2017 IEEE 24th International Conference on Web Services

On the Value of Quality of Service Attributes for Detecting Bad Design Practices

Hanzhang Wang, Marouane Kessentini and Taghreed Hassouna
Computer and Information Science Department
University of Michigan
Dearborn, MI, USA
firstname@umich.edu

Ali Ouni
College of Information Technology
UAE University
Al Ain, Emirates
ouniali@uaeu.ac.ae

***Abstract*—Service-Oriented Architectures (SOAs) successfully evolve over time to update existing exposed features to the users and fix possible bugs. This evolution process may have a negative impact on the design quality of Web services. Recent studies addressed the problem of Web service antipatterns detection (bad design practices). To the best of our knowledge, these studies focused only on the use of metrics extracted from the implementation details (source code) of the interface and the services. However, the quality of service (QoS) metrics, widely used to evaluate the overall performance, are never used in the context of Web service antipatterns detection. We start, in this work, from the hypothesis that these bad design practices may impact several QoS metrics such as the response time. Furthermore, the source code metrics of services may not be always available. Without the consideration of these QoS metrics, the current detection processes of antipatterns will still lack the integration of symptoms that could be extracted from the usage of services.**

In this paper, we propose an automated approach to generate Web service defect detection rules that consider not only the code/interface level metrics but also the quality of service attributes. Through multi-objective optimization, the proposed approach generates solutions (detection rules) that maximize the coverage of antipattern examples and minimize the coverage of well-designed service examples. An empirical validation is performed with eight different common types of Web design defects to evaluate our approach. We compared our results with three other state of the art techniques which are not using QoS metrics. The statistical analysis of the obtained results confirm that our approach outperforms other techniques and generates detection rules that are more meaningful from the services' user perspective.

***Keywords*-Web service design; antipatterns; quality of service; multi-objective optimization**

I. Introduction

Service-Oriented Architecture (SOA) enables high-value business services in an efficient, reusable and flexible way. Web services are as a self-contained black box which logically represent business activity. A service can be composited or invoked by other services through standard protocols. Therefore, client users can reduce development workload based on its high reusability and extensibility. Through a well-designed interface, users can reuse the existing Web services easily and efficiently.

To ensure their popularity and increase their usability, Web services must be designed and implemented properly. There are good quality principles proposed for service-oriented design, such as loose coupling, composability, and flexibility. However, various reasons could lead to the violations of these principles, such as deadline pressure, lack of design experience and changing requirement. These violations the existence of bad design/implementation practice, are known as antipatterns [1]. Ignoring these antipatterns could lead to maintainability, extensibility, or usability issues.

Recent studies addressed the problem detecting the antipatterns of Web services automatically using various methodologies [2], [3], [4], [5]. The common idea of these techniques is to generate detection rules, mainly based on the interface or code-level metrics of the bad-designed Web services. These metrics are extracted from the interface or the code skeleton of the Web service. Thus, current practices suggest that developers can evaluate the design quality based only on the static information exposed by the Web service provider extracted from the interface and implementation/source code details. However, these metrics, such as number of operations, coupling and cohesion, may not be sufficient to identify bad design practices.

The non-functional attributes of quality of services (QoS) such as response time, availability and reliability are considered part of the major concerns in the management of Web services. The clients compare QoS measurements to select the best service, from a list of competing ones, that may satisfy their requirements. In the real world, developers usually seek for the Web services that have not only a well-designed structure, but also a good QoS performance from the user respective. To the best of our knowledge, existing antipatterns detection studies focused only on the use of metrics extracted from the implementation details (source code) of the interface and the services. However, the quality of service (QoS) metrics, widely used to evaluate the overall performance, are never used in the context of Web service antipatterns detection. We start, in this work, from the hypothesis that these bad design practices may impact

978-1-5386-0752-7/17 $31.00 © 2017 IEEE
DOI 10.1109/ICWS.2017.126

several QoS metrics such as the response time. Furthermore, the source code metrics of services may not always be available. Without the consideration of these QoS metrics, the current detection processes of antipatterns will still lack the integration of symptoms that could be extracted from the usage of services.

In this paper, we propose an antipattern detection approach based on a combination of dynamic QoS attributes and the structural information of Web service (static interface/code metrics). In our approach, a multi-objective algorithm NSGA-II [6] is implemented to generate the best detection rule sets that maximize the coverage of Web service antipattern examples and minimize the detection of well-designed Web service design examples. In addition to regular code/interface structural metrics, we introduced 9 QoS metrics to detect antipatterns namely, response time, availability, throughput, successability, reliability, compliance, best practices, latency, documentation.

We performed an empirical study of our approach on 500 Web services from a QoS benchmark. We evaluated how well our approach can detect refactoring opportunities comparing to the state-of-the-art techniques [5], [4], [3] that use only static structural metrics. The statistical analysis of the results confirms our hypothesis that the integration of QoS metrics may better characterize the symptoms of antipatterns. It also demonstrates that the accuracy of our approach in service antipatterns detection outperform existing studies, with a precision score of 91% and a recall score of 86%.

The remainder of this paper is as follows. Section 2 provides the relevant background and a motivating example to understand the challenges of this work. Section 3 describes our approach overview. Section 4 presents the problem adaptation details and the algorithm to solve it. Section 5 is dedicated to the empirical study and result analysis. Threats to validity are discussed in Section 6, while section 7 reviews the related work. Section 7 concludes and presents some possible future work.

II. Background and Motivating Example

In this section, we introduce some relevant background, and present a real-world motivating example for our proposed work.

A. Background

1) Web service Antipatterns: The service antipatterns represent bad design and implementation practices/behaviors. These poor solutions to recurring design problems usually cause maintainability and usability issues [1]. They may be introduced during the development cycle unintentionally due to human factors or limited resources. Different types of antipattern [3], [4] have been proposed and studied in the literature[1], [7]. Popular types of service antipatterns include as follow:

God object Web service (GOWS): This antipattern implements a multitude of methods related to different business and technical abstractions in a single service.

fine-grained Web service (FGWS): It is a too fine-grained service whose overhead (communications, maintenance, and so on) outweighs its utility.

Chatty Web service (CWS): Chatty Web service represents an antipattern where a high number of operations, typically attribute-level setters or getters, are required to complete one abstraction.

Data Web service (DWS): A data Web service usually deals with very small messages of primitive types and may have high data cohesion.

Ambiguous Web service (AWS): It is an antipattern where developers use ambiguous or meaningless names for denoting the main elements of interface elements (e.g., port types, operations, messages).

Redundant PortTypes (RPT): This is an antipattern where multiple portTypes are duplicated with the similar set of operations.

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

Maybe It is Not RPC (MNR): This is an antipattern where the Web service mainly provides CRUD-type operations for significant business entities.

2) Quality of service(QoS): Quality of service (QoS) is a combination of several service qualities or properties, such as response time and availability. With the right control of QoS, the quality service product can fulfill client expectations and achieve customer satisfactions. In this paper, nine common QoS metrics are used to identify refactoring opportunities as defined in section III-A.

B. Motivating Example

In this section, we illustrate a real-world example of god object Web service(GOWS) antipattern provided by Oracle Taleo[1]. Oracle Taleo is a famous talent acquisition service which enables companies to easily source, recruit or manage talents. The antipattern in this example, GOWS, has a key characteristic that it implements uncohesive operations of many core business or/and technical abstractions. Figure 1 shows the interface[2] of Oracle Taleo which contains a large amount of operations for different business abstractions. In this example, total of 127 operations are implemented within a single port type. These operations represent different functionality aspect of the service, such as administration, employee management, task information, interview management, job entry, job allocation, and so on. For instance, $createUser()$ is an administrative operation that creates new

[1]Oracle Taleo: https://tbe.taleo.net/

[2]Oracle Taleo Interface: https://tbe.taleo.net/wsdl/WebAPI.wsdl

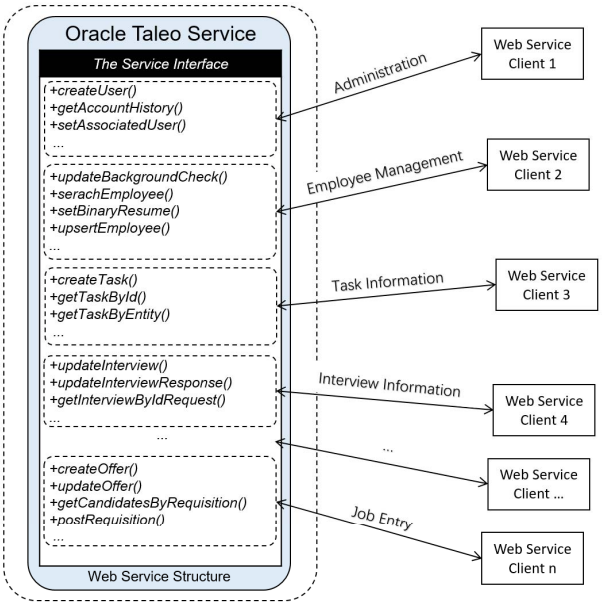

Figure 1: An example of god object Web service provided by Oracle Taleo.

authorized user of the service, $updateBackgroundCheck()$ attempts to update the background check data for employee management, while $getCandidatesByRequisition()$ lists the potential candidates who fit for a specific requisition. These operations are desired and used by different service users, e.g., companies who want to hire or companies who provide talents.

When analyzing the QoS attributes related to the Taleo service, it maybe easy to identify and cofirm also a GOWS antipattern. According to the QWS dataset[3], there is only a little documentation (i.e. description tags) to help developers understand all the operations of Taleo service. Furthermore, the Taleo service takes long ($latency = 232.46ms$) to process any request, and the availability of the services is relatively low (40% of the error messages out of the total messages). Overall, these symptoms of GOWS, could cause difficulties of reusability or reduce the practical value and popularity of the service.

To address or circumvent the above-mentioned challenges, we propose a multi-objective heuristic-based approach which automatically detects Web service antipatterns by combining dynamic QoS metrics with static code/interface ones as detailed in the next section.

[3]The QWS Dataset:http://www.uoguelph.ca/ qmahmoud/qws/

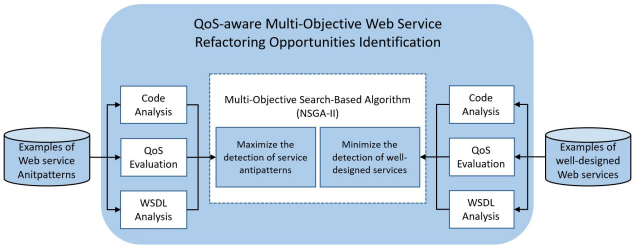

Figure 2: Approach Overview

III. Web Service Antipatterns Detection Using Dynamic QoS Metrics

A. Approach Overview

Given a set of possible Web service metrics, there are many ways that could be combined to generate detection rules. This problem is an NP-hard problem by nature, therefore it make sense to consider a meta-heuristic search-based approach. Also, our problem requires a search for a solution which balances different conflicting objectives to generate rules suitable for different scenarios.

Figure 2 shows an overview of our approach to the QoS-aware Web service refactoring opportunities identification problem. It targets to explore the large search space and finds a set of optimal detection rules, by combining metrics and their threshold values. The output is a set of detection rules which are the optimal solutions to two conflicting objectives of the search-based algorithm. The approach takes two sets of Web service examples: one set contains service antipattern examples and another has well-designed service examples. These example sets are selected from the QWS dataset[3], which includes a set of 2,507 Web services and their QWS measurements. We took a sample of 500 services from this dataset, and manually inspect and validate the antipatterns of these services based on the existing guidelines as detailed later in the experiments section.

The approach processes and calculates the metrics of each service in the sets through: (*i*) QoS Evaluation: measures real-time metrics from the services or documentation metrics from the interface file, (*ii*) Interface Analysis, parses the interface source though tree walking up the XML hierarchy to extract the Web service structure data (e.g., operation, message, and input/output), then calculates the interface level metrics, and (*iii*) Code Analysis, extracts the Web service code skeleton and uses typical object-oriented metrics to evaluate. The metric suite used in this work, which contains a total of 49 metrics, as described in the section III-A.

The metric data of the service sets are passed as inputs to multi-objective algorithm. The search-based algorithm - NSGA-II[6], generates, evaluates and selects antipattern detection rules based on the following objectives: (*i*) Maximizing detection of the service antipatterns, and (*ii*) Minimizing the detection of well-designed services. The details of this

step and algorithm are described in section III-B.

Quality metrics can be used to extract the semantic and structural attributes of the Web services. These quality indicators can then be used to quantitatively track and evaluate the design patterns of Web Services architecture. The antipatterns detection process usually involves finding the fragments of the design which violate these metrics. In previous work [5], we used a set of static Web service metrics from interface and code level[3]. The static metrics aim at measuring the structural information of Web services in different levels. Table I describes all of the metrics that are being used in this work.

Web service QoS metrics: To detect antipattern from the QoS aspect, we introduced 9 popular metrics from the literature [8] (from Response to Documentation in Table I). Documentation, compliance, and best practices are static metrics extracted based on interface level to extend our static metrics, they measure the usability of the web service interface from QoS aspect. Response, availability, throughput, successability, reliability and latency are dynamic metrics which measure the web service overall performance and experience. In this work, we use the QWS dataset[3], a widely used QoS benchmark in field of Web services [9].

Web service interface-level (WSDL) metrics: There are fifteen metrics used in this work from the interface level (from ALPS to RPT in Table I). These metrics measure design concepts from interface type, message, operation and Port type levels. Most metrics are calculated directly based on the information of Web service interface description file. For AMTO, AMTM, and AMTP, they are implemented by comparing the tokenized identifiers of ever operation, port type and message with lexical database, WordNet[4].

Web service code-level metrics: Web service only exposes the interface for clients to use while the source code is not available to access. In this work, code-level metrics (from Ca to CC in Table I) are extracted from the service code skeletons which are generated by JAX-WS[5] (a Java™API) for XML Web services. The ckjm tool[6] is used to extract these metric to reflect design quality from a deeper level of Web service.

The detection rules generated by our approach are composited by the metrics mentioned above. The dimensions of the solution space are set by the metrics associated with greater/less than, their threshold values, and logical operations between them, e.g., union ($metric1 > a \ OR \ metric2 < b$) and intersection ($metric1 > a \ AND \ metric2 < b$). A solution is a composited logical expression by multiple metrics, e.g., $metric1 > a \ AND \ (metric2 < b \ OR \ metric2 < b)$. By nature, this is a combinatorial optimization problem with a large search base(number of possible solutions is huge). A heuristic search algorithm is desired in this problem. Furthermore, since we also try to generate detection rules that can satisfy different detection strictness, multi-objective evolutionary algorithm - NSGA-II [6] is selected in this work and details are presented in section III-B.

[4]WordNet: http://wordnet.princeton.edu/
[5]JAX-WS: http://docs.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html
[6]ckjm tool: http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/

Table I: List of Web service interface metrics used.

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

B. Problem Formulation and Solution Approach

In multi-objective problems, the goal is to find solutions that can satisfy all objectives at the same time. However, it is hard to find a solution that is perfect for all objectives in the real world problems due to the conflicting relationships between them. Therefore, for many real world problems, we select multi-objective search-based algorithm to find a set of alternative solutions, rather than a single solution as result. Non-dominated sorting genetic algorithm (NSGA-II) [6] has shown high performance and efficiency in many software engineering problems [10]. The basic idea of the algorithm is to explore the search space by making a population of candidate solutions, also called individuals, and evolve this population towards an "optimal" solution.

1) Solution Representation: The candidate solutions we seek in this problem are the antipattern detection rules. Each solution is a logical expression, and the antipattern is detected while the condition in the logical expression is satisfied.

The logical expression is encoded in a tree-based structure and connects every metric with its threshold value using a logic operator ("AND" or "OR"). If the expression is satisfied by a Web service, then it is determined to be of the antipattern type associated with this solution. A solution is randomly generated at the beginning, and can be "mutated" by itself or "crossovered" with another solution to generate new ones. Formally, each candidate solution S is a sequence of detection rules where each rule is represented by a binary tree such that:

1) The Root R and each internal node N represents a logic operator to connect other nodes or leaf, either AND or OR.
2) Each leaf node L represents a metric (from the metrics described in section III-A) and its corresponding threshold (generated randomly among the range of the metric).

Each solution represents a detection rule for one specific type of service defects, and each execution of the approach only generate the solution set which is used for one antipattern. In this paper, we focus on detecting eight popular types as defined in Section II-A1.

2) Fitness Functions: The quality of each solution is determined by the fitness functions in multi-objective problems. Each fitness function evaluates one objective by calculating a specific value that is desired to be either minimized or maximized for a solution. In this problem, we aim to optimize the following two finesse functions: (*i*) Maximizing the coverage of antipattern examples. (*ii*) Minimizing the detection of good design practice examples of Web services. The collected examples of well-designed Web services, antipatterns and the metrics of these services are taken as inputs of NSGA-II. At the different iterations, each solution (detection rules) is applied to both example sets, and evaluated by the fitness functions. Analytically speaking, the formulation of the multi-objective problem can be state as follows:

$$\begin{cases} \max f_1(x) = \dfrac{\dfrac{||DCS(x)|| \cap ||ECS||}{||ECS||} + \dfrac{||DCS(x)|| \cap ||ECS||}{||ECS||}}{2} \\ \min f_2(x) = \dfrac{\dfrac{||DCS(x)|| \cap ||EGE||}{||EGE||} + \dfrac{||DCS(x)|| \cap ||EGE||}{||DCS(x)||}}{2} \end{cases}$$

where $||DCS(x)||$ is the cardinality of the set of detect antipatterns by the solution x, $||ECS||$ is the cardinality of the set of antipattern examples, and $||EGE||$ is the cardinality of the set of well-designed service examples. These two fitness functions drive the algorithm to search for the optimal solutions by comparing the list of detected antipatterns with the expected ones from the base of examples along with the percentage of covered well-designed examples. Once the bi-objective trade-off Pareto front is generated, service developers/users can select preferred detection rule from the best solution rules to detect potential antipatterns on any new Web service.

IV. Validation

This section describes the empirical study, experiment settings, and presents the results obtained from our experiments.

A. Research Questions

To validate our approach, several experiments are designed to answer the following research questions:

- **RQ1.** To what extent does QoS attributes improve the antipattern detection of Web services?
- **RQ2.** How does our multi-objective approach compare to random search and mono-objective algorithm aggregating both objectives?
- **RQ3.** How does approach perform compared to other existing Web service antipattern detection approaches [4], [3] who did not consider the use of QoS attributes?

B. Experimental Setup

To evaluate our approach, we select a popular QoS benchmark, the QWS dataset[3] which includes a set of 2,507 Web services and their QWS measurements. We randomly selected a sample of 500 available services from the dataset, extract their interface file and code skeleton, and manually inspect and validate the antipatterns of these services based on the existing guidelines [7]. To avoid possible biases in the empirical study, we select services covered different sizes, QoS performance, and application categories (such as financial, science, travel, weather, etc.). Table II summarizes some statistics about the selected Web services.

In this work, we validated our approach on eight common antipattern types as describe in section II-A1, namely,

Table II: Statistics of the used 500 Web services.

	NOD	NOM	NCT	Response	Latency
Max	231	462	287	3768.33 ms	1991 ms
Min	1	1	0	46.05 ms	0.33 ms
Average	14.43	30.44	21.40	343.10 ms	52.84 ms

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). We used 10-fold cross-validation method to evaluate our approach. Therefore, in the experiments, 450 services are selected as training examples(ground-truth) to execute the algorithm, and the rest 50 services are used as the test set. Precision and recall are used to evaluate the accuracy and effectiveness of our approach. Precision represents the ratio of correct antipatterns detected divided by the total number of detected antipatterns, and recall denotes the ratio of correctly detected antipatterns divided by the total number of expected antipatterns in the test set.

While comparing different search-based evolutionary algorithms, changing parameters could lead to completely different results (e.g., low number of iteration). To ensure fair comparisons, we used the same parameters for the all the evolutionary algorithm experiments as following: $PopulationSize = 300$, $MaxIteration = 1000$, $MaxDepth_{Solution} = 10$, $R_{crossover} = 0.8$, $P_{mutation} = 0.1$. We used a high number of population size due to the size of search space and solution combination, a small population size leads to low diversity in our approach.

To answer **RQ1**, we investigated the effectiveness of using QoS metrics on different types of service antipattern, since this is the main originality of our approach. To this end, we compared the results of our approach, with the results of the approach using the same algorithm, settings, and training/testing sets, but only based on interface-level and code-level metrics as described in Table I.

To answer **RQ2**, we investigated the efficiency of using NSGA-II and our problem formulation. We compared our approach to random search and mono-objective genetic algorithm. Since another novelty of our approach is using multi-objective optimization search-based techniques, it is important to compare with the random search (RS) to prove that the adaptation is adequate. Therefore, we implemented a random search using same training and test sets. Further more, an genetic algorithm is implemented with an aggregated fitness function (average of our two fitness functions) to validate if our multi-objective approach is able to improve the detection process.

To answer **RQ3**, we compared our approach with state-of-the-art detection approaches: a search-based approach from [4] and SODA-W of [3]. All three approaches were tested on the same services examples described in Table II and they

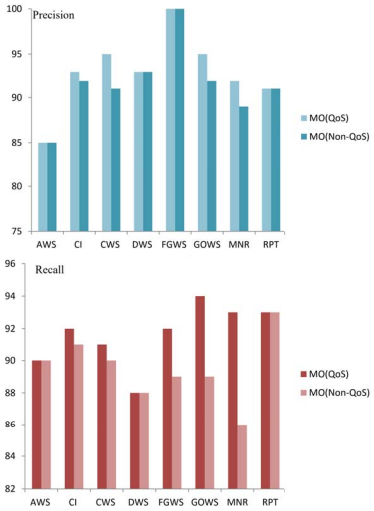

Figure 3: Comparative results of multi-objective approaches with and without QoS metrics

are not using the QoS attributes (just limited static interface and code level metrics).

C. Results and Discussions

Results for RQ1: The goal of RQ1 is to investigate the importance of QoS metric in detecting service antipattern. We executed both approaches (with and without QoS metrics) 5 times for the each fold of the validation (total of 50 runs). Figure 3 reports the comparative results. With same experiment settings, our muti-objective approach with QoS metrics performs slightly better than the one with only static metrics. The average precision is improved from 90.4% to 93.2%, while recall is improved from 87.4% to 91.6%. For antipattern types that have negative impacts on the service performance such as GOWS and MNR, the QoS metrics improved the accuracy service. However, for few antipattern types such as AWS, DWS, and RPT, the precision and recall remain the same because these antipatterns are not affected by the QoS metrics variation.

By using QoS metrics related to real-time performance like latency and availability, we manage to have promising detection results in antipatterns like FGWS, MNR and GOWS, especially for FGWS we reached a 100% precision for all the tests. The reason is that these antipatterns have substantial impacts to service performance and signatures on the structural documentation as well. While checking lower scores for few types like AWS which is more related to human understandability issue, we managed to identify most of them by using metrics like AMTO, AMTM, and AMTP.

Results for RQ2: The results of RQ2 are presented in Figure 4. Over 50 runs for 8 types of antipatterns, the random search (RS) did not perform well, with an average precision of 26.2%, and average recall of 27.3%. The main

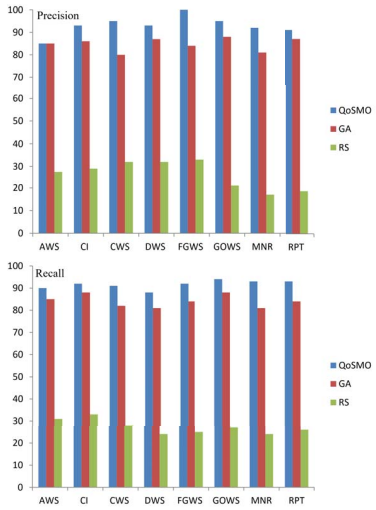

Figure 4: Comparative results of QoSMO, GA and RS

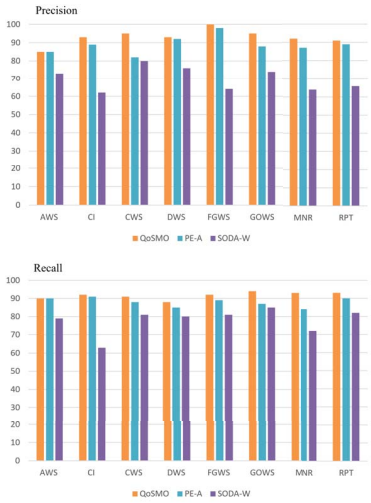

Figure 5: Comparative results of QoSMO, PE-A and SODA-W

reason is the very large search space of this problem due to the high number of possible combinations of metrics and thresholds. Regarding the mono-objective approach, the average precision and recall for GA were 84.8% and 84.1%. The experiments also show that there is no obvious bias to any specific antipattern type. However, our QoSMO approach outperforms GA significantly in average, while QoSMO can generate at least one better solution than the solution of GA in each single test. The reason is the fact that GA is only able to handle this problem as mono-objective and generates only one solution which is limited by the single fitness function. The results also confirm that the two objectives using in our formulation are confelcting. As a conclusion to this research question, our approach outperforms GA and RS based on the different correctness metrics.

Results for RQ3: Figure 5 reports the comparison study of our approach (QoSMO), PE-A [4], SODA-W [3]. PE-A performs well with an average precision of 88.8% and a recall of 90.0%. However, it is still less than QoSMO in detecting 7 types of antipatterns and performs same as QoSMO in detecting AWS. PE-A is using a cooperative parallel model to combine GA and GP, it is limited by using non-QoS metrics and mono-objective search algorithms. Several antipatterns are easier to detect using QoS metric as we describe in RQ1, and mono-objective approach may produce a solution that has a certain bias in optimizing an aggregated fitness function. SODA-W also has good results with an average precision of 69.8% and recall of 77.9%. The limitations of this work includes the ones mentioned earlier for PE-A, and also the lack of consideration of the source code of the service artifacts. These two levels of metrics reflect the design of different layers, and both layers are necessary to evaluate the static assessment of service design. Different from PE-A and SODA-W, our approach using not only static metrics, but also QoS metrics to evaluate the real-time performance of the services and multi-objective optimization to better manage the challenge of the conflicting objectives.

V. Threats to Validity

External threats: External threats may exist because in this work, we did not evaluate our approach on all possible antipattern types. However, the eight types of Web service antipatterns we employed constitute a broad representative set of standard and frequent defects. In addition, we did not yet generalize our approach for other service types such as RESTful services. It is also possible to extend the work for other domains such as mobile apps to validate the generality of our approach.

Construct threats: This type of threats are caused by the relationship between theory and what is observed. A possible threat is related to the antipattern examples that are being used to train/validate the approach, as the users may not agree with classified antipatterns. As we mentioned early, there is no general consensus on how a specific design violates the quality principles. This is indeed one key motivation to use multi-objective search-based approach to generate a set of solutions for users to choose from based on their preferences. In our experiments, the standard metrics such as precision and recall are used to validate the proposed approach, these metrics are widely used in validating code smell detection tools. As part of our future work, we may need to conduct a survey with developers to study the relevance of detected antipatterns.

VI. Related Work

Few studies have proposed to address the problem of SOA antipatterns. [7] was the first book related to this topic in the literature, it provides informal definitions of a list of Web service antipatterns. Later, Rotem-Gal-Oz described the symptoms of a set of SOA antipatterns in [11]. Then, [1] describes seven popular antipatterns which violate the SOA principles. [12] provides a set of guidelines for service providers to avoid bad practices while writing WSDL documentations and able to locate eight bad practices in writing WSDL for Web services. Beside the definition and guidelines of service antipattern, there are also several studies related to the detection part. Moha et al. proposed SODA, a rule-based approach for SCA systems (Service Component Architecture) in [2].

In another work [13], the authors created and reviewed a repository of 45 general antipatterns in SOA, and aim to help developers understand and avoid potential problems. [14] has proposed an approach to prevent antipattern during the phase of WSDL documentation generation. Recently, several search-based approaches have been proposed. [15] have proposed a bi-level search-based approach to finding the best detection rule by generating artificial defects to make up the size limitation of input antipattern examples. The limitation of the state-of-the-art approaches is the lack of using dynamic(QoS) measures to evaluate service quality.

VII. Conclusions and Future Work

In this paper, we introduce a search-based multi-objective approach to generated detection rules solutions for Web service antipatterns. These rules are a composition of QoS, Interface, and code level metrics. In our multi-objective adaptation, two fitness functions are used to maximize the coverage of antipattern examples and minimize the coverage of well-designed Web service examples. The proposed approach was evaluated on 500 Web services of a QoS benchmark and eight common Web service antipattern types. The empirical study shows that proposed QoS-aware antipattern detection outperforms our previous multi-objective approach and other state-of-the-art approaches.

As future work, we plan to extend the approach patterns when composing Web services. We also intend to apply the automate the process of service compositions/selection while avoiding antipatterns and improve QoS performance. Finally, we plan to improve the defect correction process through automated refactoring recommendations and prediction of service antipatterns.

References

[1] J. Král, M. Žemlička, J. Kral, M. Zemlicka, J. Král, and M. Zemlicka, "Popular SOA Antipatterns," in *Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns, 2009*. IEEE, 2009, pp. 271–276.

[2] N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, Y.-G. Guéhéneuc, B. Baudry, and J.-M. Jézéquel, "Specification and Detection of SOA Antipatterns." Springer, Berlin, Heidelberg, 2012, pp. 1–16.

[3] F. Palma, "Specification and Detection of SOA Antipatterns," in *2014 IEEE International Conference on Software Maintenance and Evolution*. Springer, 2014, pp. 670–670.

[4] A. Ouni, M. Kessentini, K. Inoue, and M. Ó Cinnéide, "Search-based web service antipatterns detection," *IEEE Transactions on Services Computing*, vol. PP, no. 99, pp. 1–1, 2015.

[5] H. Wang, A. Ouni, M. Kessentini, B. Maxim, and W. I. Grosky, "Identification of web service refactoring opportunities as a multi-objective problem," in *Proceedings - 2016 IEEE International Conference on Web Services, ICWS 2016*. IEEE, 2016, pp. 586–593.

[6] K. Deb, S. Pratab, S. Agarwal, and T. Meyarivan, "A Fast and Elitist Multiobjective Genetic Algorithm: NGSA-II," *IEEE Transactions on Evolutionary Computing*, vol. 6, no. 2, pp. 182–197, 2002.

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

[8] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, Q. Z. Sheng, J. Votano, M. Parham, and L. Hall, "Quality driven web services composition," *Proceedings of the twelfth international conference on World Wide Web WWW 03*, vol. 1, no. 11, p. 411, 2003.

[9] G. Fan, H. Yu, L. Chen, and D. Liu, "Petri net based techniques for constructing reliable service composition," in *Journal of Systems and Software*, vol. 86, no. 4, 2013, pp. 1089–1106.

[10] M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 45, no. 1, p. 11, 2001.

[11] A. Rotem-Gal-Oz, E. Bruno, and U. Dahan, *SOA Patterns*. Manning Publications, 2012.

[12] C. Mateos, M. Crasso, A. Zunino, and J. Coscia, "Avoiding WSDL bad practices in code-first web services," *SADIO Electronic Journal of Informatics*, 2012.

[13] 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.

[14] C. Mateos, J. M. Rodriguez, and A. Zunino, "A tool to improve codefirst Web services discoverability through text mining techniques," *Software: Practice and Experience*, vol. 45, no. 7, pp. 925–948, 2015.

[15] H. Wang, M. Kessentini, and A. Ouni, "Bi-level Identification of Web Service Defects," in *International Conference on Service-Oriented Computing*, vol. 9936 LNCS. Springer, 2016, pp. 352–368.