An Efficient Method for Automatic Antipatterns Detection of REST Web Services

Sobhan Mohammadnia, Rasool Esmaeilyfard* and Reza Akbari

Department of Computer Engineering and Information Technology, Shiraz University of Technology, Shiraz, Iran
E-mail: esmaeily@sutech.ac.ir
**Corresponding Author*

Received 21 April 2021; Accepted 17 July 2021;
Publication 21 September 2021

Abstract

REST Web Services is a lightweight, maintainable, and scalable service accelerating client application development. The antipatterns of these services are inadequate and counter-productive design solutions. They have caused many qualitative problems in the maintenance and evolution of REST web services. This paper proposes an automated approach toward antipattern detection of the REST web services using Genetic Programming (GP). Three sets of generic, REST-specific and code-level metrics are considered. Twelve types of antipatterns are examined. The results are compared with the manual rule-based approach. The statistical analysis indicates that the proposed method has an average precision and recall scores of 98% (95% CI, 92.8% to 100%) and 82% (95% CI, 79.3% to 84.7%) and effectively detects REST antipatterns.

Keywords: REST, web services, anti-patterns detection, service-oriented architecture (SOA), quality of service (QoS).

Journal of Web Engineering, Vol. 20_6, 1761–1780.
doi: 10.13052/jwe1540-9589.2063

1 Introduction

Service-Oriented Architecture (SOA) has been presented as a paradigmatic shift for designing flexible and distributed software solutions [1]. A service-oriented architecture is mainly implemented using web service technologies based on two well-known standards, so-called REST and SOAP. REST is a simpler and more efficient technology than the SOAP web services [2]. RESTful services have attracted more attention over the last decade.

However, the RESTful service design is influenced by the environment, context and designer's decisions. It may not consider qualitative principles. Koenig [3], for the first time, documented these common and repeated mistakes under the name of antipatterns.

For example, *breaking self-descriptiveness* is an instance of REST antipattern where developers tend to ignore the standardized headers, formats, or protocols and use their customized ones. This antipattern limits the reusability and adaptability of REST resources. *Ignoring MIME types* is another antipattern that developers tend to have a single representation of resources such as XML and JSON. This antipattern limits service accessibility and reusability in diverse languages. Some of the antipatterns are also related to both primary web service technologies, i.e., REST and SOAP. *Chatty service* is an instance of these antipatterns in which a large number of simple operators are required to complete an activity. The list of REST antipatterns is presented by Palma, et al. [4].

Automatic detection of these antipatterns can help service designers to make the best choice. However, recent studies on this issue are dependent on implementation technologies, mainly SOAP [5]. They cannot be generalized to RESTful services [6]. In short, extracting metrics in SOAP web services is based on their standard interface (WSDL). SOAP is operations-centric, whereas REST is resource-centric and is based on web standards. RESTful web services are on top of JSON or XML over HTTP, whereas SOAP Web services are on top of SOAP over HTTP. Therefore, SOAP methods cannot be used to identify REST-specific anti-patterns. Despite the fruitful application of REST services, no automated approach has been proposed to detect REST-Specific antipattern.

This paper presents an automated approach to identify REST antipatterns. The proposed approach is based on Genetic Programming (GP) that detects both REST-specific and generic SOA antipatterns. This approach is a search-based method that automatically generates detection rules based on real REST antipatterns. The approach specified generic, REST-specific metrics

and code-level metrics for this purpose. The main issue is to find the best metrics and their appropriate threshold values for detecting the antipatterns. Accordingly, antipattern detection is described as an optimization problem. This problem finds the best combinations of metrics and their appropriate threshold value among REST metrics and antipattern types. The experimental results show the effectiveness and accuracy of this approach.

The main contributions of this paper are:

- Automatic antipatterns detection of RESTful services. To the best of our knowledge, the proposed approach is one of the first attempts to detect such antipatterns in RESTful services.
- The structural characteristics of REST API are described based on Swagger documents to derive the anti-patterns metrics. For this purpose, a dataset has been collected from REST web services and Swagger documents.
- Both SOA and REST-specific anti-patterns are considered. Previous works used a rule-based approach that was manual and only considered REST-specific anti-patterns.

The remainder of the paper is organized as follows. The related work is presented in Section 2. In the third section, the genetic programming approach is proposed to detect REST antipatterns. The evaluation and experimental results are presented in Section 4, and finally, the conclusion is presented in Section 5.

2 Related Work

The initial studies on web services antipatterns focused on providing practical guidelines and best practices for web services—for example, Král and Zemlicka [7, 8] and Tripathi, et al. [9] introduced some service antipatterns and their symptoms. Mateos, et al. [10] considered refactoring as a practical approach to design better API. Rodriguez, et al. [11] and [12] presented a set of guidelines and best practices for developing the service-oriented application. Torkamani and Bagheri [13] proposed a checklist of forty-five antipatterns to identify web services' antipatterns. Zheng and Krause [14] introduced a new web service interaction antipatterns. However, these studies do not consider systematic approaches toward antipattern detection.

The recent studies on antipattern detection in web services considered systematic or automatic solutions. These studies were mainly dependent on implementation technologies such as SOAP antipattern. Due to SOAP's

more extended history, this technology has received the most attention. For example, Palma, et al. [4] and [6] developed a rule-based approach called SODA-W. In this approach, service antipatterns' characteristics are determined using a Domain Specific Language (DSL). They described the properties of antipatterns using a set of WSDL interface metrics. Also, Ouni et al. presented a series of studies to detect SOAP antipatterns automatically. These studies include various optimization techniques such as multi-objective approaches as well as evolutionary algorithms. Ouni, et al. [15] presented a search-based approach using GA to detect web service antipattern. Then he extended his work in [16] and presented it as an automatic approach using the PEA algorithm to detect SOAP antipattern. In the following, they also considered antipattern detection as a bi-level optimization problem [17] and a hybrid approach to improve the design quality of Web service interfaces as a combination of both deterministic and heuristic-based approaches [18]. Wang, et al. [19] proposed a multi-objective approach using NSGA II to detect web service defects. In addition to interface/code-level metrics, they added QoS metrics in their method. Saluja and Batra [20] also proposed an approach using optimized algorithms.

On the other hand, the REST antipatterns studies were limited; most of them, such as Tilkov [21], Pautasso [22] and Fredrich [23] considered a high-level view of this technology antipatterns and provided the best practices for developers. Rodriguez, et al. [24] were detected antipatterns by analyzing REST APIs HTTP request compliance with theoretical web engineering principles. Recently, Palma, et al. [4] and [6] presented manual rules for REST antipattern detection in his approach SODA-R. Alshraiedeh and Katuk [25] proposed a technique and an algorithm for detecting anti-patterns in RESTful Web services. based on the URIs parsing process. But their focus has been on the URL in particular and many quality metrics have not been considered.

A new set of studies has addressed the issue of predicting antipatterns in the early stages of development or improving the antipattern dataset. For example, Abid, et al. [26] considered the hypothesis that the quality of the source code and interface design can be used as indicators to predict the quality of service attributes of SOAP services without deploying the services. Rebai, et al. [27] designed a bi-level multi-objective optimization approach to enable the generation of antipattern examples that can improve the efficiency of detection rules on SOAP services.

Table 1 highlights a comparison of antipattern types and the detection technique of these studies. These studies are either manual and time-consuming or due to SOAP-specific metrics. These implementations could

Table 1 Taxonomy of related work

Contributions	Types of Anti-patterns	Detection Techniques	Technology
Král and Zemlicka [7] Král and Žemlicka [8]	Design-related anti-patterns , SOA anti-patterns	n/a	SOAP
Tripathi, et al. [9]	Design-related anti-patterns, SOA anti-patterns	n/a	SOAP
Torkamani and Bagheri [13]	Design-related anti-patterns, SOA anti-patterns	n/a	SOAP
Zheng and Krause [14]	Interaction-related anti-patterns	n/a	SOAP
Rodriguez, et al. [28]	Service discoverability anti-patterns	Information retrieval	SOAP
Mateos, et al. [10]	WSDL writing anti-patterns	Information retrieval	SOAP
Rodriguez, et al. [12]	WSDL writing anti-patterns	Rule-based technique	SOAP
Moha, et al. [29]	Service design and documentation	Rule cards, MDE	SOAP
Ouni, et al. [16]	Service design	Rule-based, evolutionary, GA	SOAP
Wang, et al. [17]	Service design	Bi-level optimization, rule-based,	SOAP
Ouni, et al. [18]	Service design	deterministic and heuristic-based approaches	SOAP
Ouni, et al. [15]	Service design	GA, rule-based	SOAP
Palma, et al. [6]	Syntactic design of requests/responses	Heuristics-based	SOAP
Saluja and Batra [20]	Service design	Optimized Algorithms	SOAP
Rodriguez, et al. [24]	HTTP requests	Heuristics-based	REST
Palma, et al. [4]	Service design and documentation and syntactic design of requests/responses	Rule cards, heuristics-based	REST
Alshraiedeh and Katuk [25]	URL Anti-patterns	Design science research process	REST

(*Continued*)

Table 1 Continued

Contributions	Types of Anti-patterns	Detection Techniques	Technology
Abid, et al. [26]	Source code and interface design	Hypothesis testing	SOAP
Rebai, et al. [27]	Structural and performance metrics	Bi-level multi-objective optimization	SOAP

not be generalized to RESTful services [4]. Therefore, an automatic approach toward REST antipattern detection is considered using GP.

3 The Proposed Method

In this section, the proposed approach for REST antipatterns detection is described. Figure 1 shows the schematic diagram of this method. Briefly,

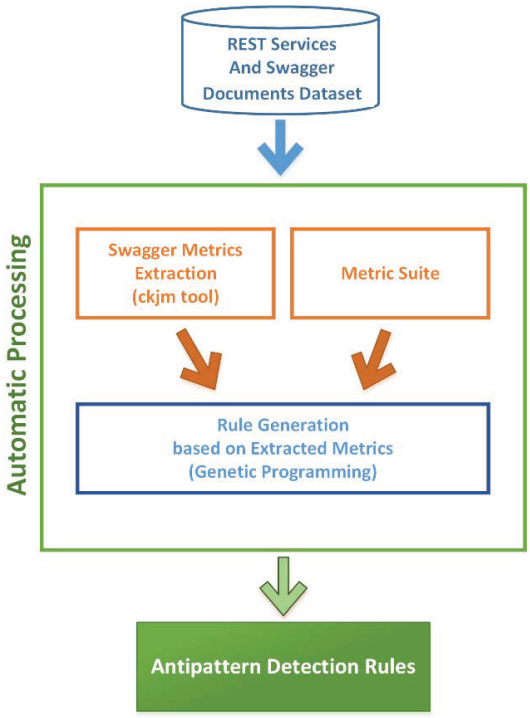


Figure 1 Schematic diagram of the proposed method.

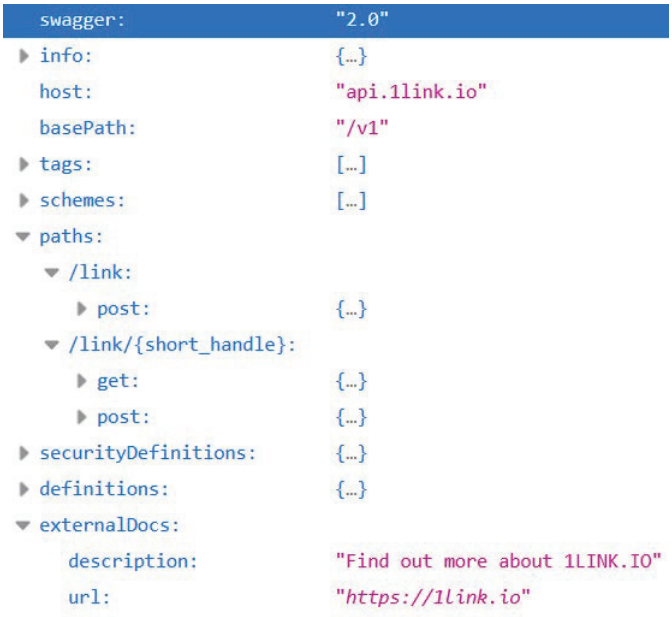


Figure 2 An instance of tiny service antipattern in swagger.

first, a dataset of REST services is gathered. Then swagger[1] is used to derive a set of REST metrics and their thresholds. Swagger is one of the RESTful API description languages for automated documentation, code generation, and test-case generation. The REST-related metrics was extracted from [16, 19, 30] and [4]. Then, random antipattern rules are generated for each type of antipattern using GP. These rules were used among the specified set of metrics and thresholds. As a result, the best rules find the highest number of antipatterns types over the dataset.

3.1 REST Services Dataset

A dataset of REST services is collected from various domains to prevent bias in results. The swagger API description was prepared for this dataset. The antipattern metrics are manually inspected over this dataset to determine the dataset of antipatterns. Figure 2 shows a REST service described in the swagger format and contains a *tiny service* antipattern. Tiny service is an antipattern that occurs in a service that comprises of few methods. Such

[1]swagger.io

services depend on other services to complete their functions and cannot work in isolation. This antipattern is the root cause of the failure of many service-oriented systems.

3.2 Swagger Metrics Extraction

In this work, three sets of metrics are considered for antipattern detection. The specification of the metrics is given below:

General interface-level metrics: These metrics exist in the web service interface (Table 2). In this study, the swagger interface is used to derive these metrics. These metrics (NMD-RPT) are derived from [4, 16].

REST-Specific metrics: These metrics are specific to the REST request and response headers (Table 3). Twelve metrics are used specifically for the REST architecture (AC-VRU). The first nine ones are defined in the literature [4]. The remaining three metrics are derived from the REST request and response header.

Code-level metrics: Code-level metrics are based on Chidamber and Kemerer Metrics [31]. These metrics are well-known object-oriented metrics (Table 4) used to evaluate service design quality [31]. To extract these metrics, the swagger interface is used to generate their java code structure using swagger-codejen. Then ckjm tool[2] is used to extract the metrics. As a result, code-level metrics are successfully extracted without access to the source

Table 2 General interface-level metrics

Metric	Description
NMD	Number of methods declared
NPH	Number of paths
NMPH	The average number of methods in paths
NPM	Number of parameters in methods
NCTP	Number of complex type parameters
COUP	Coupling
COH	Cohesion
NST	Number of primitive types
NOP	Number of parameters
RPT	The ratio of primitive types overall defined types

[2] www.spinellis.gr/sw/ckjm

Table 3 REST-Specific metrics

Metric	Description
AC	Authentication Cookie
AK	Action Keywords
CC	Client Cookie
CCV	Client Caching Value
ET	Entity Tag
HL	Header Link
HM	Http Method
LF	Location Field
RRF	Resource Representation Format
SC	Server Cookie
SCV	Server Caching Value
TLB	Total Links in response Body
VRB	Verbs in Request Body
VRU	Verbs in Request URI

code of these services, while their documentation was the only available resources. For all code-level metrics, the average value for all classes of the service was considered.

3.3 Adaptation of GP

In this section, the adaptation of the GP for antipattern detection is described. GP optimizes problems where the final solution may have a dynamic size. First, in the presented solution, the candidate solutions are modelled as a tree. Then, the anti-pattern detection is defined as an optimization problem using a fitness (objective) function.

Solution Representation: The key idea in adapting GP to the problem of antipattern detection is to provide candidate solutions and matching them to the dataset. Candidate solutions for this problem are rules that identify antipatterns. The solution to creating these anti-pattern detection rules is to consider the following structure:

if a logical combination of metrics and their permissible range is established, **then** an anti-pattern is identified.

To create this logical structure, a combination of criteria is used as a tree. This tree is shown in Figure 3. In this tree, the leaves represent the criteria

Table 4 Code-level metrics

Metric	Description
WMC	Weighted methods per class
	(NOM: Number of Methods in the QMOOD metric suite)
DIT	Depth of Inheritance Tree
NOC	Number of Children
CBO	Coupling between object classes
RFC	Response for a Class
LCOM	Lack of cohesion in methods
Ca	Afferent coupling (not a C&K metric)
Ce	Efferent coupling (not a C&K metric)
NPM	Number of Public Methods for a class
	(not a C&K metric; CIS: Class Interface Size in the QMOOD metric suite)
LCOM3	Lack of cohesion in methods Henderson-Sellers version
LCO	Lines of Code (not a C&K metric)
DAM	Data Access Metric (QMOOD metric suite)
MOA	The measure of Aggregation (QMOOD metric suite)
MFA	A measure of Functional Abstraction (QMOOD metric suite)
CAM	Cohesion Among Methods of Class (QMOOD metric suite)
IC	Inheritance Coupling (quality-oriented extension to C&K metric suite)
CBM	Coupling Between Methods (quality-oriented extension to C&K metric suite)
AMC	Average Method Complexity (quality-oriented extension to C&K metric suite)

(The metrics described in Tables 2–4) and logical rules (OR, AND) were used to combine these criteria in the internal nodes of the tree. Finally, this tree represented a candidate solution for use in the GP. Algorithm 1 describes a high-level overview of the detection algorithm. This algorithm follows the general structure of GP. However, in the first part, the described solution is used to create candidate rules (RandomlyGeneratedRules procedure) and then by extracting metrics from the dataset, the generated rules are matched with these candidate solutions (RuleMatchingWithDataset procedure)

Input and Output: GP takes the mentioned metrics and antipattern dataset as the input and provides a set of detection rules as an optimal solution. These rules are used to detect antipatterns.

Initialization: In the first steps of this algorithm (line 2), the initial population of detection rules is randomly constructed in a tree structure as described

earlier. Each node of the solution has two parts: the metric and the value. The value is assigned to the metric in a specified interval according to the metric type in the initiation.

Algorithm 1 Pseudo Code of Anti-Pattern Detection

Input: M: A set of REST-related quality metrics
D: A dataset of REST antipatterns
Output: R: Best rules for REST antipatterns detection

```
procedure Anti-patternDetection
R = RandomlyGeneratedRules(M)
while iteration < MaxOfIteration do
    for j = 1 to NumberOfCombinations do
        Select rule1, rule2 from R
        Co = CrossOver(rule1, rule2)
        RuleMatchingWithDataset(Co, D)
        Stack.Push(Co)
    end for
    for j = 1 to NumberOfMutation do
        Select rule from R
        Mu = Mutation(rule)
        RuleMatchingWithDataset(Mu, D)
        Stack.Push(Mu)
    end for
    R.Merge(Stack)
    R.Sort()
    R = R.Pop(NumberOfPopulation)
    iteration = iteration + 1
end while
end procedure
```

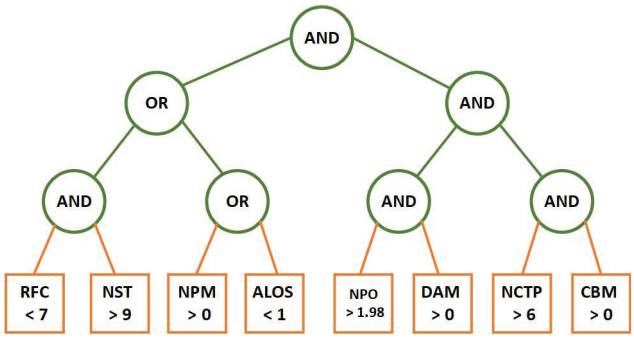


Figure 3 An instance of the antipattern rule tree.

Update: After initialization, the body of the algorithm starts. The main loops of GP, including crossover and mutation, are performed in lines 3–15. The crossover and mutation operators are used to generate new solutions. The crossover operator was random and single-point. In other words, two candidate solutions are selected, and a subtree is picked on each one. Then, the nodes and their subtrees are swapped from one parent to the other one. The mutation operator is applied to all nodes, including leaf nodes (metrics) and internal nodes (operators). If the selected node is a leaf node, it is replaced by another leaf node. The same is done for internal nodes, and a new internal node replaces the internal nodes.

Accordingly, the search space is explored, and new rules are generated by combining metrics. These rules evaluate the testing antipatterns iteratively using the following fitness function. This function aims to cover the maximum number of initial antipatterns samples:

$$\textit{Fitness} = \frac{1}{2}\left(\frac{\sum_{i=1}^{n} CD_i}{\textit{InitialAntipatterns}} + \frac{\sum_{i=1}^{n} CD_i}{\textit{DetectedAntipatterns}}\right) \epsilon[0,1] \quad (1)$$

where *InitialAntipatterns* is the number of antipatterns in the REST services dataset, *DetectedAntipatterns* is the number of detected antipatterns. If the *i-th* detected service exists in the dataset with the same antipattern type, then CD_i is 1 and 0 otherwise.

This fitness function (RuleMatchingWithDataset function) assesses the quality of each candidate's rules for detection. This assessment compares the detected antipatterns with the predetermined list of antipatterns in the dataset. This comparison takes into account the deviation from the generated rules. The best detection rules are saved (line 8, 14) and merged into the initial population (line 16). Subsequently, this population is sorted based on the cost of each member. Finally, a new population is selected for the next iteration (line 18).

Termination: The algorithm terminates after the maximum number of iterations and returns the best set of antipatterns detection rules. These rules can be used to detect potential antipatterns on any new REST web service.

4 Performance Evaluation

The empirical study to evaluate the performance of the proposed method is presented in this section. Precision and recall are used to assess the quality of the proposed method. These metrics are widely recognized as reasonable

proxies for the quality of antipattern detection solutions [4, 16]. The precision specifies the fraction of true antipatterns detected to the total number of detected antipatterns. At the same time, recall indicates the fraction of true antipatterns detected to the total number of existing antipatterns. To evaluate the automatic approach, an application using .NET/C#[3] was developed.

First, this section looks at how the automatic method is compared to manual methods. This section examined to what extent the proposed method can efficiently detect REST antipatterns. Hence, the precision and recall of antipatterns detection in REST services were considered. Second, this section looks for whether there is a bias towards the detection of any specific antipattern types. Therefore, the precision and recall of antipatterns detection are considered in different service categories and different types of antipatterns.

4.1 Dataset Preparation

A dataset of 35 REST services was collected from different web service search engines, including ServiceXplorer,[4] ProgrammableWeb.com, AnyApi.[5] Some of the services in the dataset and inspected antipatterns are summarized in Table 5.

4.2 Experimental Settings

The population size of GP was set to 100 with 1000 generations. The initial population was generated randomly. The crossover rate was set to 0.9, and

Table 5 A Sample of the dataset

Category	REST Services	Type of Anti-patterns
Travel	AviationData.Systems Airports API	Multi-Service
Sport	Baseball History API	Multi-Service
Sport	cbb-v3-stats	Multi-Service
Energy	Rebate Bus	Multi-Service
Mobile	1LINK.IO API	Tiny Service
API	APIs.guru	Tiny Service
Music	VocaDB	Multi-Service
Drugs	OtreebaOpenCannabis	Multi-Service
Sport	CFBv3Odds	Tiny Service

[3]https://visualstudio.microsoft.com/vs/features/net-development/

[4]eil.cs.txstate.edu/ServiceXplorer

[5]https://any-api.com/

the mutation probability was set to 0.4. The reported results of this algorithm were the median of 40 runs. These parameter values are specified empirically using the recommended methods in the literature [16].

4.3 Experimental Design and Results

The results are compared with the SODA-R approach [4] as a manual approach. Twelve antipatterns mentioned by SODA-R, including Chatty Service (CS), Multi-Service (MS), Tiny Service (TS), and Data Service (DS), Breaking Self-descriptiveness (BSD), Forgetting Hypermedia (FH), Ignoring Caching (IC), Ignoring MIME (IM), Ignoring Code Status (ICS), Tunneling Everything through POST (TEP), Tunneling Everything through GET (TEG), Misusing Cookies (MC) was considered.

The bootstrapping method was also used for both the SODA-R and the proposed approach results to estimate the average and standard deviation of their sampling distribution. Thousand bootstrap simulations were performed with a 95% confidence interval.

The results of each of the approaches are presented in Figure 4. Interestingly, this study discovered that the automatic approach detects antipatterns with an average precision and recall scores of 98% (95% CI, 92.8% to 100%) and 82% (95% CI, 79.3% to 84.7%). The obtained results were also compared using the Mann-Whitney test to detect significant differences. α was set at 0.05 to analyze the results statistically. The effect size was evaluated using Cohen's d statistic. The effect size is considered as follows: (1) small if $0.2 = d = 0.5$ (2) medium if between $0.5 = d < 0.8$ or (3) high if $d \geq 0.8$. The results reveal that the precision in GP differs significantly from

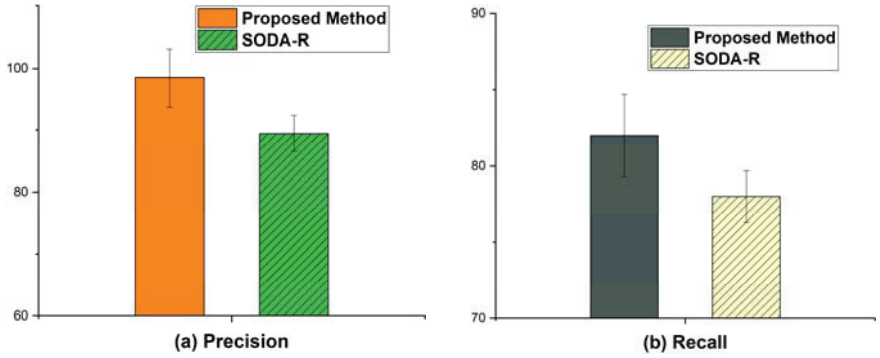


Figure 4 The precision and recall of manual vs automatic approach.

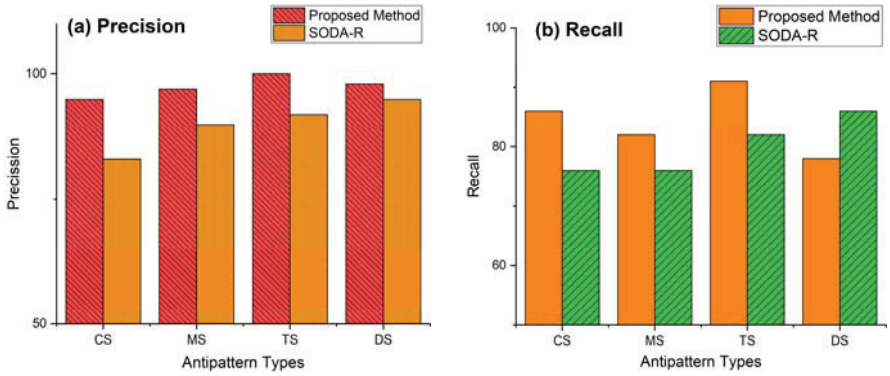


Figure 5 Detection results for common service antipattern types.

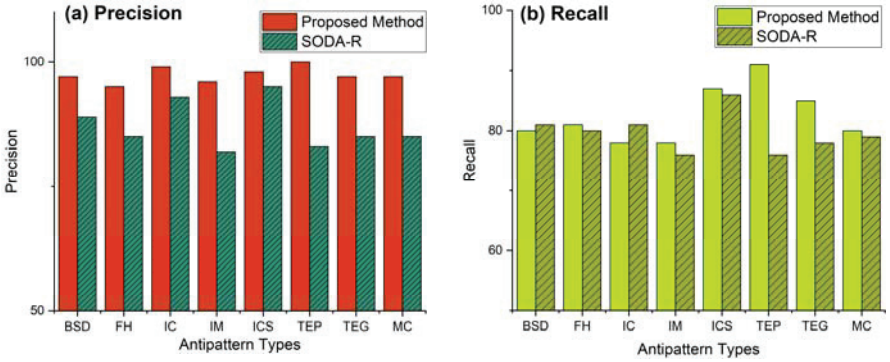


Figure 6 Detection results for REST-Specific antipattern types.

the SODA-R ($p < 0.05$ and $d = 0.5$ – Medium). The recall also differs significantly from SODA-R ($p < 0.05$ and $d = 0.55$ – Medium).

The detection results for each antipattern type for common service antipatterns and REST-specific antipattern are depicted in Figures 5 and 6. The most promising results are shown in *Tiny Service* (TS) and *Tunneling Everything through POST (TEP)*. These results indicate that the approach does not have a bias toward a specific type.

The scalability of the proposed method has been also evaluated by varying the population size in the GP from 50 to 550 and the items that may appear in the dataset from 5 to 35. Figure 7 reports the average execution time to perform the anti-pattern detection algorithm. The graphs show these execution times are fitted to a cubic function.

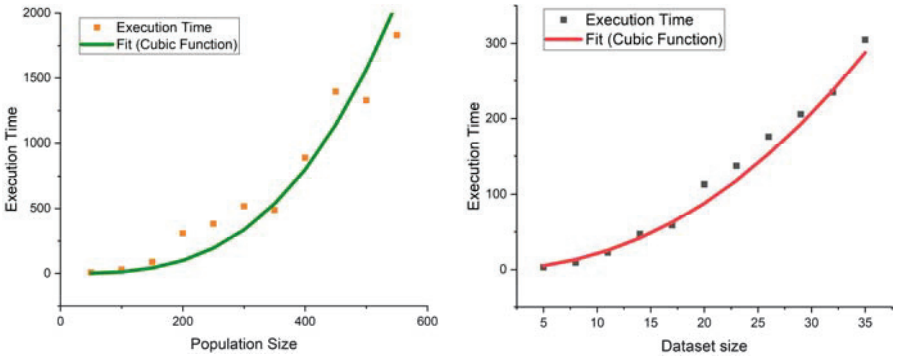


Figure 7 Detection results for REST-Specific antipattern types.

5 Conclusions

The existing manual approach for REST antipattern detection is based on the definition of detection rules. These rules largely depend on the designer's knowledge and expertise. In other words, if the designer does not have enough expertise, the defined rules may not be comprehensive and complete.

In this paper, an automated approach is introduced for antipatterns detection of RESTful services. In the proposed approach, the detection rules were generated in an optimization process and automatically combined with the metric/value threshold. Statistical analysis of the results reveals that the proposed approach has a promising average precision and recall scores of 98% (95% CI, 92.8% to 100%) and 82% (95% CI, 79.3% to 84.7%). The results indicated that the proposed approach performs better than the manual approach and there is a significant difference compared to this method.

References

[1] N. Niknejad, W. Ismail, I. Ghani, B. Nazari, M. Bahari, and A. R. B. C. Hussin, 'Understanding Service-Oriented Architecture (SOA): A systematic literature review and directions for further investigation," *Information Systems*, vol. 91, p. 101491, 2020/07/01/2020, doi: https://doi.org/10.1016/j.is.2020.101491.

[2] A. Huf and F. Siqueira, "Composition of heterogeneous web services: A systematic review," *J. Netw. Comput. Appl.,* Review vol. 143, pp. 89–110, Oct 2019, doi: 10.1016/j.jnca.2019.06.008.

[3] A. Koenig, "Patterns and antipatterns," *The patterns handbook: techniques, strategies, and applications*, vol. 13, p. 383, 1998.

[4] F. Palma, N. Moha, Y. Gu, x00E, x00E, and neuc, "UniDoSA: The Unified Specification and Detection of Service Antipatterns," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018, doi: 10.1109/TSE.2018.2819180.

[5] J. M. Roriguez, C. Mateos, and A. Zunino, "Assisting Developers to Build High-quality Code-first Web Service APIS," *Journal of Web Engineering*, vol. 14, no. 3–4, pp. 251–285, Jul 2015.

[6] F. Palma, J. Dubois, N. Moha, and Y.-G. Guéhéneuc, "Detection of REST Patterns and Antipatterns: A Heuristics-Based Approach," in *Service-Oriented Computing*, Berlin, Heidelberg, X. Franch, A. K. Ghose, G. A. Lewis, and S. Bhiri, Eds., 2014// 2014: Springer Berlin Heidelberg, pp. 230–244.

[7] J. Král and M. Zemlicka, "Crucial Service-Oriented Antipatterns, vol. 2," *International Academy, Research and Industry Association, IARIA*, pp. 160–171, 2008.

[8] J. Král and M. Žemlicka, "Popular SOA Antipatterns," in *2009 Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, 15–20 Nov. 2009 2009, pp. 271–276, doi: 10.1109/ComputationWorld.2009.80.

[9] D. Tripathi, U. Suman, M. Ingle, and S. Tanwani, "Towards Introducing and Implementation of SOA Design Antipatterns," *International Journal of Computer Theory and Engineering*, vol. 6, no. 1, p. 20, 2014.

[10] C. Mateos, M. Crasso, A. Zunino, and J. Coscia, "Detecting WSDL bad practices in code-first Web Services," *IJWGS*, vol. 7, pp. 357–387, 01/01 2011, doi: 10.1504/IJWGS.2011.044710.

[11] J. M. Rodriguez, M. Crasso, A. Zunino, and M. Campo, "Automatically Detecting Opportunities for Web Service Descriptions Improvement," ed, 2010, pp. 139–150.

[12] 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. 6, no. 43, pp. 613–639, 2013.

[13] M. A. Torkamani and H. Bagheri, "A Systematic Method for Identification of Anti-patterns in Service Oriented System Development," *International Journal of Electrical & Computer Engineering (2088-8708)*, vol. 4, no. 1, 2014.

[14] Y. Zheng and P. Krause, "Asynchronous Semantics and Anti-patterns for Interacting Web Services," in *2006 Sixth International Conference on Quality Software (QSIC'06)*, 27–28 Oct. 2006 2006, pp. 74–84, doi: 10.1109/QSIC.2006.14.

[15] A. Ouni, R. G. Kula, M. Kessentini, and K. Inoue, "Web Service Antipatterns Detection Using Genetic Programming," presented at the Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, Madrid, Spain, 2015.

[16] A. Ouni, M. Kessentini, K. Inoue, and M. Ó. Cinnéide, "Search-Based Web Service Antipatterns Detection," *IEEE Transactions on Services Computing*, vol. 10, no. 4, pp. 603–617, 2017, doi: 10.1109/TSC.2015.2502595.

[17] H. Wang, M. Kessentini, and A. Ouni, "Bi-level Identification of Web Service Defects," in *Service-Oriented Computing*, Cham, Q. Z. Sheng, E. Stroulia, S. Tata, and S. Bhiri, Eds., 2016// 2016: Springer International Publishing, pp. 352–368.

[18] A. Ouni, H. Z. Wang, M. Kessentini, S. Bouktif, and K. Inoue, "A Hybrid Approach for Improving the Design Quality of Web Service Interfaces," (in English), *ACM Trans. Internet. Technol.*, Article vol. 19, no. 1, p. 24, Mar 2019, Art no. 4, doi: 10.1145/3226593.

[19] H. Wang, M. Kessentini, T. Hassouna, and A. Ouni, "On the Value of Quality of Service Attributes for Detecting Bad Design Practices," in *2017 IEEE International Conference on Web Services (ICWS)*, 25–30 June 2017 2017, pp. 341–348, doi: 10.1109/ICWS.2017.126.

[20] S. Saluja and U. Batra, "Optimized approach for antipattern detection in service computing architecture," *Journal of Information Optimization Sciences*, vol. 40, no. 5, pp. 1069–1080, 2019.

[21] S. Tilkov, "REST Anti-Patterns," *InfoQ Article (July 2008)*, 2008.

[22] C. Pautasso, "Some REST Design Patterns (and Anti-Patterns)," ed, 2009.

[23] T. Fredrich, "Restful service best practices," *[Online]. http://www.restapitutorial.com/media/RESTfulBestPractices-v1*, vol. 1, 2012.

[24] C. Rodriguez et al., *REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices*. 2016, pp. 21–39.

[25] F. S. Alshraiedeh and N. Katuk, "A URI parsing technique and algorithm for anti-pattern detection in RESTful Web services," *International Journal of Web Information Systems*, vol. 17, no. 1, pp. 1–17, 2021, doi: 10.1108/IJWIS-08-2020-0052.

[26] C. Abid, M. Kessentini, and H. Wang, "Early prediction of quality of service using interface-level metrics, code-level metrics, and antipatterns," *Information and Software Technology*, vol. 126, p. 106313, 2020/10/01/2020, doi: https://doi.org/10.1016/j.infsof.2020.106313.

[27] S. Rebai, M. Kessentini, H. Wang, and B. Maxim, "Web service design defects detection: A bi-level multi-objective approach," *Information and Software Technology*, vol. 121, p. 106255, 2020/05/01/2020, doi: https://doi.org/10.1016/j.infsof.2019.106255.

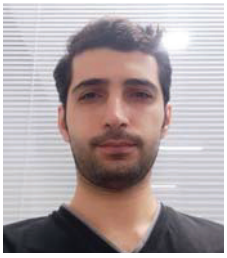
[28] J. M. Rodriguez, M. Crasso, A. Zunino, and M. Campo, "Improving Web Service descriptions for effective service discovery," *Science of Computer Programming*, vol. 75, no. 11, pp. 1001–1021, 2010/11/01/ 2010, doi: https://doi.org/10.1016/j.scico.2010.01.002.

[29] N. Moha et al., "Specification and Detection of SOA Antipatterns," in *Service-Oriented Computing*, Berlin, Heidelberg, C. Liu, H. Ludwig, F. Toumani, and Q. Yu, Eds., 2012//2012: Springer Berlin Heidelberg, pp. 1–16.

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

[31] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

Biographies



Sobhan Mohammadnia received the MSc degree from the Shiraz University of Technology, Iran, in 2019 under the supervision of Dr. Rasool Esmaeilyfard and Dr. Reza Akbari. He is currently a system analyst. His main research interests include analyzing service-oriented and process-oriented systems,

information systems, and business process management., in particular, he is interested in detecting service and process antipatterns in systems and evaluating the QoS of APIs.

Rasool Esmaeilyfard received his Ph.D. from the Isfahan University of Technology, Isfahan, Iran in 2017. Since 2018, he has been with the Faculty of the Department of Computer Engineering and Information Technology at the Shiraz University of Technology where he currently holds an assistant professor position. He is also a consultant, specializing in software architecture and distributed systems in the last ten years. His general research interests are in the areas of software architecture and crowd management.

Reza Akbari has a PhD in software engineering from Shiraz University. Currently, he is an associate professor at department of Computer Engineering and Information Technology of Shiraz University of Technology. His special fields of interest include software engineering, machine and deep learning, and optimization algorithms.