This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2502595, IEEE Transactions on Services Computing

Search-based Web Service Antipatterns Detection

Ali Ouni, Marouane Kessentini, Katsuro Inoue and Mel Ó Cinnéide

Abstract—Service Oriented Architecture (SOA) is widely used in industry and is regarded as one of the preferred architectural design technologies. As with any other software system, service-based systems (SBSs) may suffer from poor design, i.e., antipatterns, for many reasons such as poorly planned changes, time pressure or bad design choices. Consequently, this may lead to an SBS product that is difficult to evolve and that exhibits poor quality of service (QoS). Detecting Web service antipatterns is a manual, time-consuming and error-prone process for software developers. In this paper, we propose an automated approach for detection of Web service antipatterns using a cooperative parallel evolutionary algorithm (P-EA). The idea is that several detection methods are combined and executed in parallel during an optimization process to find a consensus regarding the identification of Web service antipatterns. We report the results of an empirical study using eight types of common Web service antipatterns. We compare the implementation of our cooperative P-EA approach with random search, two single population-based approaches and one state-of-the-art detection technique not based on heuristic search. Statistical analysis of the obtained results demonstrates that our approach is efficient in antipattern detection, with a precision score of 89% and a recall score of 93%.

Index Terms—Web Services, Web Service Design, Antipattern, Service-oriented Computing, Search-based Software Engineering.

1 INTRODUCTION

SERVICE-Oriented Architecture (SOA) has emerged as a logical way to design complex distributed software systems using functionality implemented by third-party providers. In an SOA, the service requester satisfies their specific needs by using services offered by service providers, through published and discoverable interfaces. Services can be implemented using a variety of technologies such as Web Services, OSGi, and SCA.

Web services is the common technological choice for materializing the Service-oriented Computing (SOC) paradigm [1] [2]. Examples include Google Maps[1], PayPal[2], FedEx[3], Dropbox[4], and many others. Web services must be carefully designed and implemented to adequately fit in the required system's design whilst achieving QoS [3] [4]. Indeed, there is no exact recipe to follow for proper service design. A set of guiding quality principles for service-oriented design exists, including such principles as service flexibility, operability, composability, and loose coupling [5]. However, the design of services is strongly influenced by the context, environment and other decisions the service designers take [4], and such factors may lead to violations of quality principles. The presence of programming patterns associated with bad design and programming practices, known as *antipatterns*, are an indication of such violations [6] [7] [8]. Furthermore, it is widely believed that such antipatterns lead to various maintenance and evolution problems including an increased bug rate, fragile design and inflexible code.

Despite the extensive adoption of Web service technologies, no automated approach has been proposed for the detection of such antipatterns. Unlike antipatterns and code smells in object-oriented programs [9] [10] [11] [12], antipattern detection in SBSs is still in its infancy. Indeed, the vast majority of existing work in Web service antipattern detection merely attempts to provide definitions and/or the key symptoms that characterize common antipatterns. Recent works [13] rely on a declarative rule-based language to specify antipattern symptoms at a higher-level of abstraction using combinations of quantitative (metrics), structural, and/or lexical information. However, in an exhaustive scenario, the number of possible antipatterns to be characterized manually and formulated with rules can be very large. To make the situation worse, it is difficult to find a consensus to characterize and formulate such symptoms. For these reasons, the detection task is still mainly a manual, time-consuming and subjective process.

To address these issues, we proposed a search-based approach [14] that was one of the first attempts to automate Web service antipattern detection. The proposed approach is based on the use of genetic programming (GP) to generate detection rules from examples of Web service antipatterns using static service interface metrics [14]. However, the quality of the generated rules depends on the coverage of the different suspicious behaviors of antipatterns, and it is difficult to ensure such coverage. Thus, there are still some uncertainties regarding the detected antipatterns due

- *A. Ouni and K. Inoue are with the Department of Computer Science, Osaka University, Japan. E-mail: {ali, inoue}@ist.osaka-u.ac.jp.*
- *M. Kessentni is with the University of Michigan, USA. E-mail: marouane@umich.edu.*
- *M. Ó Cinnéide is with the University College Dublin, Ireland. E-mail: mel.ocinneide@ucd.ie.*

Manuscript received August 17, 2015; revised , 2015. This work is supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (S) (No.25220003), by Osaka University Program for Promoting International Joint Research, by the Ford-University of Michigan alliance Program and by Science Foundation Ireland grant 10/CE/I1855 to Lero - the Irish Software Research Centre (www.lero.ie).

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

1939-1374 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2502595, IEEE Transactions on Services Computing

to the difficulty in evaluating the coverage of the base of antipattern examples. Furthermore, using only static service interface metrics might not be enough to cover such antipattern symptoms. On the other hand, in some contexts, certain Web service antipattern instances may in fact be consensually accepted as normal practice [5].

This paper extends our previous work [14]. We propose to treat Web service antipattern detection as a cooperative parallel optimization problem. The idea is that different methods are combined in parallel during an optimization process where the goal is to find a *consensus* regarding the identification of Web service antipatterns. To this end, we used Parallel Evolutionary algorithms (P-EAs) [15] where different evolutionary algorithms (EAs) [16] with different adaptations (fitness functions, solution representations, and change operators) are executed, in a parallel cooperative manner, to solve a common goal, namely the detection of Web service antipatterns.

We believe that a P-EA approach is suitable to our problem because it combines different perspectives and levels of expertise to detect potential antipatterns, as was demonstrated in its adaptation to a similar problem, namely the detection of code smells in Java programs [12]. In our P-EA, many populations' individuals evolve simultaneously. A first population generates a set of detection rules using genetic programming [17] while simultaneously a second population tries to detect antipatterns using a genetic algorithm [18]. Both populations are evolved on the Web service that is being evaluated, and the quality of the employed detection rules is updated based on the consensus found, i.e., the intersection between the detection results of both populations. The best detection results will be the antipatterns detected by both algorithms. The P-EA approach does not involve merely executing the two EAs in parallel, but also in building a *consensus* between them to classify the detected candidates based on several interactions at the fitness function level. The main contributions of this paper can be summarized as follows:

1) We propose a novel automated approach for Web service antipattern detection as a cooperative parallel optimization problem. In our approach, different EAs, with different adaptation schemes, are executed in parallel to meet a consensus regarding the detection of antipatterns.
2) We extend our initial metric suite [14] that was limited to only Web service interface-level metrics (WSDL). We include i) Web service code-level metrics, and ii) Web service dynamic metrics to better uncover potential antipattern symptoms. Hence, static properties are recoverable from service interface and artefacts. In contrast, dynamic properties are obtained by concretely invoking the Web service.
3) We extend our base of Web service antipattern examples. Our dataset is available online in order to encourage future research in the area of automated Web service antipattern detection [19].
4) We extend the evaluation of the approach. We present an empirical evaluation of our approach on i) a benchmark of 415 Web services from ten different application domains and ii) three additional common Web service antipattern types (eight antipatterns in total). We compare our P-EA approach with i) random search, ii) single population-based approaches and iii) an existing rule-based approach [13] (not based on computational search techniques). Statistical analysis demonstrates the efficiency of our approach in detecting Web service antipatterns, with a precision score of 89% and a recall score of 93%.

The remainder of this paper is organized as follows. Section 2 is dedicated to background material related to this research. Section 3 presents the various issues related to the automation of Web service antipattern detection while Section 4 describes our P-EA based approach to this problem. Section 5 presents and discusses the obtained experimental results while Section 6 discusses threats to validity. Section 7 surveys related work and finally we conclude and outline our future research directions in Section 8.

2 Background

2.1 Web service and Web service antipatterns

A Web Service is defined according to the W3C[5] (World Wide Web Consortium), as "a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artefacts" [20]. Its interface is described as a WSDL (Web service Description Language) document that contains structured information about the Web service's location, its offered operations, the input/output parameters, etc. The aim of the Web service 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 bad solutions to recurring design problems. They often lead to software which is hard to maintain and evolve [21], and may be introduced unintentionally during initial design or during software development due to bad design choices, poorly planned changes or time pressure.

Different types of antipatterns presenting a variety of symptoms have been recently studied with the intent of improving their detection and suggesting improvements paths [22] [7] [13]. Common Web service antipatterns include:

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 [22].

Fine grained Web service (FGWS): 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 [22].

Chatty Web service (CWS): represents an antipattern where a high number of operations, typically attribute-level

[5]http://www.w3.org/TR/wsdl20/

1939-1374 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2502595, IEEE Transactions on Services Computing

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 [22].

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 data Web service usually deals with very small messages of primitive types and may have high data cohesion [13].

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 discoverability and reusability of Web services [23] [24].

Redundant PortTypes (RPT): is an antipattern where multiple portTypes are duplicated with the similar set of operations. Very often, such portTypes deal with the same messages. Redundant PortType antipattern may negatively impact the ranking of the Web Services [25].

CRUDy Interface (CI): is an antipattern where the design encourages services the RPC-like behavior by declaring create, read, update, and delete (CRUD) operations, e.g., `createX()`, `readY()`, etc. 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 [22].

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 [22].

In this paper, we focus mainly on these eight antipattern types as they are the antipatterns that occur most frequently in SBSs based on recent studies [13] [4] [6] [7] [26].

2.2 Illustrative Examples

Figure 1 illustrates the salient aspects of a GOWS antipattern. The core identifying aspect of a GOWS is that it implements multiple core business and/or technical abstractions with uncohesive operations. This is manifested at the service interface as 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 is used to book a flight trip, while the `reserveHotel()` method attempts to reserve the specified hotel room. Overall, this GOWS 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 GOWS will include many methods related to each abstraction, resulting in a service with huge number of methods.

On the other extreme, i.e., FGWS, we consider the example of a Calculator service taken from real-world Web

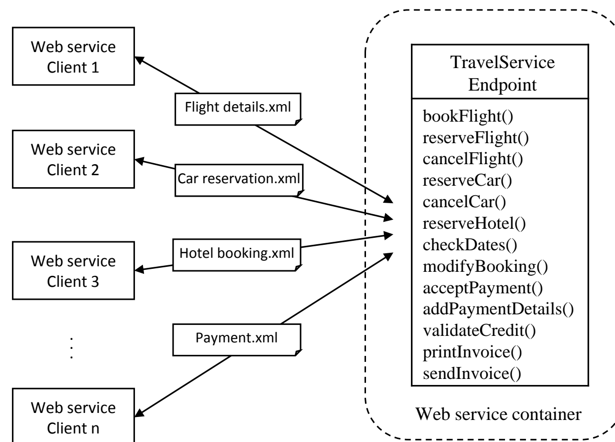


Fig. 1: An example of god object Web service.

service provided by Apache Geronimo[6]. A basic calculator service would not be complicated; it supports several simple operations such as add, subtract, multiply, divide. Figure 2 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 a network (Internet or LAN), they may be bound by the limitations and costs incurred by communications over those networks (e.g., the time needed to send/receive messages) [4]. 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 GOWS, FGWS and appropriately-sized services is not obvious. In addition, even for service providers, service logics may look promising at design level, but can prove to be antipatterns when they are implemented. To make the situation worse, a comprehensive service contract does not guarantee that a service is not an antipattern. Thus, it is very important to provide efficient detection techniques for both Web service clients and providers.

3 Problem Statement

In this section, we introduce the specific Web service antipattern detection issues and challenges that are addressed by our approach.

How to decide if a candidate antipattern is an actual antipattern? The main issue with Web service antipattern detection is that there is no general consensus on how to decide if a particular design violates a quality heuristic. Indeed, there is a difference between detecting symptoms and asserting that the detected situation is an actual antipattern. Deciding which Web services are antipattern candidates heavily depends on the interpretation of each analyst. In some contexts, an apparent violation of a design principle may be consensually accepted as normal practice. For example, a translation Web service[7] may have in its interface

[6]https://cwiki.apache.org/confluence/display/GMOxDOC21/jaxws-calculator+-+Simple+Web+Service+with+JAX-WS
[7]http://www.webservicex.net/TranslateService.asmx?WSDL

1939-1374 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2502595, IEEE Transactions on Services Computing


```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <wsdl:definitions name="Calculator"
3     xmlns="http://schemas.xmlsoap.org/wsdl/"
4     xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
5     xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
6     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
7     targetNamespace="http://jws.samples.geronimo.apache.org"
8     xmlns:tns="http://jws.samples.geronimo.apache.org">
9
10    <wsdl:types>
11        <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
12                    xmlns="http://jws.samples.geronimo.apache.org"
13                    targetNamespace="http://jws.samples.geronimo.apache.org"
14                    attributeFormDefault="unqualified" elementFormDefault="qualified">
15
16            <xsd:element name="add">
17                <xsd:complexType>
18                    <xsd:sequence>
19                        <xsd:element name="value1" type="xsd:int"/>
20                        <xsd:element name="value2" type="xsd:int"/>
21                    </xsd:sequence>
22                </xsd:complexType>
23            </xsd:element>
24
25            <xsd:element name="addResponse">
26                <xsd:complexType>
27                    <xsd:sequence>
28                        <xsd:element name="return" type="xsd:int"/>
29                    </xsd:sequence>
30                </xsd:complexType>
31            </xsd:element>
32        </xsd:schema>
33    </wsdl:types>
34
35    <wsdl:message name="add">
36        <wsdl:part name="add" element="tns:add"/>
37    </wsdl:message>
38
39    <wsdl:message name="addResponse">
40        <wsdl:part name="addResponse" element="tns:addResponse"/>
41    </wsdl:message>
42
43    <wsdl:portType name="CalculatorPortType">
44        <wsdl:operation name="add">
45            <wsdl:input name="add" message="tns:add"/>
46            <wsdl:output name="addResponse" message="tns:addResponse"/>
47        </wsdl:operation>
48    </wsdl:portType>
49
50    <wsdl:binding name="CalculatorSoapBinding" type="tns:CalculatorPortType">
51        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
52
53        <wsdl:operation name="add">
54            <soap:operation soapAction="add" style="document"/>
55            <wsdl:input name="add">
56                <soap:body use="literal"/>
57            </wsdl:input>
58            <wsdl:output name="addResponse">
59                <soap:body use="literal"/>
60            </wsdl:output>
61        </wsdl:operation>
62
63    </wsdl:binding>
64
65    <wsdl:service name="Calculator">
66        <wsdl:port name="CalculatorPort" binding="tns:CalculatorSoapBinding">
67            <soap:address location="http://localhost:8080/jaxws-calculator/calculator"/>
68        </wsdl:port>
69    </wsdl:service>
70
71 </wsdl:definitions>
```

Fig. 2: An example of a fine-grained Web service interface provided by Apache Geronimo[6].

only a single operation `translate` which is responsible for translating text from one language to another language. Although this service might be designed properly, from a strict antipattern definition, it may be considered as a fine-grained Web service.

How to find the appropriate metrics that characterize an antipattern? The most challenging issues when detecting Web service antipatterns are how to find the appropriate metrics that characterize such antipattern, and, most importantly, how to find the best *combination* of these metrics. Indeed, Most of the existing works are limited to providing definitions for Web service antipatterns and/or characterizing their common symptoms [6] [13] [22]. Recent work [13] relies on declarative rule specification where rules are manually defined to identify the key symptoms that characterize a Web service antipattern. Unfortunately, it is difficult to translate these symptoms into metrics. To make the situation worse, the same symptom may be associated with many antipattern types, which may compromise the precise identification of antipattern instances. Indeed, translating antipattern definitions from natural language to metrics is

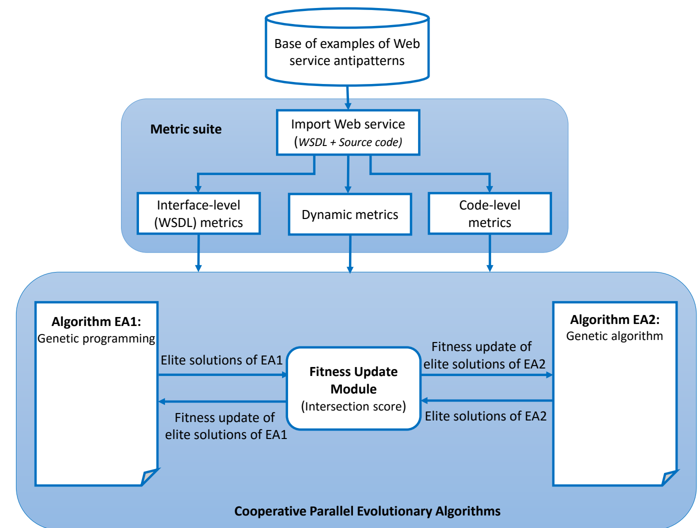


Fig. 3: The proposed cooperative parallel model scheme.

still mainly a subjective task.

How to find the appropriate metric threshold value? 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 [4] [22]. That is, for each antipattern, rules that are expressed in terms of metrics need substantial calibration efforts to find the right threshold value for each metric, above which an antipattern is said to be detected.

To address or circumvent the above mentioned issues and challenges, we introduce a cooperative parallel search-based approach to automatically detect Web service antipatterns and rank them in order of priority/severity.

4 Proposed Approach to Web Service Antipattern Detection

4.1 The Proposed Cooperative Parallel Model Schema

Figure 3 provides a high-level overview of the approach proposed. 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. As output, our approach derives a set of detection rules.

The base of examples: contains different Web service antipatterns from various application domains (e.g., weather, finance, shipping, etc.) that can be collected from different Web service search engines, such as ServiceXplorer[8], programmableweb.com[9], etc. These antipatterns were manually inspected and validated based on existing guidelines from the literature [4] [22].

Metric suite: In earlier work, we employed a set of 23 Web service interface (WSDL) metrics [14]. In this paper, we extend our metric suite to include a set of static and

[8] eil.cs.txstate.edu/ServiceXplorer
[9] programmableweb.com

1939-1374 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2502595, IEEE Transactions on Services Computing

dynamic Web service metrics. Static metrics aim at measuring the structural properties of Web services in both the interface (WSDL) and code levels, whereas dynamic metrics aim at invoking the Web services and measuring different properties, e.g., response time. Our extended metric suite is based on this variation of these metrics in order to uncover as much as possible the different antipattern symptoms:

Web service interface-level (WSDL) metrics: Table 1 summarizes the employed metrics. The first fifteen metrics (NPT-ALMS) are defined in the literature [13] [27] [28]. We also derived and defined eight other metrics (RPT-AMTP) based on existing metrics. The last three metrics, AMTO, AMTM, and AMTP, are implemented based on WordNet[10], a widely used lexical database. Each operation, port type and message identifier is tokenized using a camel case splitter. Then we assume that the greater the number of extracted tokens that exist in the WordNet database, the more meaningful the identifier is, i.e., the more semantically and syntactically sound it is. For COH we are using a well known service-oriented cohesion metric [27] namely Total Interface Cohesion of a Service (TICS) that combines four aspects of cohesion. For COUP, we used the standard coupling metric defined by Sindhgatta et al. [28].

Web service code-level metrics: In addition to WSDL metrics, our approach uses Web service code-level metrics [8] [29]. As Web service technology suggests that the Web service is accessible only through its WSDL, we use the Java™ API for XML Web Services (JAX-WS)[11] to generate the Java artifacts of the Web service. Then our approach follows a long tradition of using object-oriented metrics to evaluate the quality of the design [30]. The most widely-used metrics are those defined by Chidamber and Kemerer [30] as described in Table 2 including: Depth of Inheritance Tree (DIT), Weighted Methods per Class (WMC), and Coupling Between Objects (CBO). Our approach is based on the *ckjm* tool (Chidamber & Kemerer Java Metrics)[12]. Note that for all code-level metrics we calculate the average value for all the classes that implement the Web service under consideration.

Web service dynamic metrics: Due to the dynamic nature of Web services, we extended our metric suite to include a new dynamic metric, namely Response Time (RT) [13]. To measure the response time of a Web service, we used the SAAJ[13] standard implementation and SoapUI[14]. To reduce the impact of the network latency and physical location of a Web service, we randomly invoked at least five operations from each Web service, measured their response times, and calculate the average.

Many metric combinations are possible, so the detection rules generation process is, by nature, a combinatorial optimization problem. The number of possible solutions quickly becomes huge as the number of metrics and possible threshold values increases. A deterministic search is not practical in such cases, and hence the use of heuristic search is warranted. The dimensions of the solution space are set by the metrics, their threshold values, and logical operations between them, e.g., union (metric1 OR metric2) and intersection (metric1 AND metric2). A solution is determined by assigning a threshold value to each metric.

Our proposal is based on a parallelization at the solution level without interchanging solutions between the considered search algorithms. In fact, in each generation, the fitness values of the best solutions are updated based on an intersection score that is detailed later in Section 4.2.

TABLE 1: List of Web service interface metrics used.

Metric	Description	Metric level
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

TABLE 2: List of Web service code metrics used.

Metric	Description	Metric level
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

4.2 Parallel Evolutionary Algorithms Adaptation

This section shows how P-EA is adapted to Web service antipattern detection. To ease the understanding of this formulation, we first describe the pseudo-code of our adaptation and the present the solution encoding, the objective functions to optimize, and the employed change operators.

Algorithms 1 and 2 describe the algorithms used in our P-EA formulation. The first employs an EA based on genetic programming (GP) [17] to generate antipattern detection rules. The second is executed in parallel and employs an EA based on a genetic algorithm (GA) [18] that generates detectors from well-designed Web service examples. Both algorithms are powerful metaheuristic search optimization methods that have already shown good performance in solving several software engineering problems [31] [32].

[10] wordnet.princeton.edu
[11] http://docs.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html
[12] http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/
[13] saaj.java.net
[14] www.soapui.org

1939-1374 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2502595, IEEE Transactions on Services Computing

The basic idea of both algorithms is to explore the search space by making a population of candidate solutions, also called individuals, and evolve this population towards an "optimal" solution for a specific problem. To evaluate the solutions, the fitness function in both algorithms has two components. For the first component of the fitness function, GP evaluates the detection rules based on the achieved coverage of antipattern examples (input), while the GA evaluates the detectors by calculating the deviance from well-designed Web services (input). Then a set of best solutions are selected from both algorithms in each iteration. The algorithms interact with each other using the second component of the fitness function called the *intersection_function*, where a set of Web services, different from the ones used to produce the inputs, is evaluated using these solutions in order to maximize the intersection between the sets of detected antipatterns by each solution. Thus, some solutions will be penalized due to the absence of consensus while others will be favored due to the consensus they achieve.

A high-level view of both algorithms used in our P-EA approach is described by Algorithm 1 and 2. In the remainder of this section, we describe both algorithms.

In the initialization of the P-EA algorithm (lines 1-3 for GP and lines 1-2 for GA), our base of examples is split into ten subsets, each representing a different application domain, e.g., finance, travel, etc. One subset (*WS*) is the test dataset and the remaining subsets (*B* or *GE*) are the training datasets (the ground truth). Thus P-EA is run to detect antipatterns in the selected subset (*WS*), which is not of course part of the training set.

Lines 5-6 and 4-6 construct initial populations for GP and GA respectively, which are sets of individuals (*I*) that stand for possible solutions representing detection rules (metrics combination) for GP and detectors (artificial code that represents pseudo antipattern examples) for GA. Lines 9-20 and 7-17 encode the main GP and GA loops, respectively, which explore the search space and constructs new individuals by combining metrics for GP to generate rules and Web service elements to generate detectors for GA. In each iteration of the training process (lines 10 and 13 for GP, and lines 8 and 10 for GA), antipatterns are iteratively evaluated using rules generated by GP, and deviance from good design practices (detectors) generated by GA. As described earlier, 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 along with a deviance score from the generated detectors. After calculating the intersection between both algorithms (line 18 for GP and line 15 for GA), the fitness of the best solutions of Algorithm 1 is updated based on the fitness values of elite solutions of Algorithm 2 (line 19) and vice versa (line 16).

Then the best solution for each algorithm is saved and a new population of individuals is generated (line 22 for GP, and line 19 for GA) by iteratively selecting pairs of parent individuals from population *Pop* and applying genetic operators to them (crossover and mutation). We include both the parent and child variants in the new population. We then apply the mutation operator, with a probability score, for both parent and child to ensure solution diversity; this produces the population for the next generation. When

Algorithm 1 Pseudo code of P-EA adaptation: Genetic Programming (**GP**)

1: **Input:** M : Set of Web service quality metrics
2: **Input:** B : Base of Web service antipattern examples
3: **Input:** *WS* : Set of new Web services to evaluate
4: **Process:**
5: $R :=$ rule(M)
6: $I1 :=$ rules(R, antipattern_Type)
7: $Pop1 :=$ set_of(I1)
8: initial_population(Pop1, *Max_size*)
9: **while** $it < MAX_it$ **do**
10: **for** $all\ I \in \text{Pop1}$ **do**
11: detected_antipattern:= execute_rules(R, I1)
12: fitness(I1):=compare(detected_antipatterns, antipattern_examples)
13: **end for**
14: best_sol_Pop1 := select(Pop1, best_solutions)
15: best_sol_Pop2 := receive(best_sol_Pop1)
16: **for** $all\ I1 \in \text{best_sol_Pop1}$ **do**
17: detected_antipatterns_GP(*WS*) := execute_rules(I1, *WS*)
18: fitness_intersection(I1) := Max_intersection(detected_antipatterns_GP(WS, I1) ∩ detected_antipatterns_GA(WS, best_sol_Pop2)
19: fitness(I1) := update_fitness(fitness_intersection(I1))
20: **end for**
21: best_sol_Pop1 := best_fitness(Pop1)
22: Pop1 := generate_new_population(Pop1)
23: *it* := *it*+1
24: **end while**
25: **Output:** *DR*: antipattern detection rules

Algorithm 2 Pseudo code of P-EA adaptation: Genetic Algorithm (**GA**)

1: **Input:** *GE* : Set of good Web service examples
2: **Input:** *WS* : Set of new Web services to evaluate
3: **Process:**
4: I2 := detectors(*GE*)
5: $Pop2 :=$ set_of(I2)
6: initial_population(Pop2, *Max_size*)
7: **while** $it < MAX_it$ **do**
8: **for** $all\ I2 \in \text{Pop2}$ **do**
9: fitness(I2) := distance(detectors_I2, GE)
10: **end for**
11: best_sol_Pop2 := select(Pop2, number_solutions)
12: best_sol_Pop1 := receive(best_sol_Pop2)
13: **for** $all\ I2 \in \text{best_sol_Pop2}$ **do**
14: detected_antipatterns_GA(*WS*) := execute_detectors(I2, *WS*)
15: fitness_intersection(I1) := Max_intersection(detected_antipatterns_GP(WS, I1) ∩ detected_antipatterns_GA(WS, best_sol_Pop2)
16: fitness(I2) := update_fitness(fitness_intersection(I2))
17: **end for**
18: best_sol_Pop2 := best_fitness(Pop2)
19: Pop2 := generate_new_population(Pop2)
20: *it* := *it*+1
21: **end while**
22: **Output:** *D*: best_solution_detector

applying change operators, no individuals are exchanged between the parallel GA/GP. Both algorithms terminate when the termination criterion (maximum iteration number) is met, and return the best set of rules and detectors found so far. Developers can use the best rules and detectors to detect potential antipatterns on any new Web service. The tool ranks the detected antipatterns based on an intersection score that will be discussed in next section.

In the following, we describe the three main steps of adaptation of both GP and GA algorithms to our problem.

1) Solution representation

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

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

The antecedent of the IF statement combines some metrics and their threshold values using logic operators (AND, OR). If these conditions are satisfied by a Web service, then it is determined to be of the antipattern type featuring in the

1939-1374 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2502595, IEEE Transactions on Services Computing

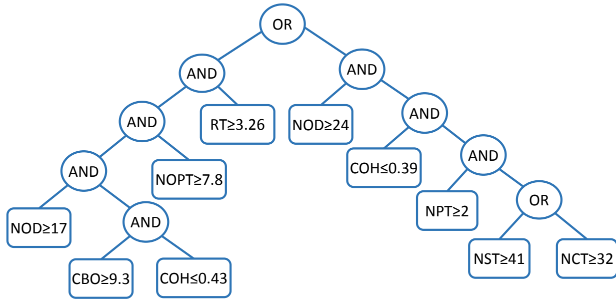


Fig. 4: Example of GP solution representation.

THEN clause of the rule. Figure 4 provides an example. More formally, each candidate solution S is a sequence of detection rules where each rule is represented by a binary tree such that:

1) Each leaf node (terminal) L represents a metric (our metric suite described earlier) and its corresponding threshold, generated randomly.
2) Each internal node (function) N represents a logic operator, either AND or OR.

We will have as many rules as types of antipatterns to be detected. In this paper, we focus on the detection of eight common types as defined in Section 2.1.

For the GA, the detectors represent artificially-generated Web services. Thus, detectors are represented simply as a vector where each element is a (metric, threshold) pair that characterises the generated Web service. See Figure 5 for an example.

NOD=9	COH=0.2	NPT=1	CBO=6	...	NST=12

Fig. 5: Example of GA solution representation.

2) Evaluation functions

GP fitness function: For GP, our fitness function aims at maximizing the number of detected antipatterns in comparison to the expected ones in the base of examples (coverage) in terms of precision and recall. Formally,

$$Coverage_{GP} = \frac{\frac{\sum_{i=1}^{n} a_i}{t} + \frac{\sum_{i=1}^{p} a_i}{p}}{2} \in [0,1] \quad (1)$$

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.

GA fitness function: For GA, we seek to optimize the following two objectives: 1) maximize the generality of the detector by minimizing the similarity with good design practices, 2) minimize the overlap (similarity) between detectors. These two objectives define the fitness function that evaluates the quality of a detector. The fitness of a solution D (set of detectors) is evaluated as the average fitness of the included detectors. We derive the fitness of a detector d_i as a weighted average between the scores of respectively, the deviance (Dev) and the overlap (O). Formally,

$$Deviance_{GA}(d_i) = \frac{Dev(d_i) + O(d_i)}{2} \in [0,1] \quad (2)$$

Dev is measured by a deviance score between the metric values of a detector d_i and those of all the Web services in the base of examples B. Intuitively, for each considered metric in our metric suite, metric values outside boxplots (i.e., outliers) representing our base of examples are likely to be abnormal, and therefore, risky as they deviate from common design practices. Let $M' = \{m_1, m_2, ..., m_n\}$ and $M = \{m'_1, m'_2, ..., m'_n\}$ be respectively the current metric values of the detector d_i and the average metric values from B. Formally, $Dev(d_i) = \sum_{i=1}^{n} |m_i - m'_i| \times \frac{1}{n}$. Similarly, the overlap O_i, is measured by the difference between the metric values of the detector d_i and average metric values of all the other detectors d in the solution D. Formally, $O(d_i) = \sum_{i=1}^{n} |m_i - m_{i_D}| \times \frac{1}{n}$. All metrics are normalised in the range [0,1].

Intersection score (used by both algorithms): A set of best solutions are selected from both algorithms, in each iteration, and then executed on a new set of Web services *WS* to evaluate (test dataset). A matrix M is constructed where rows are composed by the best solutions of GP $\{SGP_i\}$, columns are composed by the best solutions of GA $\{SGA_j\}$ and each case (SGP_i, SGA_j) is defined as the average of precision P_{int} and recall R_{int} as follows:

$$M_{intersection}(SG_i, SGA_j) = \frac{P_{int}(SGP_i, SGA_j) + R_{int}(SGP_i, SGA_j)}{2} \quad (3)$$

$$P_{int}(SGP_i, SGA_j) = \frac{|\{detect(SGP_i, WS)\} \cap \{detect(SGA_j, WS)\}|}{Max\{|\{detect(SGP_i, WS)\}|, |\{detect(SGA_j, WS)\}|} \quad (4)$$

$$R_{int}(SGP_i, SGA_j) = \frac{|\{detect(SGP_i, WS)\} \cap \{detect(SGA_j, WS)\}|}{actual\ antipatterns\ in\ WS} \quad (5)$$

where *detect(A, D)* returns the detected antipatterns by the algorithm A from the dataset D.

Then, the intersection score for each solution is defined as $f_{intersection}(SGP_i) = Min\left(M_{intersection}(SGP_i, SGA_{\forall j})\right)$ and $f_{intersection}(SGA_i) = Min\left(M_{intersection}(SGP_{\forall i}, SGA_j)\right)$. Therefore, the fitness of the solution S_i for GP and GA are respectively:

$$fitness_{GP}(S_i) = \frac{Coverage_{GP}(S_i) + f_{intersection}(S_i)}{2} \quad (6)$$

$$fitness_{GA}(S_i) = \frac{Deviance_{GA}(S_i) + f_{intersection}(S_i)}{2} \quad (7)$$

For GA, the best detectors (solution) are used to detect antipatterns. The Web service to evaluate is compared to the obtained detectors. The risk of a Web service *ws* being an antipattern is defined as the average metric difference values obtained by comparing *ws* to respectively all the detectors d_i of a set D. Formally,

$$risk(ws) = \frac{\sum_{d_i \in D} dev(ws, d_i)}{|D|} \quad (8)$$

where *dev(ws, d_i)* returns the metrics difference between *ws* and the set of detectors D. The antipatterns can then be ranked according to their risk scores. In our adaptation, we consider a Web service to be an antipattern only if the risk is more than 75 percent.

3) Genetic operators

Mutation: For GP, the mutation operator can be applied to a function node, or a terminal node. It starts by randomly

1939-1374 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2502595, IEEE Transactions on Services Computing

selecting a node in the tree. If the selected node is a terminal (metric), it is replaced by another terminal (metric or another threshold value); if it is a function (AND-OR) node, it is replaced by a new function; and if tree mutation is to be carried out, the node and its subtree are replaced by a new randomly generated subtree. For GA, the mutation operator consists of randomly changing one or more dimensions in its vector.

Crossover: We use the "standard" random single-point crossover. For GP, two parent solutions are selected, and a subtree is picked on each one. Then, the crossover operator swaps the nodes and their relative subtrees from one parent to the other. For GA, crossover combines two solutions by replacing the dimensions of the first one, from the beginning of the offspring up to the crossover point, with those of the second one, and vice versa.

5 Evaluation

This section describes the design of our empirical study and presents the results obtained from our experiments.

5.1 Research Questions

We designed our experiments to answer the following research questions:

- **RQ1.** How does our P-EA approach compare to GP, GA and random search?
- **RQ2.** To what extent can the proposed approach efficiently detect Web service antipatterns?
- **RQ3.** What types of Web service antipatterns does it detect correctly?
- **RQ4.** How does P-EA perform compared to existing Web service antipattern detection approaches (including an approach not based on search)?

5.2 Experimental Design

To evaluate our approach, we collected a set of Web services using different Web service search engines including eil.cs.txstate.edu/ServiceXplorer, programmableweb.com, biocatalogue.org, webservices.seekda.com, taverna.org.uk and myexperiment.org. Table 3 summarizes the collected services. Furthermore, so as to not bias our empirical study, our collected Web services are drawn from ten different application domains: financial, science, search, shipping, travel, weather, media, education, messaging and location. All services were manually inspected and validated to identify antipatterns based on guidelines from the literature [4] [22]. Furthermore, our dataset is available online [19] to encourage future research in the area of automated detection of Web service antipatterns.

We considered antipattern types range from eight common antipatterns, namely god object web service (GOWS), fine-grained Web service (FGWS), chatty Web service (CWS), data Web service (DWS), ambiguous Web service (AWS), redundant port types (RPT), CRUDy interface (CI), and maybe it is not RPC (MNR) (cf. Section 2.1). In our study, we employed a 10-fold cross validation procedure. We split our data into training data and evaluation data. For each fold, one category of services is evaluated by using the remaining nine categories as a base of examples (ground-truth). For instance, weather services are analyzed using antipattern instances from travel, shipping, search, science financial, media, education, messaging, and location services. We use *precision* and *recall* [33] to evaluate the accuracy of our approach. Precision denotes the ratio of true antipatterns detected to the total number of detected antipatterns, while recall indicates the ratio of true antipatterns detected to the total number of existing antipatterns.

TABLE 3: Web services used in the empirical study.

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

In general, after executing our P-EA technique the best detectors and detection rules are used to find antipatterns in new Web services. These antipatterns are ranked using a severity score defined as:

$$Severity(ws_i) = \frac{Risk(ws_i) + RuleDetect(ws_i)}{2} \quad (9)$$

where *Risk*(ws_i) is defined by Equation 8, and *RuleDetect*(ws_i) takes the value 1 if the Web service ws_i is detected by the set of rules, otherwise it takes 0. We set the risk at 0.75 to be considered as acceptable score to identify a Web service as an antipattern. We used a trial and error strategy to find this suitable threshold value after executing our approach more than 30 times. However, this threshold could adjusted by the users.

To answer **RQ1**, we investigate and report on the effectiveness of P-EA, since one of our primary novelties lies in the adoption of cooperative P-EA. To this end two independent search methods for both GP and GA, and a cooperative P-EA are implemented. For GP and GA, all searches proceed independently and the best solution is collected at the end. In P-EA it is hoped that by exchanging information among the search EAs the efficiency of the global search will increase to yield a higher-quality final solution. Furthermore, we implemented random search (RS) with the same fitness functions as P-EA. Indeed, it is important to compare our search technique to random search, since if an intelligent search method fails to outperform random search, then the proposed formulation is not adequate [34].

To answer **RQ2**, we use both recall and precision to evaluate the efficiency of our approach in identifying antipatterns.

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

To answer **RQ4**, we compared our approach with the SODA-W approach of Palma et al. [13], and with our previous approach [14]. SODA-W manually translates antipattern symptoms into detection rules and algorithms based on a

1939-1374 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2502595, IEEE Transactions on Services Computing

literature review of Web service design. In our earlier work [14] that we extend in this paper, we used the standard genetic programming algorithm to generate Web service antipattern detection rules using only Web service interface metrics. All three approaches are tested on the same benchmark described in Table 3.

5.3 Parameter Tuning and Setting

While the use of different EAs simultaneously in solving a given problem reduces the sensitivity to the chosen parameter values, still an important aspect of research on meta-heuristics lies in the selection and tuning of the algorithmic parameters to ensure fair comparison and potential replication. For GP and GA, the initial populations/solutions are completely random. The population size is fixed at 100 and the number of generations at 3,500. For GP, the max depth of the tree is fixed to 10. For P-EA and RS, the population size is 100 and number of generations 1,750. In this way, all algorithms perform 350,000 fitness function evaluations, so a fair comparison can be made. Crossover rate is set to 0.9 and 0.4 for mutation probability. We used a high mutation rate to ensure the diversity of the population and prevent premature convergence from occurring [35]. In fact, after several trial runs of the simulation, these parameter values are fixed. Indeed, there are no general rules to determine these parameters, and therefore we set the combination of parameter values by a trial-and-error method, which is commonly used by the SBSE community [36] [37].

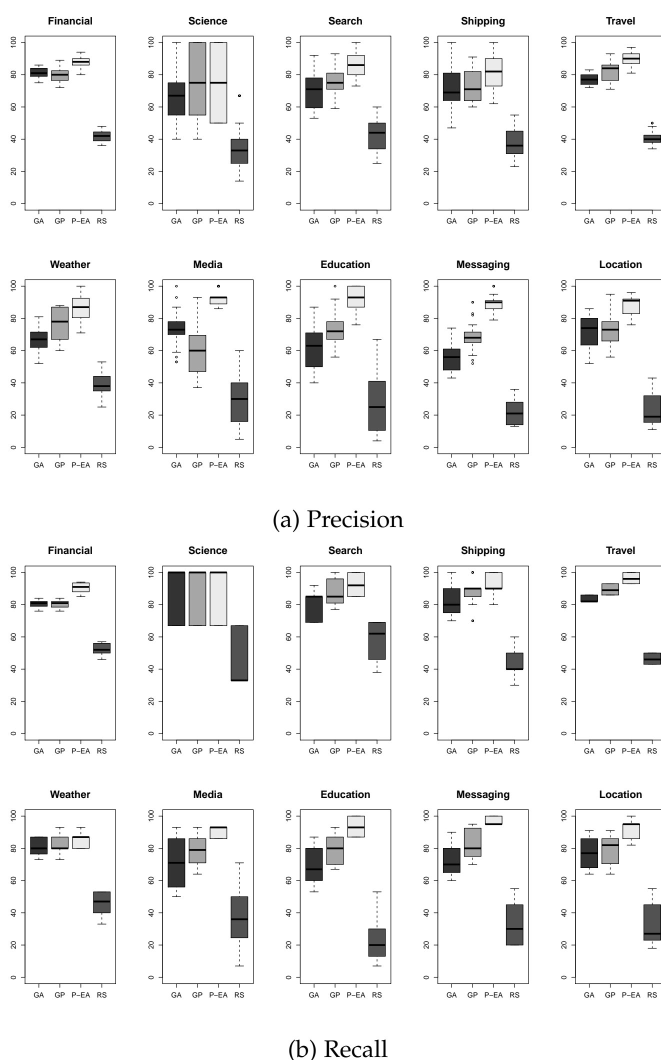
5.4 Inferential Statistical Tests Used

We used the Wilcoxon rank sum test in a pairwise fashion [38] 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 P-EA approach against GP, GA and RS. The results reported in this paper are the median values of the 31 runs. To assess the effect size, we use Cohen's d statistic [38] [36]. The effect size is considered: (1) small if $0.2 \leqslant d < 0.5$, (2) medium if $0.5 \leqslant d < 0.8$, or (3) high if $d \geqslant 0.8$.

5.5 Results

Results for RQ1. The goal of RQ1 is to investigate how well cooperative P-EA performs against each individual EA. Table 4 and Figure 6 report the comparative results. Over 31 runs, RS did not perform well when compared to P-EA in terms of precision and recall achieving values of only 36% and 41% respectively. This is mainly due to the large search-space of possible combinations of metrics and threshold values to explore. For the different categories, the statistical analysis provides evidence that our P-EA approach performs better than both the plain GA and GP.

Furthermore, we observed that P-EA was able to detect Web service antipatterns in the different categories with a high median precision and recall scores, 89% and 93% respectively. The Wilcoxon test results showed that for all 20 experiments (10 categories, 2 measures of precision and recall), P-EA was significantly better than GA, with a Cohen effect size 'high' in 18 out of 20 cases. Similarly, P-EA was significantly better than GP with a 'high' effect size in 14 out of 20 cases, and a 'medium' effect size in 4 out of 20 cases. In the other cases, the effect size was 'small' and identical in one case. On the other hand, overall, GP provides better results than GA in most of the cases (only in 3 experiments, GA was better than GP as shown in figure 6). This provides evidence that learning from antipattern examples is a better criterion than deviance from common design practices. Hence, we conclude that there is strong empirical evidence that our P-EA formulation is much more suitable and efficient than GA and GP as well as RS.



(a) Precision

(b) Recall

Fig. 6: Boxplots for the precision and recall results through 31 independent simulation runs of P-EA, GA, GP and RS.

Thus the efficiency of cooperative parallel search techniques for uncertain decision problems shown in this work provides software engineers this take-home message: "*If your decision problem is uncertain, consider using cooperative parallel search techniques to achieve a consensus.*"

Results for RQ2. The results for RQ2 are presented in Table 5. As can be seen from the table, we were able to detect antipatterns in the different categories with a median precision higher than 87%. The higher precision value for

1939-1374 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2502595, IEEE Transactions on Services Computing

TABLE 4: Statistical significance p-value (α=0.05) and effect size comparison results of P-EA against GA, GP and RS.

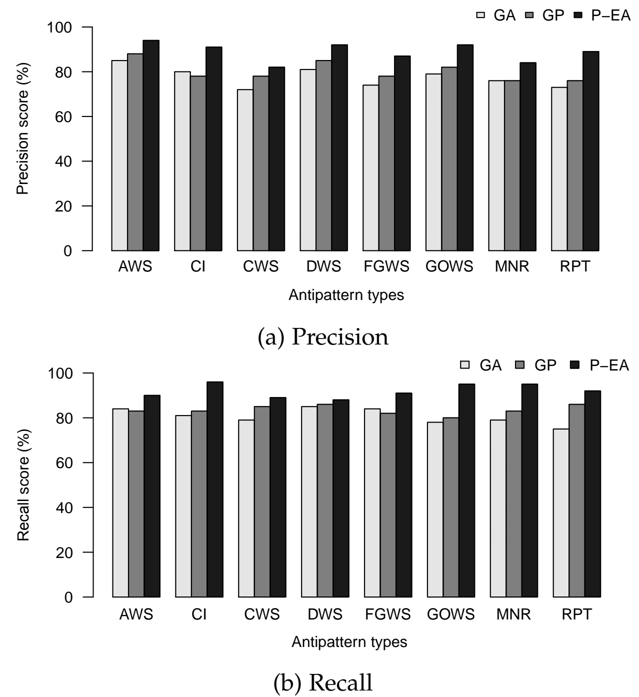
		P-EA vs GA		P-EA vs GP		P-EA vs RS	
		Precision	Recall	Precision	Recall	Precision	Recall
Financial	p-value	1.771E-06	1.166E-06	3.347E-06	1.168E-06	1.215E-06	1.209E-06
	effect size	high	high	high	high	high	high
Science	p-value	0.1129	*identical*	0.6752	0.2357	1.019E-05	3.088E-05
	effect size	small	*identical*	medium	small	high	high
Search	p-value	1.419E-05	0.00015	0.000128	0.07427	1.228E-06	1.173E-06
	effect size	high	high	high	small	high	high
Shipping	p-value	0.00205	0.00198	0.009809	0.04906	1.225E-06	1.043E-06
	effect size	high	high	medium	medium	high	high
Travel	p-value	1.164E-06	9.939E-07	5.514E-05	3.512E-06	1.193E-06	1.014E-06
	effect size	high	high	high	high	high	high
Weather	p-value	2.218E-06	0.00175	0.001537	0.02254	1.226E-06	1.065E-06
	effect size	high	high	high	medium	high	high
Media	p-value	3.151e-06	3.927e-05	1.807e-06	3.833e-05	1.229e-06	1.19e-06
	effect size	high	high	high	high	high	high
Education	p-value	1.228e-06	1.163e-06	1.077e-05	1.221e-05	1.229e-06	1.157e-06
	effect size	high	high	high	high	high	high
Messaging	p-value	1.224e-06	1.132e-06	2.623e-06	5.426e-06	1.223e-06	1.1e-06
	effect size	high	high	high	high	high	high
Location	p-value	4.945e-06	1.248e-05	1.247e-05	8.493e-06	1.223e-06	1.211e-06
	effect size	high	high	high	high	high	high

TABLE 5: Median precision and recall scores obtained for P-EA over 31 independent simulation runs.

Category	Precision (%)	Recall (%)
Financial	88	91
Science	87	92
Search	86	92
Shipping	82	90
Travel	90	96
Weather	87	91
Media	93	93
Education	93	93
Messaging	90	95
Location	91	95
Average	**89**	**93**

travel and finance (greater than 88%) can be explained by the fact that these Web services are larger than the others and contain a greater number of operations and complex types that match the GOWS antipattern. For shipping, the precision is lower (82%), but is still an acceptable score. This is due to the nature of the antipatterns involved which are typically data or chatty Web services. Indeed, some false positives are recorded for the DWS and CWS antipatterns. These antipatterns are likely to be difficult to detect using metrics alone, but using the deviance from common practices such antipatterns can be detected if we use a severity score lower than 75 percent. Similar interpretations can be made for recall. The obtained results indicate that our approach is able to achieve a recall of 92%. The highest values were recorded for travel services with 96% where most of the detected services are GOWS and AWS. The lowest recall score was recorded for the weather service (87%) which is attributable mostly to FGWS. Indeed, weather Web services typically provide one or two operations that return the temperature for a given city, which falsely matches the symptoms of FGWS.

Results for RQ3. Based on the results of Fig. 7, we observe that P-EA does not have a bias towards the detection of any specific antipattern type. As described the figure, we had an almost equal distribution of each antipattern type. On some Web services such as weather, the distribution is not as balanced. This is principally due to the number of actual antipattern types detected. Overall, all the 8 antipattern types are detected with good precision and recall scores (more



(a) Precision

(b) Recall

Fig. 7: Detection results for each antipattern type.

than 85%). Most existing guidelines/definitions [22] [13] rely heavily on the notion of size to detect antipatterns. This is reasonable for antipatterns like GOWS and FGWS that are associated with a notion of size, but for antipatterns like AWS, however, the notion of size is less important and this makes this type of anomaly hard to detect using structural information. This difficulty limits the performance of GP and GA alone in detecting this type of antipattern. Thus, we can conclude that our P-EA approach detects well all the types of considered antipatterns (RQ3).

Furthermore, we report in Table 6 a random sample of Web service antipattern instances that are detected using P-EA. For instance, the `XigniteRegistration` Web service is detected as *god object Web service (GOWS)* because it has 3 port types (*NPT = 3*) with 107 operations declared (*NOD = 107*) making the Web service very large. This service also suffers from a high response time *RT = 4.28*. Furthermore, it defines 251 complex types in its interface, thus causing it to be identified also as a *data Web service* antipattern. Similarly the Web service `XigniteQuotes` is detected as both GOWS and DWS as it has large NPT (*NPT = 3*), a large number of declared operations (*NOD = 98*) with 95 of them being accessor operations (*NAOD = 95*), a large number of messages (*NOM = 230*), relatively low cohesion (*COH = 0.4*), as well as a large number of complex types (*NCT = 39*). Furthermore, the code of `XigniteQuotes` exhibits high coupling (*CBO = 3.696*), a high response of class score (*RFC = 11.91*) and a low Measure of Functional Abstraction (*MFA = 0.076*). A client that uses such interdependent and tightly-coupled web services is exposed to all these dependencies, while all that may be desired is to use one particular operation. Successful Web service reuse is predicated on well-defined, well-understood components [22]. A murky Web service will confuse, appear less credible and reliable, and most developers will opt to look elsewhere or reimplement it.

The Web service `GHT_HotelDirectUpdateService`

1939-1374 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

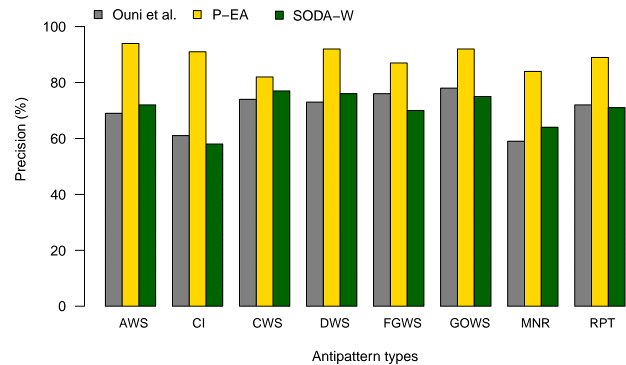
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2502595, IEEE Transactions on Services Computing

is detected as a *fine-grained Web service (FGWS)* antipattern. This Web service has only one port type (*NPT* = *1*) declaring a single operation (*NOD* = *1*) with low number of messages (*NOM* = 2). Similarly, `FlightsNearSoapApiService` and `Conversor` are also detected as *FGWS* antipatterns. A client of such a Web service must execute multiple interactions to complete a meaningful task, and poor performance results because each interaction takes significant time at each end-point. Furthermore, the client is required to know details of the Web Service to coordinate it correctly.

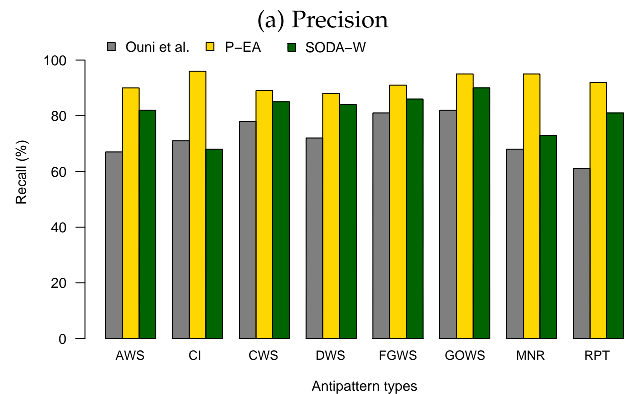
`XigniteOutlook` is also identified as an ambiguous Web service, providing several operations, messages, and types with very long signatures with ALOS = 4.9, AMTM = 0.63 and AMTO = 0.48. For instance, the interface element signatures include several baroque names, e.g., *MarketReflectionsAndWhosSpeakingAndFYIAlerts*, *GetOutlookForRangeLengthBackwardResult* and *GetOutlookForRangeLengthBackwardHttpPostOut*. Thus, the developers of this Web service appear to have ignored service interface naming principles and years of consensus on the correct way to codify software APIs. Developers are advised to write names according to common practices [39] to facilitate Web service reuse, automated analyses and human interpretation [40].

Results for RQ4. Figure 8 reports the comparison result of P-EA, Ouni et al. [14], and SODA-W. While SODA-W shows promising results with an average precision of 71% and recall of 83%, it is still less than P-EA in all the eight considered antipattern types. We conjecture that a key problem with SODA-W is that it simplifies the different notions/symptoms that are useful for the detection of certain antipatterns. Indeed, SODA-W is limited to a set of WSDL interface metrics, but ignores the source code of the Web service artifacts. In fact, service design may look promising at the interface level, but can prove to be an antipattern if the source code is not implemented well. In contrast, our approach is based on both interface and code metrics. Another limitation of SODA-W is that in an exhaustive scenario, the number of possible antipatterns to manually characterize with rules can be very large, and rules that are expressed in terms of metric combinations need substantial calibration efforts to find the suitable threshold value for each metric. By contrast, our approach needs only some examples of antipatterns to generate detection rules. Figure 8 also shows that our earlier work [14] provides lower detection results for the eight studied antipatterns with an average of 72% for both precision and recall. The lower performance can be explained by the fact that of our earlier work is based only on interface metrics that may not be able to capture all possible antipattern symptoms.

An important consideration is the impact of the size of the base of examples on detection quality. Drawn from *weather*-related Web service category, the results of Figure 9 show that our approach also proposes reasonably good detection results in situations where only a few Web service antipattern examples are available (94 instances in this case). The precision and recall scores seem to grow steadily and linearly with the number of examples, and rapidly grow to acceptable values (87%). Thus, our approach does not need a large number of examples to obtain good detection results.



(a) Precision



(b) Recall

Fig. 8: Comparison results of P-EA, Ouni et al. and SODA-W

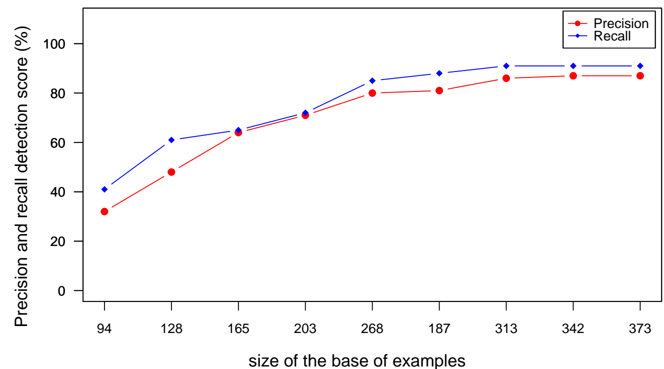


Fig. 9: The effect of the base of examples-size variation on the detection results.

6 Threats to Validity

This section describes threats to the validity of our study.

External threats to validity may arise because we did not evaluate the detection of all antipattern types. However, the eight types of Web service antipatterns we employed constitute a broad representative set of standard antipatterns. In addition, we validated our approach only on SOAP Web services, and therefore cannot generalize our results to other technologies such as REST Web services. However, a large body of web services use SOAP so the antipattern detection for this architecture is important.

Construct threats to validity are concerned with the relationship between theory and what is observed. Most of what we measure in our experiments are standard metrics such as precision and recall that are widely accepted as good proxies for the quality of antipattern and code smell

1939-1374 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2502595, IEEE Transactions on Services Computing

TABLE 6: A sample of Web service antipattern instances detected using P-EA.

WSDL	Service name	GOWS	FGWS	DWS	CWS	AWS	RPT	CI	MNR
http://developer.ebay.com/webservices/finding/latest/FindingService.wsdl	FindingService								
http://eil.cs.txstate.edu/ServiceXplorer/wsdl_files/14890_United_States_bankdemo.wsdl	BankingService								
http://www.xignite.com/xregistration.asmx?wsdl	XigniteRegistration	✓		✓					
http://www.xignite.com/xOutlook.asmx?WSDL	XigniteOutlook					✓			
http://eil.cs.txstate.edu/ServiceXplorer/wsdl_files/7242_Russian_Federation_currency.wsdl	Service1		✓						
http://www.xignite.com/xReleases.asmx?WSDL	XigniteReleases				✓				
http://webservices.lb.lt/ExchangeRates/ExchangeRates.asmx?WSDL	ExchangeRates					✓			
https://esb.alfabank.kiev.ua:8243/services/Currency?wsdl	Currency								✓
http://www.xignite.com/xBATSLastSale.asmx?WSDL	XigniteBATSLastSale							✓	
http://www.xignite.com/xcompensation.asmx?WSDL	XigniteCompensation				✓				
http://eil.cs.txstate.edu/ServiceXplorer/wsdl_files/17528_Ireland_currencyservice.wsdl	CurrencyService		✓						
http://www.xignite.com/xMaster.asmx?WSDL	XigniteMaster			✓	✓				
http://webservices.eurotaxglass.com/wsdl/specification.wsdl	SpecService				✓				
http://www.xignite.com/xanalysts.asmx?WSDL	XigniteAnalysts	✓			✓				
http://www.xignite.com/xQuotes.asmx?WSDL	XigniteQuotes			✓					
https://taxrequest.exactor.com/request/soap/?wsdl	ExactorTaxService							✓	
https://api.taxcloud.net/1.0/?wsdl	TaxCloud	✓							
http://apps.hha.co.uk/mis/api.asmx?WSDL	Api	✓							
https://www.myshawtracking.ca/otsWebWS/services/OTSWebSvcs/wsdl/OTSWebSvcs.wsdl	OTSWebSvcsService					✓			
http://vlbapi.bvdep.com/VlbOnlineAPI.asmx?WSDL	VlbNewOnlineApi	✓							
http://www.carrierrate.com/RateQuoteService/service.asmx?WSDL	Service	✓				✓			
http://shippingapi.ebay.cn/production/v3/orderservice.asmx?WSDL	OrderService					✓			
http://www.elguille.info/Net/WebServices/CelsiusFahrenheit.asmx?WSDL	Conversor		✓						
http://climhy.lternet.edu/wambam/soap_server.pl?wsdl=climdb_raw	climdb_raw		✓						
http://eil.cs.txstate.edu/ServiceXplorer/wsdl_files/13868_United_States_ShowWeatherSvc.wsdl	ShowTheWeather		✓						
http://eil.cs.txstate.edu/ServiceXplorer/wsdl_files/18249_United_States_FastTrack.wsdl	FlightExplorerFastTrack				✓				
http://eil.cs.txstate.edu/ServiceXplorer/wsdl_files/service43.wsdl	USWeather								
http://eil.cs.txstate.edu/ServiceXplorer/wsdl_files/11200_United_States_FastWeather.wsdl	DOTSFastWeather				✓	✓			
http://eil.cs.txstate.edu/ServiceXplorer/wsdl_files/service38.Accounts.wsdl	DOTSFastWeather					✓			
http://climhy.lternet.edu/wambam/soap_server.pl?wsdl=climdb_agg	climdb_agg		✓						
http://eil.cs.txstate.edu/ServiceXplorer/wsdl_files/service60.Utility.wsdl	WorldWeatherByICAO					✓			
http://eil.cs.txstate.edu/ServiceXplorer/wsdl_files/10736_United_States_WeatherForecast.wsdl	WeatherForecast					✓			
https://api.flightstats.com/flex/schedules/docs/v1/lts/soap/scheduledFlightsService.wsdl	ScheduledFlightsV1SoapService								
http://toolbox.webservice-energy.org/TOOLBOX/WSDL/AIP3_PV_Impact/AIP3_PV_Impact.wsdl	AIP3_PV_Impact						✓		
https://api.flightstats.com/flex/flightstatus/soap/v2/flightsNearService?wsdl	FlightsNearSoapApiService		✓						
https://www.platforma-broker.ro/ws/travel.svc?wsdl&wsdl	TravelService								
http://weather.terrapin.com/axis2/services/HurricaneService?wsdl	HurricaneService		✓						
http://webservices.sabre.com/wsdl/sabreXML1.0.00/GHT/HotelDirectUpdate.wsdl	GHT_HotelDirectUpdateService		✓						
http://tpb1.cangooroo.net/ws/2013/hotel_a.asmx?wsdl	Hotel_a	✓							✓
http://flytour-ad-01.dualtec.com.br:4122/FTV_Hotel/Venda.svc?wsdl	VendaService							✓	
https://wcffs.travelexplorer.com.br/FlightService.svc?wsdl	FlightService	✓			✓				
https://api.flightstats.com/flex/alerts/docs/v1/lts/soap/flightAlertsService.wsdl	FlightAlertsV1SoapService								✓
http://www.flyfrontier.com/f9_services/wordwheel/wordwheellocal.asmx?WSDL	WordWheelLocal					✓			
http://xml.bookingengine.es/webservice/OTA_HotelAvail.asmx?wsdl	OTA_HotelAvail	✓						✓	

detection solutions [13] [12] [14]. Another threat is related to the corpus of antipattern examples, as developers may not all agree if a candidate Web service is an antipattern or not. Since this is one of the first attempts to address this problem of automated detection of web service antipatterns, there is no currently established state of the art in terms of automated detection. Indeed, we found little literature to guide us on what we should consider to constitute Web service antipatterns [22] [23] [13]. In addition, various threshold values were used in our experiments based on trial-and-error, however these values can be configured once then used independently from the Web services to evaluate. Another threat is related to the automatic generation of code using `JAX-WS`. While using other tools (such as `WSDL2Java`) may yield different code, this cannot bias our results as we are using the same tool for all the experiments.

7 Related work

Detecting and specifying antipatterns in SOA and Web services is a relatively new field. Only few works have addressed the problem of SOA antipatterns. The first book in the literature was written by Dudney et al. [22] 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 [4]. Furthermore, Král et al. [7] listed seven "popular" SOA antipatterns that violate accepted SOA principles. In addition, a number of research works have addressed the detection of such antipatterns. Recently, Moha et al. [41] have proposed a rule-based approach called SODA for SCA systems (Service Component Architecture). Later, Palma et al. [13] extended this work for Web service antipatterns in SODA-W. The proposed approach relies on declarative rule specification using a domain-specific language (DSL) to specify/identify the key symptoms that characterize an antipattern using a set of WSDL metrics. In another study, Rodriguez et al. [42] [24] and Mateos et al. [23] provided a set of guidelines for service providers to avoid bad practices while writing WSDLs. Based on some heuristics, the authors detected eight bad practices in the writing of WSDL for Web services. In other work [43], the authors presented a repository of 45 general antipatterns in SOA. The goal of this work is a comprehensive review of these antipatterns that will help developers to work with clear understanding of patterns in phases of software development and so avoid many potential problems. Mateos et al. [44] have proposed an interesting approach towards generating WSDL documents with less antipatterns using text mining techniques. Recently, Ouni et al. [14] proposed a search-based approach based on standard GP to find regularities, from examples of Web service antipatterns,

1939-1374 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2502595, IEEE Transactions on Services Computing

to be translated into detection rules. However, the proposed approach can deal only with Web service interface metrics and cannot consider all Web service antipattern symptoms. Similar to [13], the latter consider neither deviation from common design practices nor code metrics, which leads to several false positives.

Unlike service-oriented computing, there is an extensive body of research that tries to detect object oriented antipatterns and code smells [10] [9] [11] [12] [45]. One of the first attempts to automate code smell detection for Java programs was conducted by van Emden and Moonen [46]. The authors examined a list of code smells and found that each of them is characterized by a number of "smell aspects" that are visible in source code entities such as packages, classes, methods, etc. A given code smell is detected when all its aspects are found in the code. The identified aspects are mainly related to non-conformance to coding standards. Later, Marinescu el al. [9] have proposed a mechanism called "detection strategy" to detect object oriented code smells by formulating metric-based rules that capture deviations from design principles and heuristics. Similarly to SODA-W [13], Moha et al. [10] proposed a description of antipattern symptoms using a domain-specific-language (DSL) for their antipatterns detection approach called DECOR. They proposed a consistent vocabulary and DSL to specify antipatterns based on their review of existing work on code smells found in the literature. To describe antipattern symptoms different notions are involved, such as class roles and structures. Symptoms descriptions are later mapped to detection algorithms. Recently, Ouni et al. [11] proposed a search-based approach to detect code smells in object oriented software systems, which was the first approach to infer detection rules from examples. Kessentini et al. [12] proposed an extended version using a cooperative parallel search technique to detect code smells in Java systems.

However, code smell 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 technologies and metrics. Furthermore, unlike object oriented software systems, Web service source code is not open source; that is, only Web service interfaces are publicly available for clients from which only the skeleton of the code can be automatically generated. This makes the detection of such antipatterns more challenging.

8 Conclusion and Future Work

In this paper, we introduced a new SBSE approach for Web Service antipattern detection. In our cooperative parallel metaheuristic adaptation, two populations evolve simultaneously with the objective of finding consensus regarding the identification of Web service antipatterns. The proposed approach is evaluated on a benchmark of 415 Web services and eight common Web service antipattern types. Statistical analysis of the obtained results provides compelling evidence that cooperative P-EA outperforms individual population, random search, and a recent state-of-the art approach with a median precision of more than 89% and a median recall of more than 93%.

As future work, we plan to validate our approach with additional antipattern types in both individual Web services and business process in order to investigate more thoroughly the general applicability of our methodology. In addition, we plan to include other SOAP static and dynamic metrics and empirically investigate their effect on the quality of Web services. We also plan to extend the approach to detect business process antipatterns in SBS in addition to individual Web service antipatterns. Finally, another promising research direction is to automate the correction, through refactoring, of the detected antipatterns.

References

[1] M. D. Hansen, *SOA Using Java Web Services*. Pearson Education, 2007.

[2] 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, no. 1, pp. 9–29, 2005.

[3] J. M. Rodriguez, M. Crasso, C. Mateos, and A. Zunino, "Best practices for describing, consuming, and discovering web services: a comprehensive toolset," *Software: Practice and Experience*, vol. 43, no. 6, pp. 613–639, 2013.

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

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

[6] J. Král and M. Žemlička, "Crucial service-oriented antipatterns," *International Journal On Advances in Software*, vol. 2, no. 1, pp. 160–171, 2009.

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

[8] J. L. O. Coscia, M. Crasso, C. Mateos, and A. Zunino, "Estimating web service interface quality through conventional object-oriented metrics," *CLEI Electron. J.*, vol. 16, no. 1, 2013.

[9] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," *2013 IEEE International Conference on Software Maintenance*, vol. 0, pp. 350–359, 2004.

[10] N. Moha, Y. Gueheneuc, L. Duchien, and A. Le Meur, "Decor: A method for the specification and detection of code and design smells," *Software Engineering, IEEE Transactions on*, vol. 36, no. 1, pp. 20–36, Jan 2010.

[11] A. Ouni, M. Kessentini, H. Sahraoui, and M. Boukadoum, "Maintainability defects detection and correction: a multi-objective approach," *Automated Software Engineering*, vol. 20, no. 1, pp. 47–79, 2013.

[12] 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, no. 9, pp. 841–861, 2014.

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

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

[15] E. Alba, *Parallel Metaheuristics: A New Class of Algorithms*. Wiley-Interscience, Aug. 2005.

[16] T. Back, *Evolutionary algorithms in theory and practice*. Oxford Univ. Press, 1996.

[17] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. MIT press, 1992, vol. 1.

[18] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman, 1989.

[19] "Experimental data," https://github.com/ouniali/WSantipatterns, accessed: 2015-08-14.

[20] R. Chinnici, M. Gudgin, J.-J. Moreau, J. Schlimmer, and S. Weerawarana, "Web services description language (wsdl) version 2.0 part 1: Core language," *W3C working draft*, vol. 26, 2004.

[21] M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Softw. Engg.*, vol. 11, no. 3, pp. 395–431, Sep. 2006.

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

1939-1374 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSC.2015.2502595, IEEE Transactions on Services Computing

[23] C. Mateos, A. Zunino, and J. L. O. Coscia, "Avoiding WSDL Bad Practices in Code-First Web Services," *SADIO Electronic Journal of Informatics and Operational Research*, vol. 11, no. 1, pp. 31–48, 2012.
[24] J. M. Rodriguez, M. Crasso, A. Zunino, and M. Campo, "Automatically detecting opportunities for web service descriptions improvement," in *Software Services for e-World*. Springer, 2010, pp. 139–150.
[25] A. Heß, E. Johnston, and N. Kushmerick, "ASSAM: A Tool for Semi-automatically Annotating Semantic Web Services," in *The Semantic Web ISWC 2004*, S. McIlraith, D. Plexousakis, and F. van Harmelen, Eds. Springer Berlin Heidelberg, 2004, vol. 3298, ch. 23, pp. 320–334.
[26] C. Mateos, M. Crasso, A. Zunino, and J. L. O. Coscia, "Detecting WSDL bad practices in codefirst Web Services," *International Journal of Web and Grid Services*, vol. 7, no. 4, pp. 357–387, 2011.
[27] M. Perepletchikov, C. Ryan, and Z. Tari, "The impact of service cohesion on the analyzability of service-oriented software," *Services Computing, IEEE Transactions on*, vol. 3, no. 2, pp. 89–103, 2010.
[28] R. Sindhgatta, B. Sengupta, and K. Ponnalagu, "Measuring the quality of service oriented design," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, vol. 5900, pp. 485–499.
[29] J. Coscia, M. Crasso, C. Mateos, A. Zunino, and S. Misra, "Predicting web service maintainability via object-oriented metrics: A statistics-based approach," in *Computational Science and Its Applications ICCSA 2012*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7336, pp. 29–39.
[30] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *Software Engineering, IEEE Transactions on*, vol. 20, no. 6, pp. 476–493, 1994.
[31] M. Harman, "The current state and future of search based software engineering," in *Future of Software Engineering (FOSE '07)*. IEEE, 2007, pp. 342–357.
[32] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 11, 2012.
[33] W. B. Frakes and R. Baeza-Yates, Eds., *Information Retrieval: Data Structures and Algorithms*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992.
[34] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.
[35] E. Cantú-Paz, "A survey of parallel genetic algorithms," *Calculateurs paralleles, reseaux et systems repartis*, vol. 10, no. 2, pp. 141–171, 1998.
[36] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Software Engineering (ICSE), 2011 33rd International Conference on*. IEEE, 2011, pp. 1–10.
[37] A. Eiben and S. Smit, "Parameter tuning for configuring and analyzing evolutionary algorithms," *Swarm and Evolutionary Computation*, vol. 1, no. 1, pp. 19 – 31, 2011.
[38] J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic press, 1988.
[39] M. Blake and M. Nowlan, "Taming web services from the wild," *Internet Computing, IEEE*, vol. 12, no. 5, pp. 62–69, Sept 2008.
[40] J. M. Rodriguez, M. Crasso, A. Zunino, and M. Campo, "Discoverability anti-patterns: frequent ways of making undiscoverable web service descriptions," in *Proceedings of the 10th Argentine Symposium on Software Engineering - 38th JAIIO*, 2009, pp. 1–15.
[41] N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, Y.-G. Guéhéneuc, B. Baudry, and J.-M. Jézéquel, "Specification and detection of soa antipatterns," in *Service-Oriented Computing*. Springer, 2012, pp. 1–16.
[42] J. M. Rodriguez, M. Crasso, C. Mateos, and A. Zunino, "Best practices for describing, consuming, and discovering web services: a comprehensive toolset," *Software: Practice and Experience*, vol. 43, no. 6, pp. 613–639, 2013.
[43] M. A. Torkamani and H. Bagheri, "A Systematic Method for Identification of Anti-patterns in Service Oriented System Development," *International Journal of Electrical and Computer Engineering*, vol. 4, no. 1, pp. 16–23, 2014.
[44] C. Mateos, J. M. Rodriguez, and A. Zunino, "A tool to improve code-first web services discoverability through text mining techniques," *Software: Practice and Experience*, vol. 45, no. 7, pp. 925–948, 2015.
[45] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "Detecting bad smells in source code using change history information," in *IEEE/ACM 28th International Conference on Automated Software Engineering*. IEEE, 2013, pp. 268–278.
[46] E. van Emden and L. Moonen, "Java quality assurance by detecting code smells," in *Reverse Engineering, 2002. Proceedings. Ninth Working Conference on*, 2002, pp. 97–106.

Ali Ouni is a research Assistant Professor at Osaka University, Japan. He is a member of the software engineering laboratory (SEL). He holds a Ph.D. in computer science from University of Montreal, Canada, in 2014. For his exceptional Ph.D. research productivity, he was awarded the Excellence Award from the University of Montreal. His research interests are in software engineering including software maintenance and evolution, refactoring of software systems, software antipatterns, service-oriented computing, and the application of artificial intelligence techniques to software engineering. He served as a program commitee member and reviewer in several conferences and journals.

Marouane Kessentini is a tenure-track assistant professor at University of Michigan, Dearborn campus. He is the founder of the research group: Search-based Software Engineering@Michigan. He holds a Ph.D. in Computer Science, University of Montreal (Canada), 2012. His research interests include the application of artificial intelligence techniques to software engineering (search-based software engineering), software testing, model-driven engineering, software quality, and re-engineering. He has published around 50 papers in conferences, workshops, books, and journals including three best paper awards. He has served as program-committee/organization member in several conferences and journals.

Katsuro Inoue received the BE, ME, and DE degrees in information and computer sciences from Osaka University, Japan, in 1979, 1981, and 1984, respectively. He was an assistant professor at the University of Hawaii at Manoa from 1984-1986. He was a research associate at Osaka University from 1984-1989, an assistant professor from 1989-1995, and a professor beginning in 1995. His interests are in various topics of software engineering such as program analysis, repository mining, software development environment, and software process modeling. He is a member of the IEEE, the IEEE Computer Society, and the ACM.

Mel Ó Cinnéide holds BSc, MSc and PhD degrees in Computer Science from University College Cork, National University of Ireland and Trinity College Dublin respectively, and is a chartered engineer. He worked in the software industry for several years before moving to academia and is currently a lecturer in the School of Computer Science, University College Dublin. Dr. Ó Cinnéide has published over 45 papers in the areas of software quality and reuse, including two best paper awards. His primary research interest is currently the automated improvement of software design quality using Search-Based Software Engineering techniques.

1939-1374 (c) 2015 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.