific antipattern types.

4.2.2 Web services used in the empirical study

Unlike OO open-source systems, Web service providers do not make their source code publicly available; instead, they only provide Web service interface described as a WSDL document. We collected different Web services using different search engines including eil.cs.txstate.edu/ServiceXplorer, biocatalogue.org, webservices.seekda.com, taverna.org.uk, programmableweb.com, and myexperiment.org. Furthermore, to not bias our empirical study, we used different Web services from different application domains. Table 3 and Figure 4 summarize the collected services ranging from a variety of six categories, i.e., application domains, including financial, science, search, shipping, travel and weather. All services were manually inspected and validated to identify antipatterns based on guidelines from the literature [3] [16].

In our study, we used a 6-fold cross validation procedure. We split our data into training data and evaluation data. For each fold, one category of services is evaluated by using the remaining five categories as a base of examples. For instance, weather services are analyzed using antipattern instances from travel, shipping, search, science and financial services. Hence, precision and recall scores are calculated automatically by comparing the detected antipatterns with the expected ones.

4.2.3 Inferential statistical test methods used

Due to the stochastic nature of the used algorithms, they may produce slightly different results when applied to the same problem instance over different runs. To cope with this stochastic nature, the use of a rigorous statistical testing is essential to provide support to the conclusions derived from analyzing such data. Thus, we used

[3]http://www-etud.iro.umontreal.ca/~ouniali/WebServiceAntipatterns/

the Wilcoxon rank sum test in a pairwise fashion [27] in order to detect significant performance differences between the algorithms under comparison. We set the confidence limit, α, at 0.05. In these settings, each experiment is repeated 31 times, for each algorithm and for each category. The obtained results are subsequently statistically analyzed with the aim to compare our GP approach to PSO, SA, as well as RS. The results reported in this paper are the median values of the 31 runs.

The Wilcoxon rank sum test allows verifying whether the results are statistically different or not. However, it does not give any idea about the difference magnitude. To assess the effect size, we use the Cohen's *d* statistic [27]. The effect size is considered: (1) small if $0.2 \leq d < 0.5$, (2) medium if $0.5 \leq d < 0.8$, or (3) high if $d \geq 0.8$.

4.2.4 Parameter Tuning and Setting

An important aspect for metaheuristic search algorithms lies in the parameters tuning and selection, which is necessary to ensure not only fair comparison, but also for potential replication. To this end, we report in Table 4 our algorithmic parameter tuning and selection used to facilitate the replication of our findings. The initial population/solution of GP, PSO, SA, and RS are completely random. The stopping criterion is when the maximum number of fitness evaluations, set to 25000, is reached. The max depth of the tree is fixed to 10. After several trial runs of the simulation, the parameter values of the four algorithms are fixed. There are no general rules to determine these parameters, and thus, we set the combination of parameter values by trial-and-error method, which is commonly used in the SBSE community [28].

4.3 Results and discussions

This section presents the experimental results obtained for our three research questions.

Results for RQ1. Table 5 and Figure 3 report the statistical results for RQ1. As presented in Table 5 and Figure 3, over 31 runs, the RS did not perform well in terms of precision and recall (only 30% and 42% respectively) due to the huge search-space of possible combinations of metrics and threshold values to explore. Indeed, in any attempt at an SBSE formulation of a problem, if the proposed formulation does not allow an intelligent computational search technique to outperform random search convincingly, then there is clearly something wrong with the formulation [23].

On the other hand, for the different categories, the statistical analysis provide evidence that our GP-based approach performs better (with a 95% confidence level) than two other metaheuristic search algorithms (PSO and SA). GP provides better performance than SA in all the six categories with high Cohen effect size. The median recall and precision scores of GP for all studied services (union of the six categories) are 87% and 85% respectively, whereas SA provide only 70% of both recall and precision. Similarly, according to Figure 3 and Table 5, GP provides better performance than PSO in four out of six cases with high effect size. Only in science and travel Web services, GP and PSO provide similar results with small effect size in terms of recall, but still with better performance for GP in terms of precision manifested by high Cohen effect size. Overall, the median recall and precision scores of GP for all studied services were better than PSO that provides only 82% of recall and 76% of precision.

Based on these results, we can conjecture that GP performs much better in comparison with PSO and SA. Moreover, we notice that SA turns out to be the worst algorithm in comparison with GP and PSO. Thus, it seems that population-based metaheuristic algorithms tend to be more efficient than local search

Table 5. Precision and recall median values of GP, PSO, SA, and RS over 31 independent simulation runs.

Category	GP		PSO		SA		RS	
	Precision (%)	*Recall* (%)	*Precision* (%)	*Recall* (%)	*Precision* (%)	*Recall* (%)	*Precision* (%)	*Recall* (%)
Financial	88 (o-++)	85 (o+++)	79 (-o++)	78 (+o++)	75 (++o+)	75 (++o+)	42 (+++o)	45 (+++o)
Science	75 (o+++)	100 (o+++)	50 (+o-+)	100 (+o-+)	43 (+-o+)	100 (-+o+)	17 (+++o)	67 (+++o)
Search	85 (o+++)	85 (o+++)	79 (+o++)	85 (+o++)	63 (++o+)	77 (++o+)	26 (+++o)	46 (+++o)
Shipping	90 (o+++)	90 (o+++)	53 (+o-+)	80 (+o-+)	57 (+-o+)	80 (+-o+)	19 (+++o)	40 (+++o)
Travel	80 (o+++)	86 (o-++)	89 (+o++)	86 (-o++)	81 (++o+)	79 (++o+)	27 (+++o)	36 (+++o)
Weather	76 (o+++)	87 (o-++)	72 (+o++)	87 (-o++)	61 (++o+)	73 (++o+)	20 (+++o)	33 (+++o)
All	85 (o+++)	87 (o+++)	76 (+o++)	82 (+o++)	70 (++o+)	76 (++o+)	30 (+++o)	42 (+++o)

A "+" symbol at the ith position means that the algorithm precision median value is statistically different from the ith algorithm one; while a "-" symbol at the ith position means the opposite. A "o" symbol refer to the current position of the algorithm. For instance, for financial services, GP precision is not statistically different from PSO one, however, it is statistically different from SA and RS ones).

metaheuristics for this problem especially that we use tree representation.

Results for RQ2. To answer RQ2, we focus only on the results of our GP presented in Table 5 and Figure 3. Overall, as shown in Table 5, we were able to detect antipatterns on the different service categories with a precision score of 85 percent. For science and weather services, the precision is lower than the other categories with respectively 75 and 76 percent. This can be explained by the fact that these services are medium-sized and often contain too much data and accessor operations to these data which might be relatively confusing between multiservice, data service and chatty service. For financial and shipping services, the precision score is higher (88 and 100 percent), i.e., most of the detected antipatterns are true positives. In terms of recall, the obtained results are higher than precision. According to Table 5, the median GP recall score on all services is 87 percent. We found that, science and shipping services have the highest recall scores with 100% and 90% respectively. We also had a good trade-off between both recall and precision. Therefore, we can conclude that our approach provides good performance to detect most of the existing antipatterns, which could be very helpful to provide advice to both service clients and providers on the quality of their Web services.

Results for RQ3. Based on the results of Figure 5, we noticed that our technique does not have a bias towards the detection of specific antipattern types. Figure 5 shows that we had, in all categories, a relatively equitable detection results in terms of both precision and recall for each antipattern type. For some categories such as weather and search, the distribution of antipatterns detection is not as balanced (cf. Figure 4). This is principally due to the number of actual antipattern types in these categories (none dataservice instance exists in weather and search category). Consequently, any single false positive will lead to 0% of precision score.

Overall, all the five antipattern types are detected with good precision and recall scores (85% and 87% respectively). Most of detected antipatterns are true positives and we do not miss any existing antipattern type. This ability to identify different types of antipatterns underlines a key strength to our approach. Most other existing approaches [10] rely heavily on the notion of size to specify antipatterns. This is reasonable considering that some antipatterns like the multiservice are associated with a notion of size. For antipatterns like data service and ambiguous service, however, the notion of size is less important and this makes this type of antipatterns hard to detect using structural information. The

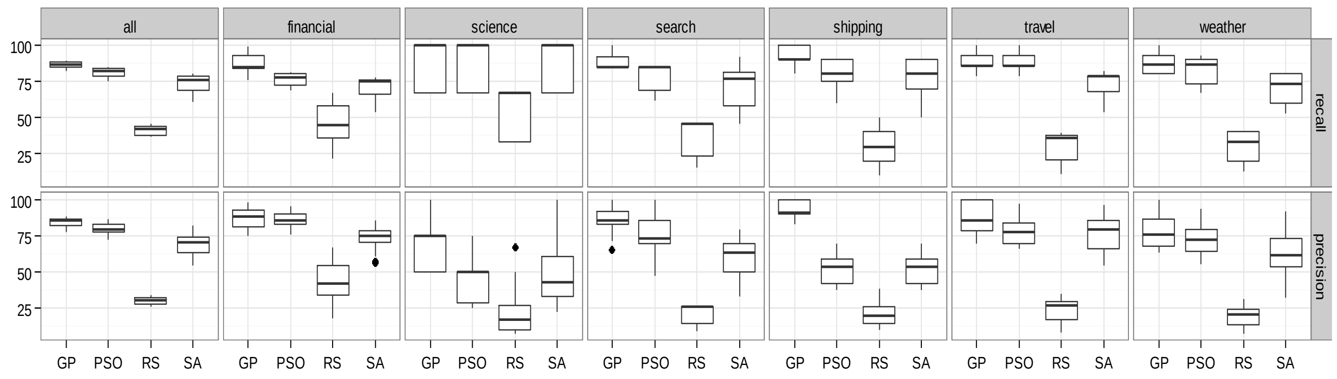

Figure 3. Boxplots for the obtained detection results in terms of recall and precision, for each Web service category, and for each search algorithm GP, PSO, SA and RA.

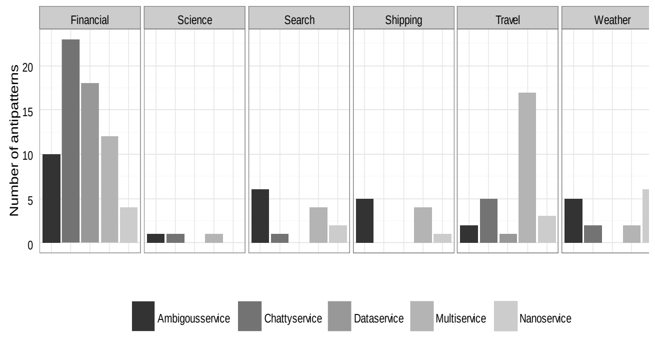

Figure 4. Antipatterns distribution for each category.

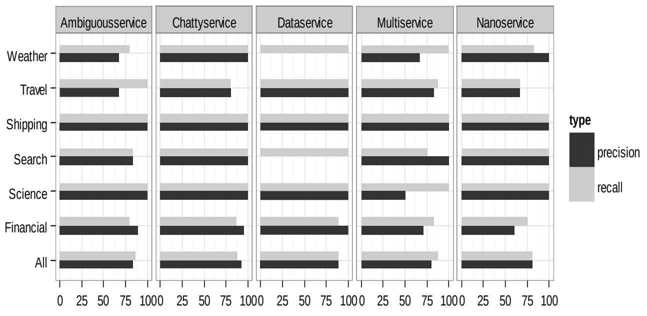

Figure 5. Detection results for each antipattern type.

obtained results provides evidence that such difficulty does not limits the performance of our approach in well detecting these types of antipatterns. Thus, we can conclude that our GP-based approach detects well all types of the considered antipatterns (RQ3).

Furthermore, it is important to evaluate the scalability of the performance of our approach, as scalability is widely considered as one of the key issues for software engineering research and development. To evaluate scalability of our approach for services of increasing size, we executed our approach on the six categories of services. Figure 6 illustrates the evolution of precision, recall and CPU time with respect to the increase of service size (in terms of number of operations). We see from this figure that the precision and recall values are relatively stable (between [75, 90]) even if the services size increases. The same observation could be seen for CPU time which is between [224, 241] seconds. We can say that our approach is scalable with respect to service size since it gives high precision and recall values, and acceptable execution time.

5. THREATS TO VALIDITY

In our study, external threat to validity may arise because, although we considered five types of Web service antipatterns, we did not evaluate the detection of other antipattern types. In addition, we validated our approach on SOAP Web services; and we cannot generalize our results to other technologies such as REST. In future work, we plan to evaluate the performance of our approach to detect other types of antipatterns, and other SOA technologies.

Construct threats to validity can be related to the set of used metrics, and the corpus of antipattern examples as developers may not all agree if a candidate Web services is an antipattern or not according to their level of expertise on antipatterns. Since we are the first to address this problem for automating the detection of web service antipatterns, there is no currently established state of the art in terms of automated detection. We also found few literature to guide us on what we should consider to inspect Web service antipatterns [9] [16] [10]. In future work, we will consider more static and dynamic Web service metrics, and ask some new experts to extend the existing corpus and provide additional feedback.

6. RELATED WORK

Detecting and specifying antipatterns in SOA and Web service is relatively a new field. Only few works have addressed the problem of SOA antipatterns. The first book in the literature was written by Dudney et al. [16] where a set of Web service antipatterns have been informally defined. Recently, a new book [3] have been written to describe the symptoms of some other SOA antipatterns. Furthermore, Král et al. [14] listed seven “popular” SOA antipatterns that violate SOA principles. In addition, some few research works have addressed the detection of such antipatterns. Recently, Palma et al. [10] have proposed an approach to detect Web service antipatterns. The proposed approach relies on declarative rule specification using domain-specific language (DSL) to specify/identify the key symptoms that characterize an antipattern. Similarly, Moha et al. [19] have proposed a rule-based approach called SODA for SCA systems (Service Component Architecture). However, unlike our approach, in an exhaustive scenario, the number of possible antipatterns to manually characterize with rules can be very large; and rules that are expressed in terms of metric combinations need substantial calibration efforts to find the right threshold value for each metric. In another study, Rodriguez et al. [2] [9] [17] 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.

Unlike service-oriented systems, there is an extensive research effort on detecting object oriented antipatterns and code-smells [6] [29] [7] [8]. For instance, Marinescu el al. [29] have proposed a mechanism called "detection strategy" OO code-smells by

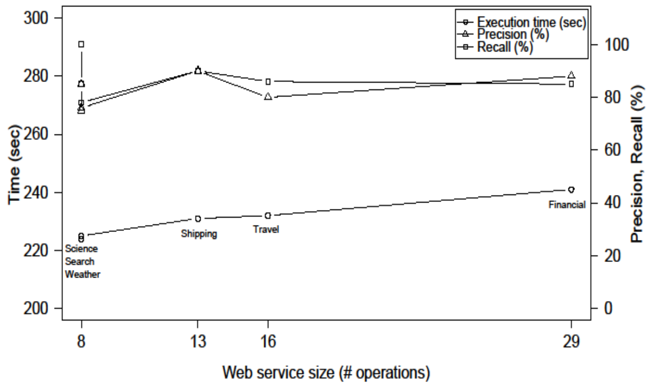

Figure 6. Scalability with respect to Web service size.

formulating metric-based rules that capture deviations from good design principles and heuristics. Ouni et al. [7] proposed a search-based approach to detect code-smells in OO software systems. However, OO antipatterns detection techniques are not applicable in the context of Web services as we deal with different level of granularity (service vs class levels), and different metrics. Furthermore, unlike OO systems, Web service source code is not publicly available; that is only WSDL interfaces are available for clients. This makes the detection of such antipatterns harder.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced a new search-based approach for Web service antipatterns detection. In our GP adaptation, detection rules are represented as a combination of metrics and threshold values that should detect as much as possible the number of antipatterns from a base of examples. The statistical analysis of the obtained results provides compelling evidence that GP outperforms particle swarm optimization, simulated annealing as well as random search based on a benchmark of 310 Web services including 136 real-world antipattern instances. As future work, we plan to validate our approach with additional antipattern types, and SOAP static and dynamic metrics in order to conclude about the general applicability of our methodology. Another research direction worth to explore is to consider both bad and good Web service instances to deduce antipattern detection rules, i.e., good detection rules should maximize the distance with well-designed Web services while minimizing the distance with badly-designed ones. Furthermore, in this paper, as we mainly focus on the detection of Web service antipatterns, we are planning to extend the approach by automating their correction using SBSE.

Acknowledgments

This work is supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (S) (No.25220003), and also by Osaka University Program for Promoting International Joint Research.

8. REFERENCES

[1] M. P. Singh and M. N. Huhns, *Service-oriented computing - semantics, processes, agents*: Wiley, 2005.

[2] J. Rodriguez, M. Crasso, C. Mateos, A. Zunino, "Best practices for describing, consuming, and discovering web services: a comprehensive toolset," *Software: Practice and Experience,* vol. 43, pp. 613-639, 2013.

[3] A. Rotem-Gal-Oz, *SOA Patterns*: Manning Publications, 2012.

[4] J. Král M. Žemlička, "Crucial Service-Oriented Antipatterns," *Int. Journal On Advances in Software,* vol. 2, pp. 160-171, 2009.

[5] J. L. O. Coscia, M. Crasso, C. Mateos, and A. Zunino, "Estimating Web Service interface quality through conventional object-oriented metrics," *CLEI Electron. J.,* vol. 16, 2013.

[6] N. Moha, G. Yann-Gaël, L. Duchien, A. Le Meur, "DECOR: A Method for the Specification and Detection of Code and Design Smells," *Software Engineering, IEEE Transactions on,* vol. 36, pp. 20-36, 2010.

[7] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering,* vol. 20, pp. 47-79, 2013.

[8] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection," *Software Engineering, IEEE Transactions on,* vol. 40, pp. 841-861, 2014.

[9] C. Mateos, M. Crasso, A. Zunino, J. L. O. Coscia, "Avoiding WSDL Bad Practices in Code-First Web Services," *SADIO Electronic Journal of Informatics and Operational Research,* vol. 11, pp. 31-48, 2012.

[10] F. Palma, N. Moha, G. Tremblay, and Y.-G. Guéhéneuc, "Specification and Detection of SOA Antipatterns in Web Services," in *Software Architecture*. vol. 8627, P. Avgeriou and U. Zdun, Eds., ed: Springer International Publishing, pp. 58-73, 2014.

[11] M. zur Muehlen, J. V. Nickerson, and K. D. Swenson, "Developing web services choreography standards - the case of REST vs. SOAP," *Decision Support Systems,* vol. 40, pp. 9-29, 2005.

[12] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana, "Web services description language (wsdl) version 2.0 part 1: Core language," *W3C recommendation,* vol. 26, p. 19, 2007.

[13] M. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Software Engineering,* vol. 11, pp. 395-431, 2006.

[14] J. Král M. Zemlicka, "Popular SOA Antipatterns," in *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009. COMPUTATIONWORLD'09. Computation World:*, 2009, pp. 271-276.

[15] J. Krai and M. Zemlicka, "The Most Important Service-Oriented Antipatterns," in *Software Engineering Advances, 2007. ICSEA 2007. International Conference on*, 2007.

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

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

[18] M. Nayrolles, F. Palma, N. Moha, and Y.-G. Guéhéneuc, "Soda: A Tool Support for the Detection of SOA Antipatterns," in *Service-Oriented Computing - ICSOC 2012 Workshops*. vol. 7759, A. Ghose, H. Zhu, Q. Yu, A. Delis, Q. Sheng, O. Perrin, *et al.*, Eds., ed: Springer Berlin Heidelberg, pp. 451-455, 2013.

[19] N. Moha, F. Palma, M. Nayrolles, B. Conseil, Y.-G. Guéhéneuc, B. Baudry, "Specification and Detection of SOA Antipatterns," in *Service-Oriented Computing*. vol. 7636, C. Liu, H. Ludwig, F. Toumani, and Q. Yu, Eds., ed: Springer Berlin Heidelberg, pp. 1-16, 2012.

[20] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*: MIT Press, 1992.

[21] P. Mikhail, R. Caspar, and T. Zahir, "The Impact of Service Cohesion on the Analyzability of Service-Oriented Software," *IEEE Transactions on Services Computing,* vol. 3, pp. 89-103, 2010.

[22] M. Harman, P. McMinn, J. de Souza, and S. Yoo, "Search Based Software Engineering: Techniques, Taxonomy, Tutorial," in *Empirical Software Engineering and Verification*. vol. 7007, B. Meyer and M. Nordio, Eds., ed: Springer Berlin Heidelberg, , pp. 1-59, 2012.

[23] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology,* vol. 43, pp. 833-839, 2001.

[24] D. C. Karnopp, "Random search techniques for optimization problems," *Automatica,* vol. 1, pp. 111-121, 1963.

[25] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Neural Networks, 1995. Proceedings., IEEE International Conference on*, , pp. 1942-1948, vol.4, 1995.

[26] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science,* vol. 220, pp. 671-680, 1983.

[27] J. Cohen, *Statistical power analysis for the behavioral sciences*, 2nd ed.: Lawrence Erlbaum Associates, Inc, 1988.

[28] A. E. Eiben and S. K. Smit, "Parameter tuning for configuring and analyzing evolutionary algorithms," *Swarm and Evolutionary Computation,* vol. 1, pp. 19-31, 2011.

[29] R. Marinescu, "Detection strategies: metrics-based rules for detecting design flaws," in *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, , pp. 350-359, 2004.