A Machine Learning-Based Approach to Detect Web Service Design Defects

Ali Ouni*, Marwa Daagi†, Marouane Kessentini‡, Salah Bouktif*, Mohamed Mohsen Gammoudi†
*Department of Computer Science and Software Engineering, CIT, UAE University, UAE
†ISAMM / Riadi Laboratory-ENSI, University of Mannouba, Mannouba, Tunisia
‡ Department of Information and Computer Science, University of Michigan, MI, USA
{ouniali, salahb}@uaeu.ac.ae, {daagi, gammoudi}@fst.rnu.tn, marouane@umich.edu

***Abstract*—Design defects are symptoms of poor design and implementation solutions adopted by developers during the development of their software systems. While the research community devoted a lot of effort to studying and devising approaches for detecting the traditional design defects in object-oriented (OO) applications, little knowledge and support is available for an emerging category of Web service interface design defects. Indeed, it has been shown that service designers and developers tend to pay little attention to their service interfaces design. Such design defects can be subjectively interpreted and hence detected in different ways. In this paper, we propose a novel approach, named *WS3D*, using machine learning techniques that combines Support Vector Machine (SVM) and Simulated Annealing (SA) to learn from real world examples of service design defects. *WS3D* has been empirically evaluated on a benchmark of Web services from 14 different application domains. We compared *WS3D* with the state-of-the-art approaches which rely on traditional declarative techniques to detect service design defects by combining metrics and threshold values. Results show that *WS3D* outperforms the the compared approaches in terms of accuracy with a precision and recall scores of 91% and 94%, respectively.**

***Keywords*-Web service design; design defects; Service interface**

I. Introduction

Service-oriented architecture (SOA) has become prevalent in architectures that support Web services, facilitating loosely coupled, distributed services that are vended as Web Service interfaces [1]. The Web service interface plays an extremely important role in enabling third-party customers to understand, discover, and reuse services [1], [2], [3].

Like any software system, a Web service is subject to several changes to enhance it with new features, adapt it to new contexts or to fix bugs and performance issues. Hence, service developers need to manage the growing complexity of changes as early as possible. Poorly planned changes and bad design choices may lead to the introduction of so-called design defects [4], i.e., "*not-quite-right*" design that developers provide to meet a deadline or to deliver the software to the market in the shortest time possible. Design defects are symptoms of poor design and/or implementation choices applied by programmers during the development of a software system [4]

Design defects are widely recognized by researchers and practitioners as a harmful source of technical dept leading to several understandability, usability, maintenance and evolution problems [5], [6], [7], [8], [9], which result in an increased bug rate, fragile design and inflexible code, lower productivity, and higher rework for developers [9], [10], [11]. For these reasons, researchers have been particularly active in the definition of techniques for detecting design defects in traditional OO applications [12], [13], [14], [15], [16], [17], as well as in the understanding of the effects of such design defects on non-functional attributes of source code [10], [11], [18]. However, little effort was devoted for design defects detection in service-based systems (SBSs). In this context, a set of new peculiar bad design/programming practices of Web service development has been recently identified.

The vast majority of existing work on Web service design defects detection attempts basically to provide definitions and/or the key symptoms that characterize common design defects [2], [6], [19], [20]. Recent works [21] rely on a declarative rule-based language to specify design defect symptoms at a higher-level using combinations of quantitative (metrics), structural, and/or lexical information. However, in an exhaustive scenario, the number of possible design defects to be characterized manually and formulated with rules can be very large. In addition, it is difficult to find a consensus to characterize and formulate each symptom. Recently, Ouni et al. [8], [22] proposed a search-based approach using genetic algorithms to generate detection rules for Web service design defects in the form of combinations of several metrics and their threshold values. However, the generated rules are complex and generic and require an extensive knowledge and high calibration effort to validate and adapt them. For these reasons, the detection task is still mainly a manual, time-consuming and subjective process.

In this paper, we introduce a new approach, named *WS3D* (Web Service Design Defects Detector), using Support Vector Machine (SVM) [23] to learn from real-world instances of service design defects. *WS3D* uses over 40 Web service metrics including interface, performance, and code quality metrics to characterize design defect symptoms. Given a set of training examples of design defect instances and their categories, our SVM-based approach builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier. To

978-1-5386-0752-7/17 $31.00 © 2017 IEEE
DOI 10.1109/ICWS.2017.62

find the best parameters configuration of SVM, we used simuated annealing (SA) [24] as an optimization technique. The proposed approach does not require any rules calibration effort or extensive expertise on Web service design defects. We empirically evaluated the efficiency of *WS3D* on a benchmark of Web services from 14 different application domains. We compared *WS3D* with two recent state-of-the-art approaches which rely on traditional declarative techniques to detect service design defects by combining metrics and threshold values including Palma et al. [21], and Ouni et al. [8]. Results show that *WS3D* outperforms the existing approaches in terms of accuracy with a precision and recall scores of 91% and 94%, respectively.

The remainder of this paper is organized as follows. Section II is provides the background material related to Web service design defects. Section III describes our SVM based approach for service design defects detection. Section IV presents and discusses the obtained experimental results while Section V discusses threats to validity. Section VI surveys related work and finally we conclude and outline our future research directions in Section VII.

II. Background and challenges

This section describes the basic concepts used in this paper, and the different challenges to detect design defects.

A. Web service design defects

Design defects 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 [18], and may be introduced unintentionally during initial design or during software development due to bad design choices, lack of developers experience, poorly planned changes or time pressure.

Different types of design defects presenting a variety of symptoms have been recently studied with the intent of improving their detection and suggesting improvements paths [6], [19], [21]. Common Web service design defects 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 [19].
- *Fine grained Web service (FGWS)*: is a too fine-grained service whose overhead (communications, maintenance, and so on) outweighs its utility. This defect refers to a small Web service with few operations implementing only a part of an abstraction. It often requires several coupled Web services to complete an abstraction, resulting in higher development complexity, reduced usability [19].
- *Chatty Web service (CWS)*: is a service where a high number of operations, typically attribute-level setters or getters, are required to complete one abstraction. This antipattern may have many fine-grained operations, which degrades the overall performance with higher response time [19].
- *Data Web service (DWS)*: contains typically accessor operations, i.e., getters and setters. In a distributed environment, some Web services may only perform some simple information retrieval or data access operations. A data Web service usually deals with very small messages of primitive types and may have high data cohesion [21].
- *Ambiguous Web service (AWS)*: is an design defect 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 [20], [25].
- *Redundant PortTypes (RPT)*: is a service where multiple portTypes are duplicated with the similar set of operations. Very often, such portTypes deal with the same messages. Redundant PortType antipattern may negatively impact the ranking of the Web Services [26].
- *CRUDy Interface (CI)*: is a service with RPC-like behavior by declaring create, read, update, and delete (CRUD) operations, e.g., `createX()`, `readY()`, etc. Interfaces designed in that way might be chatty because multiple operations need to be invoked to achieve one goal. In general, CRUD operations should not be exposed via interfaces [19].
- *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 [19].

We focus mainly in this paper on these eight design defect types as they are the most common ones in SBSs based on recent studies [3], [5], [6], [8], [21], [22], [27].

B. Support Vector Machine

Support vector machines (SVM) is a widely used machine learning technique that is based on the statistical theory of supervised learning introduced by Vapnik [23]. SVM solves mathematical optimization problems to find the separating hyperplane that has maximum margin between two classes. By maximizing the margin, the capacity or complexity of a function class (separating hyperplanes) is minimized.

SVM relies on the existence of a linear classifier in an appropriate space and uses a set of training data to train the parameters of the classifier. Let $\{(x_1, y_1), ..., (x_l, y_l)\}$ be a

set of training examples, where x_i is a data vector in the space R^n, and $y_i \in \{0, 1\}$ is its label. The SVM builds a function f to assign for a given x_i, the label y_i. The function f uses a hyperplane to assign the labels by separating the hyperspace in two. SVM attempts to find the hyperplane for which the minimum distance to the training examples is maximal. The margin is the distance between the hyperplane and the closest example. These margins are called support vectors. The optimal separating hyperplane is the one that maximizes the margin.

A hyperplane is defined as a set of points x satisfying $f(x) = w \cdot x + b = 0$ where $\cdot$ is the dot product space H, and $w \in H$ is a weight vector, $b \in R$ is a threshold. The weight vector w determines a direction perpendicular to the hyperplane, while b determines the distance to the hyperplane from the origin. However, even when using powerful kernels, sometimes data might not be fully separable, and some tolerance has to be considered when calculating the separating hyperplane. For that, Cortes and Vapnik [23] suggested a soft margin improvement to the original SVM algorithm by introducing slack variables that measure the degree of misclassification for each training instance. A new example z is classified according to which side of the hyperplane it belongs. The hyperplane is uniquely defined by the *support vectors.*. Thus, the training of an SVM consists of the following minimization problem: $min_{w,\xi}\{\frac{1}{2}\|w\|^2 + C\sum_i \xi_i\}$, where w represents the hyperplane normal vector and C is the weight of the penalty function which is formulated by the sum of all ξ slack variables [23].

C. Challenges for Web service design defects detection

Several issues and challenges are related to the detection of an emerging category of Web service design defects. The main challenge is that there is no general consensus on how to decide if a particular design violates a quality heuristic. In fact, there is a difference between detecting symptoms and asserting that the detected situation is an actual design defect. The few existing works [8], [21] are trying to translate these symptoms into detection rules in the form of a combination of metrics and their threshold values. However, one of the fundamental issues is how to identify the most suitable list of metrics from an exhaustive list of metrics that best characterize the actual symptoms. Another issue is related to the definition of thresholds when dealing with quantitative information.

For example, the GOWS defect detection involves information such as the interface size (number of operations). Although we can measure the size of an interface, an appropriate threshold value is not trivial to define. An interface that is considered large in a given service/community of users could be considered average in another. The generation of detection rules requires a extensive knowledge about such defects to cover most of the possible bad-practice behaviors. To make the situation worse, defects are not usually documented by developers (unlike bugs report and object oriented design). Thus, it is time-consuming and difficult to manually inspect and validate detection rules that cover all possible symptoms.

Another important challenge, is that such detection rules are difficult to generalize in different contexts. Moreover, such detection rules are only generated from bad examples (instance of design defects) and do not consider good design practices which would result into high rate of false positives. We thus believe that a machine learning learning approach that is not based on detection rules would one of the best ways to detect design defects.

III. Web Service Design Defects Detection Using Support Vector Machine

The general structure of our approach is described in Figure 1. Our approach, *WS3D*, takes as input a training dataset of Web service interfaces that contain occurrences of design defects from different types. Then, for each Web service, our approach calculates an exhaustive list of quality metrics to best characterize the existant defect symptoms. These quality metrics are then used by our detection model based on machine learning algorithm using Support Vector Machine (SVM) model. Due to the nature of the detected design defects having similar symptoms and ambiguous characteristics, finding a separation between the SVM classes is complex and strongly depends on the available feature set and the tuning of hyper-parameters. To this end, we use an optimization technique based on Simulated Annealing (SA) [24] to tune the parameters of our SVM model and its optimal configuration. As output, our approach identifies if a given Web service is affected by a specific design defect or not. In the following we describe the SVM adaptation to our Web services design defects detection problem.

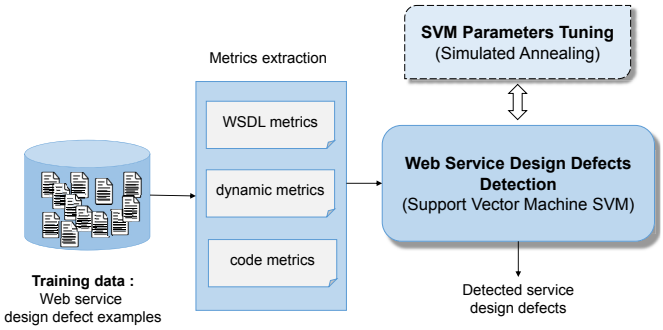

Figure 1: The proposed approach for Web service design defects detection.

A. Training Dataset Collection

Our SVM-based approach uses a dataset of real world instances of Web services. The used Web services are from various application domains, e.g., travel, weather, finance,

shipping, etc. They could be collected from different existing Web service search engines such as ServiceXplorer[1], programmableweb.com[2], etc. These Web services were manually inspected and validated based on existing guidelines from the literature [19], [27]. The training dataset contains instances of Web service design defects as well as instances of well designed Web services that are not affected by design defects. As described in Section II, we consider eight common design defects: *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).

Our training dataset, T, is split per design defect type. For each defect type t, T_t is denoted by (x_i, y_i), $1 \leqslant i \leqslant N$, where N represents the number of training data. Each datum must conform to the criteria $x_i \in R^d$ and $y_i \in \{-1, 1\}$, where d denotes the number of quality metrics used in each service of the input data.

B. Metrics selection and extraction

Our approach uses a list of existing Web service metrics [8], [21], [22], [28], [29], [30]. As described in Table I, our approach uses a large metric suite of over 42 quality metrics including (*i*) WSDL interface metrics, (*ii*) code level metrics, and (*iii*) performance metrics. WSDL metrics aim at capturing the structural properties of Web services in the interface (WSDL) level. Code metrics are extracted from the service Java artefacts extracted using the Java™ API for XML Web Services (JAX-WS)[3] [7], [31] as well as the *ckjm* tool[4] (Chidamber & Kemerer Java Metrics) [32]. Moreover, we used dynamic metrics based on Web service invocations, e.g., response time, availability.

C. Support Vector Machine Modeling

Our approach uses SVM which provides a well-established and powerful classification method to analyse data and find the minimal-risk separation between different classes. The adopted SVM model uses a polynomial kernel. For each design defect type, our dataset $T = \bigcup T_T$, where $t \in \{$*GOWS, FGWS, CWS, DWS, AWS, RPT, CI, MNR, NONE*$\}$. For example, $T_{GOWS} = \{(x_i, y_i) | x_i \in R^d, y_i \in \{-1, 1\}, 1 \leqslant i \leqslant N\}$ denotes the dataset of god object web services, where y_i indicates whether the Web service x_i is a GOWS occurence (1) or not (-1). Similarly, T_{NONE} denotes the data set of Web services that are well designed (i.e., defect free).

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

[2] programmableweb.com

[3] http://docs.oracle.com/javase/6/docs/technotes/tools/share/wsimport.html

[4] http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/

Table I: List of Web service quality metrics used.

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

D. SVM parameter tuning using SA

Due to the nature of the dataset of such service design defect instances having similar, noisy and sometimes incomplete symptoms, the learning processus using SVM should comply with the optimal parameters. Indeed, finding hyperplane separation strongly depends on the tuning of hyper-parameters. Techniques for SVM parameters optimization are known to improve classification accuracy, and its literature is extensive. We first used a trial and error strategy to find this suitable parameters of SVM including the penalty weight C, and the The polynomial degree d. Then, we adopted the popular search-based algorithm simulated annealing (SA) [24] to search for the best value after executing our approach more than 30 times. However, this threshold could adjusted by the users.

In order to optimize the SVM parameters using SA, the selected parameters are placed in a vector form which will evolve to find the best combination of parameters. The evolution is achieved by modifying a random value in the vector in each iteration of the algorithm, and evaluating the

Table II: Web services used in the empirical study.

Category	# services
Financial	94
Science	34
Search	37
Shipping	38
Travel	65
Weather	42
Media	19
Education	26
Messaging	29
Location	31
Social	64
Payment	53
Cloud	49
Tools	57
All	**609**

performance of the SVM using the parameters in the new vector. If the new vector achieves a better performance, then it becomes the current vector. Performance is based on the accuracy within the training dataset.

IV. Empirical Study

This section presents the evaluation of our approach. We first present our research questions, the experiments setup and then describe and discuss the obtained results.

A. Research questions

We designed our experiments to address the following research questions:

- **RQ1.** To what extent can the proposed approach efficiently detect Web service design defects compared to state-of-the-art Web service design defects detection approaches?
- **RQ2.** What types of Web service design defects does it detect correctly?

B. Analysis Method

To evaluate our approach, we conducted a set of experiments where we trained our SVM model from a real-world data set that consists of a set of Web services.

Training Dataset. We collected a random 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 II summarizes the collected services. Furthermore, so as to not bias our empirical study, our collected Web services are drawn from 14 different application domains: financial, science, search, shipping, travel, weather, media, education, messaging, location, social, payment, cloud and tools. All services were manually inspected and validated to identify design defect occurrences based on guidelines from the literature [27], [19]. Furthermore, our dataset is available online [33] to encourage future research in the area of automated detection of Web service design defects.

Design Defect Types. To evaluate our approach, we considered a set of eight different Web service design that are commonly found in SBSs and well documented in the literature [19], [21], [27]. In particular, we considered 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*). The basic definitions and symptoms of each of these design defects are summarized in Section II-A.

Evaluation Measures and Procedure. Our experiment involved Web services from fourteen different categories, i.e., application domains. To evaluate our approach, we perform a fourteen-fold cross validation procedure to measure the accuracy of our approach. We distributed our dataset into training data and evaluation data. Each fold consists of 13 parts, each part is a category from our data as shown in Table II, and the remaining part is the testing data. For instance, weather services are analyzed by training from design defect instances from travel, shipping, search, science financial, media, education, messaging, location, social, payment, cloud, and tools services.

To answer **RQ1**, we employed widely-used evaluation metrics: *precision* and *recall* [34] to evaluate the accuracy of our approach. Precision denotes the ratio of true design defects detected over the total number of detected defects, while recall indicates the ratio of true design defects detected over the total number of existing design defects. Specifically, precision and recall measures are defined as follows:

$$Precision = \frac{Detected\ Defects \bigcap Expected\ Defects}{Detected\ Defects} \in [0,1] \quad (1)$$

$$Recall = \frac{Detected\ Defects \bigcap Expected\ Defects}{Expected\ Defects} \in [0,1] \quad (2)$$

Precision and recall metrics were used also to compare our approach with two recent state-of-the-art approaches, namely *SODA-W* proposed by Palma et al. [21], and with *P-EA* proposed by Ouni al. [8]. SODA-W manually translates design defect symptoms into detection rules based on a literature review of Web service design. Ouni et al. [8] used a parallel evolutionary algorithm (P-EA), to generate Web service design defects detection rules in the form of combinations of quality metrics and threshold values. All three approaches are tested on the same benchmark described in Table II.

Statistical Test Method. Moreover, to compare the different approaches, we used appropriate statistical tests based on Wilcoxon signed rank test in a pairwise fashion in order to detect significant performance differences between the different approaches (*WS3D*, *SODA-W*, and *P-EA*) under comparison. The Wilcoxon test does not require that the data sets follow a normal distribution since it operates on values' ranks instead of operating on the values themselves. We set the confidence limit, α, at 0.01. Moreover, we investigate

Table III: Comparison results of *WS3D* against *P-EA* and *SODA-W* using Wilcoxon test and Cliffs delta (d).

		WS3D vs P-EA	WS3D vs SODA-W
Precision	*p-value*	<0.01	<0.01
	d	0.73 (Large)	1.14 (Large)
recall	*p-value*	<0.01	<0.01
	d	0.59 (Large)	1.27 (Large)

the effect size using Cliff's delta (d) . The effect size is considered:

$$\begin{cases} Negligible & if \mid d \mid < 0.147 \\ Small & if \ \ 0.147 \leq \mid d \mid < 0.33 \\ Medium & if \ \ 0.33 \leq \mid d \mid < 0.474 \\ Large & if \ \ \mid d \mid \geq 0.474 \end{cases} \quad (3)$$

To answer **RQ2**, we investigated the design defects types that were detected to find out whether there is a bias towards the detection of specific design defect types. We thus use the same precision and recall metrics defined above.

C. Results

Results for RQ1. Figures 2 and 3 report precision and recall results achieved by our *WS3D* approach, *P-EA* [8], and *SODA-W* [21]. From all the considered categories, our approach achieved promising detection results with an average precision of 91% and recall of 94%. However, both rule-based approaches *P-EA* and *SODA-W* acheived 82% and 67% of precision, respectively, and 84% and 78%, respectively. In addition, Table III reports the results of the Wicoxon test (adjusted p-values) and the Cliffs d effect size. The statistical tests provide evidence that *WS3D* outperforms both rule-based approach with a 'large' effect size.

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 design defects. Indeed, SODA-W is limited to a set of WSDL interface metrics, but ignores the source code of the Web service artifacts. That is a, service design may look promising at the interface level, but can prove to be an antipattern if the source code is not implemented well. Similarly to *SODA-W*, *P-EA* achieved less performance which indicate that machine learning technique are more appropriate for the problem of Web service design defects. In contrast, our approach does need to specify detection rules and to select metrics that best describe specific symptoms, then associate threshold which need a high calibration effort to find the right one.

Results for RQ2. Results for RQ2 are reported in Figures 4 and 5. We observe that *WS3D* does not have a bias towards the detection of specific antipattern types. We observe from the figures that we had, in all categories, a relatively equitable detection results in terms of both precision and recall for each design defect type. Moreover, based on a manual inspection of the results, we noticed that for some categories such as weather and search, the distribution of design defects detection is not as balanced. This is principally due to the number of actual design defect types in these categories

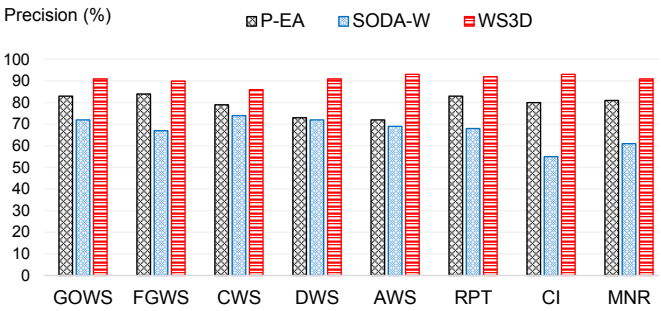

Figure 2: The achieved precision detection results for each Web service design defect type.

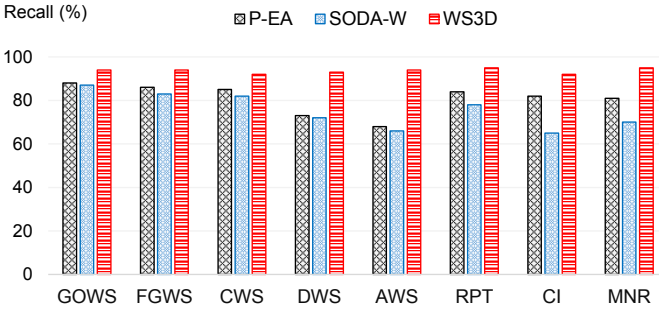

Figure 3: The achieved recall detection results for each Web service design defect type.

(none DWS instance exists in weather and search category). Consequently, any single false positive will lead to 0% of precision score.

Overall, all the eight defect types are detected with good precision and recall scores (91% and 94%, respectively). Most of detected design defects are true positives and *WS3D* did not miss any existing defect type. This ability to identify different types of design defect underlines a key strength to our approach. Most other existing approaches [8], [21] rely heavily on the notion of size to specify design defect. This is reasonable considering that some design defects like the GOWS are associated with a notion of size. For defects like data service and ambiguous service, however, the notion of size is less important and this makes this type of design defect hard to detect using structural information.

V. Threats to Validity

External threats to validity may arise because we did not evaluate the detection of all defect types. However, the eight types of Web service design defects we employed constitute a broad representative set of standard design defects. 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

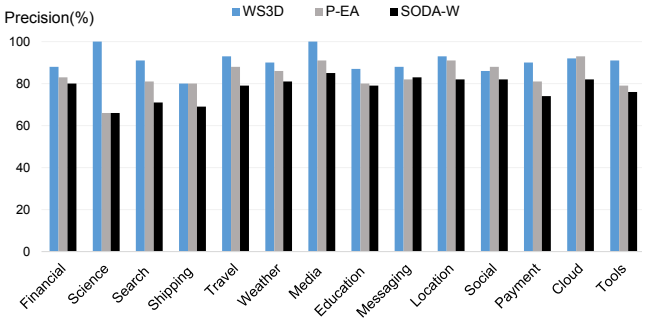

Figure 4: The achieved precision detection results for each Web service category.

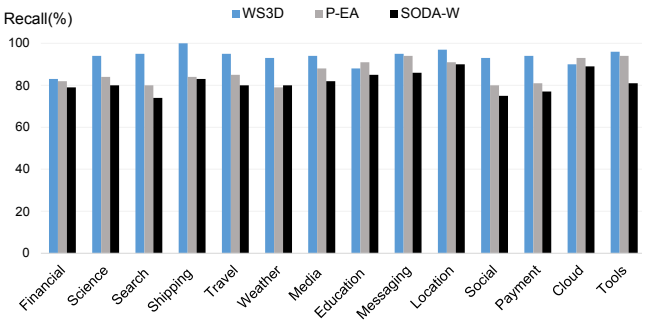

Figure 5: The achieved recall detection results for each Web service category.

good proxies for the quality of antipattern and code smell detection solutions [8], [21], [22]. Another threat is related to the corpus of design defect examples, as developers may not all agree if a candidate Web service is a design defect or not. To mitigate this threat, we followed detection guidances from the existing literature [19], [21], [25].

VI. Related work

Detecting and specifying design defects 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. [19] 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 [27]. Furthermore, Král et al. [6] listed seven "popular" SOA antipatterns that violate accepted SOA principles. In addition, a number of research works have addressed the detection of such antipatterns. Recently, Palma et al. [21] have proposed a rule-based approach called SODA-W that relies on declarative rule specification using a domain-specific language (DSL) to specify/identify the key symptoms that characterize an antipattern using a set of WSDL metrics. In another study, Rodriguez et al. [35], [20] and Mateos et al. [25] 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 [36], 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. [37] have proposed an interesting approach towards generating WSDL documents with less antipatterns using text mining techniques. Recently, Ouni et al. [22], [8] proposed a search-based approach based on evolutionary computation to find regularities, from examples of Web service antipatterns, to be translated into detection rules. However, detections rules based approaches tend to have a higher number of false positives.

Unlike service-oriented computing, there is an extensive body of research that tries to detect object oriented design defects and code smells [12], [13], [14]. Marinescu el al. [12] 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. Maiga et al. [38] proposed a similar approach our using SVM to detect code smells in some specific location in the code while considering the feedback of the developers. 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.

VII. Conclusion

In this paper, we introduced *WS3D*, a SVM-based approach for Web service design defects detection. Our approach uses simulated annealing to determine the optimal parameters for SVM. We evaluated our approach on eight common defect types, and over 600 Web services. Results show that *WS3D* achieves an average of 91% and 94% of precision and recall, respectively. The statistical analysis of the obtained results indicate that *WS3D* outperforms state-of-the-art approaches. As future work, we plan to validate our approach with additional design defect types. Another research direction worth to explore is to consider both bad and good Web service instances to deduce design defects detection rules, i.e., good detection rules should maximize the distance with well-designed Web services while minimizing the distance with badly-designed ones. Furthermore, in this paper, as we mainly focus on the detection of Web service design defects, we are planning to extend the approach by automating their correction too.

Acknowledgment. This Work is supported by the Research Start-up (2) 2016 Grant G00002211 funded by UAE University, and based in part upon support of the National Science Foundation under Grant Number 1661422.

References

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

[2] M. Crasso, J. M. Rodriguez, A. Zunino, and M. Campo, "Revising WSDL Documents: Why and How," *IEEE Internet Computing*, vol. 14, no. 5, pp. 48–56, 2010.

[3] 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.

[4] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

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

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

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

[8] A. Ouni, M. Kessentini, K. Inoue, and M. O Cinneide, "Search-based web service antipatterns detection," *IEEE Transactions on Services Computing*, vol. PP, no. 99, 2016.

[9] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[10] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2013.

[11] A. Yamashita and L. Moonen, "Exploring the impact of inter-smell relations on software maintainability: An empirical study," in *35th International Conference on Software Engineering*, 2013, pp. 682–691.

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

[13] 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.

[14] 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.

[15] W. Kessentini, M. Kessentini, H. Sahraoui, S. Bechikh, and A. Ouni, "A Cooperative Parallel Search-Based Software Engineering Approach for Code-Smells Detection," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 841–861, 2014.

[16] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1617–1664, 2014.

[17] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241–2260, oct 2012.

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

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

[20] 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.

[21] 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.

[22] A. Ouni, R. Gaikovina Kula, M. Kessentini, and K. Inoue, "Web service antipatterns detection using genetic programming," in *Genetic and Evolutionary Computation Conference (GECCO)*, 2015, pp. 1351–1358.

[23] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, no. 3, pp. 273–297, 1995.

[24] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi *et al.*, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983.

[25] 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.

[26] 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.

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

[28] M. Perepletchikov, C. Ryan, and Z. Tari, "The impact of service cohesion on the analyzability of service-oriented software," *IEEE Transactions on Services Computing*, vol. 3, no. 2, pp. 89–103, 2010.

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

[30] H. Wang, M. Kessentini, and A. Ouni, "Bi-level identification of web service defects," in *International Conference on Service-Oriented Computing*. Springer, 2016, pp. 352–368.

[31] 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.

[32] 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.

[33] "Experimental data," https://github.com/ouniali/WSantipatterns, accessed: 2017-02-01.

[34] W. B. Frakes and R. Baeza-Yates, Eds., *Information Retrieval: Data Structures and Algorithms*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992.

[35] 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.

[36] 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.

[37] 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.

[38] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y. Guéhéneuc, and E. Aïmeur, "SMURF: A svm-based incremental anti-pattern detection approach," in *Working Conference on Reverse Engineering, WCRE*, 2012, pp. 466–475.