Improving Web Services Design Quality Using Heuristic Search and Machine Learning

Marouane Kessentini, Hanzhang Wang and Josselin Troh Dea
Computer and Information Science Department
University of Michigan
MI, USA
firstname@umich.edu

Ali Ouni
Computer Science Department,
UAE University, UAE
ouniali@uaeu.ac.ae

***Abstract*—Web services evolve over time to fix bugs or update and add new features. However, the design of the Web service's interface may become more complex when aggregating many unrelated operations in terms of context and functionalities. A possible solution is to refactor the Web services interface into different modules that help the user quickly identifying relevant operations. The most challenging issue when refactoring a Web services interface is the high number of possible modularization solutions. The evaluation of these solutions is subjective and difficult to quantify. This paper introduces the use of a neural network-based evaluation function for the problem of Web services interface modularization. The users evaluate manually the suggested modularization solutions by a Genetic Algorithm (GA) for a number of iterations then an Artificial Neural Network (ANN) uses these training examples to evaluate the proposed Web services design changes for the remaining iterations. We evaluated the efficiency of our approach using a benchmark of 82 Web services from different domains and compared the performance of our technique with several existing Web services modularization studies in terms of generating well-designed Web services interface for users.**

Keywords-Web services design; quality; genetic algorithms; machine learning.

I. INTRODUCTION

One of the important factors for deploying successful and popular services is to provide a well-designed interface to the users that can help them to easily find relevant operations [6]. In fact, the Web services interface. Web services interface could be provided by different service providers such as FedEx, Google, PayPal and Google, and represents the only visible part for the users to select the operations that they want to adopt in their implementation of services-based systems. Thus, the design quality of Web services interface is a critical and an important problem.

The evolution of Web services may have a negative impact on the design quality of the interface by concatenating many non-cohesive operations that are semantically unrelated. The Web services interface design becomes unnecessarily complex for users to find relevant operations to be used in their services-based systems. An example of well-known interface design defect is the God object Web service (GOWS) [1][3]. GOWS implements many operations related to different business and technical abstractions in a single service interface leading to low cohesion of its operations and high unavailability to end users because it is overloaded. Indeed, the modularization process of how operations should be exposed through a service interface can have an impact on the performance, popularity and reusability of the service and it is not a trivial task.

Recently, several studies provided solutions to improve the design of Web service interfaces for the users/subscribers [11][6][1][5][2][8]. However, most of these studies addressed the problem of the detection of design defects of Web services interface based on declarative rule specification and not the correction step to fix these design defects. In these existing techniques, Web services modularization solutions are evaluated based on the use of quality metrics. However, the evaluation of the design quality is subjective and difficult to formalize using quality metrics with the appropriate threshold values due to several reasons.

Several challenges could be discussed around the modularization of Web services interface. First, there is no consensus about the definition of Web services design defects [3][20][9][17], also called antipatterns, due to the various user behaviors and contexts. Thus, it is difficult to formalize the definitions of these design violations in terms of quality metrics then use them to evaluate the quality of a Web service modularization solution. Second, existing studies do not include the user in the loop to analyze the suggested modularization solutions and give their feed-back during the design improvement process. Third, the computational complexity of some Web services quality metrics is expensive thus the defined fitness function to evaluate proposed Web services design changes can be expensive. Fourth, deciding on how to decompose/modularize an interface is subjective and difficult to automate since it is required to integrate the feedback of users during the modularization process Finally, quality metrics can just evaluate the structural improvements of the design after applying the suggested interface changes but it is difficult to evaluate the semantic coherence of the design without an interactive user interpretation.

We propose, in this paper, a Genetic Algorithm (GA)-based interactive learning algorithm [18] for Web services interface modularization based on Artificial Neural Networks (ANN) [12]. The proposed approach is based on

978-1-5386-0752-7/17 $31.00 © 2017 IEEE
DOI 10.1109/ICWS.2017.63

the important feedback of the user to guide the search for relevant Web services modularization solutions using predictive models. To the best of our knowledge, the use of predictive models has not been used to improve the quality of Web services design. In the proposed approach, we are modeling the user's design preferences using ANN as a predictive model to approximate the fitness function for the evaluation of the Web services modularization solutions. The user is asked to evaluate manually Web services interface modularization solutions suggested by a Genetic Algorithm (GA) for few iterations then these examples are used as a training set for the ANNs to evaluate the solutions of the GA in the next iterations.

We evaluated our approach on a set of 82 real-world Web services, extracted from an existing benchmark [1][5]. Statistical analysis of our experiments shows that our interactive approach performed significantly better than the state-of-the-art modularization techniques [5][2] in terms of design improvements and fixing design defects. The primary contributions of this paper can be summarized as follows:

1. The paper introduces a novel way to modularize and improve the design quality of Web services using interactive predictive modeling optimization. The proposed technique supports the adaptation of interface design solutions based on the user without the need to use specific design quality metrics. To the best of our knowledge, we propose the first approach to interactively generate a modularized Web services interface using predictive modeling techniques.
2. The paper reports the results of an empirical study on an implementation of our approach. The obtained results provide evidence to support the claim that our proposal is more efficient, on average, than existing Web services modularization techniques based on 82 real-world services.

The rest of this paper is outlined as follows. Section 2 is dedicated to the problem statement. Section 3 presents the proposed approach. Section 4 presents results of experimentations. Finally, section 5 summarizes our findings.

II. Background and Related Work

A. Background

The interface of a Web service is described as a WSDL (Web service Description Language) document that contains structured information about the offered operations and their input/output parameters [4]. A port Type is a set of abstract operations. Each operation refers to an input message and output messages. The users select the desired operation on their services-based system implementation via the interface by specifying the name of the operations and the required parameters (inputs) and they receive the required outputs without accessing to the source code of these used operations.

Most of existing real-world Web services interface regroup together a high number operations implementing different abstractions such as the Amazon EC2 that contains more than 100 operations in some releases. There are few WSDL design improvement tools [4][14] that have emerged to provide basic refactorings on WSDL files however applying these design changes is fully manual and time consuming as discussed in the next section. These interface design changes correspond mainly to Interface Decomposition, Interface Merging (to merge multiple interfaces) and Move Operation (to move an operation between different interfaces).

Web service interface defects are defined as bad design choices that can have a negative impact on the interface quality such as maintainability, changeability and comprehensibility which may impacts the usability and popularity of services [3]. To this end, recent studies defined different types of Web services design defects [1][15]. In our experiments, we focus on the seven following Web service defect types: *God object Web service (GOWS):* implements a high number of operations related to different business and technical abstractions in a single service. *Fine grained Web service (FGWS):* is a too fine-grained service whose overhead (communications, maintenance, and so on) outweighs its utility. *Chatty Web service (CWS):* represents an antipattern where a high number of operations are required to complete one abstraction. *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. *Ambiguous Web service (AWS):* is an antipattern where developers use ambiguous or meaningless names for denoting the main elements of interface elements (e.g., port types, operations, messages). *Redundant PortTypes (RPT):* is an antipattern where multiple port Types are duplicated with the similar set of operations. *CRUDy Interface (CI):* is an antipattern where the design encourages services the RPC-like behavior by declaring create, read, update, and delete (CRUD) operations, e.g., createX(), readY(), etc.

We choose these defect types in our interactive interface design tool because they are the most frequent and hard to detect [8], cover different interface design issues, due to the availability of defect examples and could be detected using a tool proposed in the literature [1][8][19][7].

B. Related Work and Challenges

Detecting and specifying antipatterns in SOA and Web services is a relatively new area. The first book in the literature was written by Dudney et al. [3] 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 [6]. Furthermore, Rodriguez et al. [14]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 [15], 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. [4] have proposed an interesting

approach towards generating WSDL documents with less antipatterns using text mining techniques.

In our previous work [1], we proposed a search-based approach based on standard GP to find regularities, from examples of Web service antipatterns, to be translated into detection rules. However, the proposed approach can deal only with Web service interface metrics and cannot consider all Web service antipattern symptoms.

Recently, few studies are proposed to restructure the design of the Web services interface [6][5][2]. We can distinguish two main categories: manual and fully-automated techniques. The manual approaches propose a set of interface design changes that the user can select and execute to split an interface, extract an interface and merge two interfaces [11]. However, manual refactoring of the interface's design is a tedious task for developers that involve exploring the whole operations in the interface to find the best refactoring solution that improves the modularity of an interface. In the fully-automated approach, developers should accept the entire design changes solution and existing tools do not provide the flexibility to adapt the suggested solution interactively. In addition, most of these manual and fully-automated techniques focus on fixing design defects rather than the modularity of the interface [1][8].

In the following, we introduce some issues and challenges related to restructuring the design quality of the Web service interfaces. Figure 1 illustrates a fine-grained service that can lead to a system with a poor performance due to an excessive number of calls to one interface regrouping all the operations. Thus, it is critical to fix this issue by creating new port Types that group together the most cohesive operations to decompose the Amazon Simple Notification Service interface.

Overall, there is no consensus on how to decide if a 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. 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 as illustrated in Figure 1. Although we can measure the size of an interface, an appropriate threshold value is not trivial to define. An interface considered large by a community of service users could be considered average by others. Thus, it is important to consider the user in the loop when identifying such design violations.

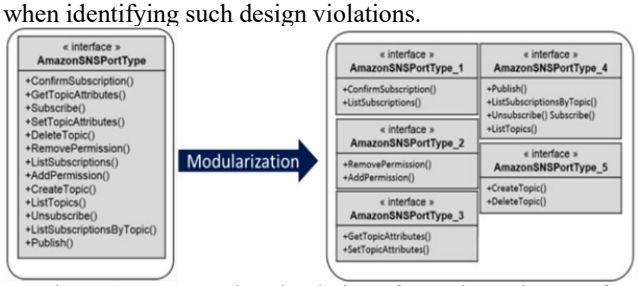

Figure 1. Restructuring the design of a Web service Interface example (Amazon Simple Notification Service)

Several possible levels of interaction are not considered by existing Web services interface refactoring techniques. Overall, most of refactoring studies are based on the use of quality metrics as a fitness function to evaluate the quality of the design after applying design changes. However, these metrics can only evaluate the structural improvements. Furthermore, the efficient evaluation of the suggested refactoring from a semantic perspective requires an interaction with the designer. In addition, the symptoms of design defects are difficult to formalize using quality metrics due to the very subjective process to identify them that depends on the programming context and the preferences of developers. Finally, the definition of a fitness function based on quality metrics can be expensive.

To address these challenges, we describe in the next section our approach based on machine learning and heuristic-based techniques to evaluate the Web services modularization solutions without the need to explicitly define a fitness function. This work represents one of the first studies in this area.

III. REFACTORING AS AN INTERACTIVE HEURISTIC-BASED LEARNING PROBLEM

A. *Approach Overview*

As described in Figure 2, our approach takes as input the Web services interface to modularize, list of possible operators (decompose a port Type or merge port Types or move operations) and the number of user's interactions during the search process. It generates as output the best sequence of design changes/operators that improves the quality of the Web service interface. Our approach is composed of two main components: the interactive component (IGA) and the learning module (LGA).

The algorithm starts first by executing the IGA component where the designer evaluates the modularization solutions manually generated by a genetic algorithm (GA) [18] for a number of iterations. The user evaluates the feasibility and the efficiency/quality of the suggested suggestions one by one since each modularization solution is a sequence of change operator (decompose or merge or move). Thus, the user classifies all the suggested design changes (modules) as good or not one by one based on his preferences and gives the different port Types values between 0 and 1.

After executing the IGA component for a number of iterations, all the evaluated solutions by the user are considered as training set for the second component LGA of the algorithm. The LGA component executes an Artificial Neural Network (ANN)[10] to generate a predictive model to approximate the evaluation of the interface modularization solutions in the next iteration of the GA. Thus, our approach does not require the definition of a fitness function. Alternatively, the LGA incorporates many components to approximate the unknown target function f. Those components are the training set, the learning algorithm and the predictive model. For each new sequence of refactoring X_{k+1}, the goal of learning is to maximize the accuracy of the evaluation y_{k+1}. We applied the ANN as being among the most reliable predictive models, especially,

in the case of noisy and incomplete data. Its architecture is chosen to be a multilayered architecture in which all neurons are fully connected; weights of connections have been, randomly, set at the beginning of the training. Regarding the activation function, the sigmoid function is applied [12] as being adequate in the case of continuous data. The network is composed of three layers: the first layer is composed of p input neurons. Each neuron is assigned the value x_{kt} . The hidden layer is composed of a set of hidden neurons. The learning algorithm is an iterative algorithm that allows the training of the network. Its performance is controlled by two parameters. The first parameter is the momentum factor that tries to avoid local minima by stabilizing weights. The second factor is the learning rate which is responsible of the rapidity of the adjustment of weights.

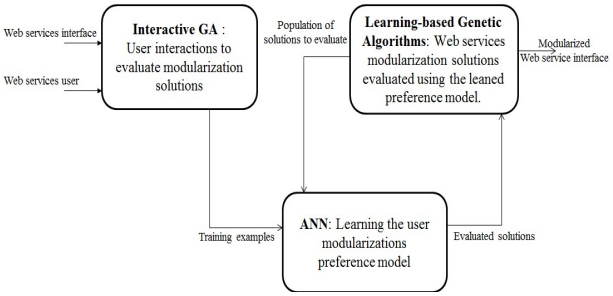

Figure 2. Approach overview.

B. Algorithm Adaptation

1) Solution coding

A solution consists of a sequence of n interface change operations assigned to a set of port types. A port type could contain one or many operations but an operation could be assigned to only one port type. A vector-based representation is used to cluster the different operations of the original interface, taken as input from the WSDL file description, into appropriate interfaces, i.e., port types. Figure 3 describes an example of 5 operations assigned to two port types.

PortType2	*PortType1*	*PortType1*	*PortType1*	*PortType2*
Op1	*Op2*	*Op3*	*Op4*	*Op5*

Fig. 3. Example of a solution representation

The initial population is generated by randomly assigning a sequence of operations to a randomly chosen set of port Types. The size of a solution, i.e. the vector's length corresponds to the number of operations of the Web service however the number of port Types is randomly chosen between upper and lower bound values. The determination of these two bounds is similar to the problem of bloat control in genetic programming [18] where the goal is to identify the tree size limits. The number of required port types depends on the size of the target interface design. Thus, we performed, for each target design, several trial and error experiments using the HyperVolume (HP) [18] performance indicator to determine the upper bound after which, the indicator remains invariant. For the lower bound, it is arbitrarily chosen.

2) Training Set and Data Normalization

Before the learning process, the data used in the training set should be normalized. In our case, we choose to apply the Min-max technique since it is among the most accurate techniques according to [16]. We used the following data representation to the GA-based learning problem using ANN for software refactoring. Let us denote by E the training set of the ANN. It is composed of a set of couples that represent the refactoring sequence and its evaluation.

$$E = \{(X_1, y_1),(X_2, y_2),(X_3, y_3),..,(X_k, y_k),...,(X_n, y_n)\}, k \in [1..n]$$

X_k is an interface refactoring sequence represented as $X_k = \lfloor x_{k1}, x_{k2},...,x_{kt},...,x_{kp} \rfloor, t \in [1..p]$.

y_k is the evaluation associated to the k[th] refactoring sequence in the range $y_k \in [0..1]$.

Let's denote by O the matrix that includes numerical values related to the set of refactorings and by Y the vector that contains numerical values representing X_k's evaluations. O is composed of n lines and p columns where n is equal to the number of refactoring sequences and p is equal to the number of solutions.

$$O = \begin{bmatrix} x_{11} & x_{12} & ... & x_{1p} \\ x_{21} & x_{22} & ... & x_{2p} \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ x_{n1} & x_{n2} & ... & x_{np} \end{bmatrix} \quad Y = \begin{bmatrix} y1 \\ y2 \\ . \\ . \\ . \\ y_n \end{bmatrix}$$

3) Change operators

In each search algorithm, the variation operators play the key role of moving within the search space with the aim of driving the search towards optimal solutions. For the crossover, we use the one-point crossover operator. It starts by selecting and splitting at random two parent solutions. Then, this operator creates two child solutions by putting, for the first child, the first part of the first parent with the second part of the second parent, and vice-versa for the second child. It is important to note that in multi-objective optimization, it is better to create children that are close to their parents to have a more efficient search process. For mutation, we use the bit-string mutation operator that picks probabilistically one or more modularization operations from its or their associated sequence and replaces them by other ones from the initial list of possible refactorings.

When applying the change operators, different pre- and post-conditions are checked to ensure the applicability of the newly generated solutions such as removing redundant operations or conflicts between operations such as assigning the same operation to two different port types.

IV. VALIDATION

To evaluate the ability of our Web services modularization framework to generate a good design quality, we conducted a set of experiments based on 82 real-world web services as described in Table 1. the obtained results are subsequently statistically analyzed with the aim of comparing our proposal with a variety of existing fully-automated approaches. In this section, we first present our

research questions and then describe and discuss the obtained results.

A. Research Questions

We defined three research questions that address the applicability, performance in comparison to existing fully-automated Web services modularization approaches [5][2], and the usefulness of our approach. The three research questions are as follows:

RQ1: To what extent can our approach recommend relevant Web services design improvements?

RQ2: How does our interactive formulation perform compared to fully-automated Web services restructuring techniques?

RQ3: Can our approach be useful for the users of Web services (the developers of service-based systems)?

To answer these research questions, we considered the best interface design restructuring solutions recommended by our approach. To answer RQ1, it is important to validate the proposed modularization solutions on the different Web services highlighted in Table 1. We asked a group of developers, as detailed in the next section, to manually modularize the design of the different interfaces considered in our experiments. Then, we calculated precision and recall scores to compare between the generated design and the expected one:

$$\mathrm{PR}_{\mathrm{precision}} = \frac{\text{suggested portTypes} \cap \text{expected portTypes}}{\text{suggested portTypes}} \in [0,1]$$

$$\mathrm{RC}_{recall} = \frac{\text{suggested portTypes} \cap \text{expected portTypes}}{\text{expected portTypes}} \in [0,1]$$

When calculating the precision and recall, we consider a two port types are similar if they contain the same operations. We divided the participants in groups to make sure that they do not use our tool on the Web services that they are asked to manually modularize.

Another metric that we considered for the quantitative evaluation is the percentage of fixed design antipatterns (*NF*) by the proposed modularization solution. The detection of design antipatterns after applying a modularization solution is performed using the detection rules of our previous work [1]. Formally, *NF* is defined as

$$NF = \frac{\#\text{fixed desing antipatterns}}{\#\text{design antipatterns}} \in [0,1]$$

For the qualitative validation, we asked groups of potential users of our Web services modularization tool to evaluate, manually, whether the suggested interface design modularizations are feasible and efficient at improving the quality of Web services interface design. We define the metric Manual Correctness (*MC*) to mean the number of meaningful Web services interface refactorings divided by the total number of recommended refactorings by our tool. *MC* is given by the following equation:

$$MC = \frac{\#\text{correct modularization operations}}{\#\text{proposed modularization operations}}$$

To answer RQ2, we compared our approach to two other existing fully-automated Web services decomposition techniques [5][2]. Ouni et al. [2] proposed an approach to decompose Web services using graph partitioning to improve cohesion. Similarly, Athanasopoulos et al. [5] used a greedy algorithm to decompose the interface based on cohesion as well. All these existing techniques are fully-automated and do not provide any interaction with the developers to update their solutions towards a desired design. We also compared the running time *T* of the proposed algorithm comparing to fully automated techniques. Thus, we used the metrics *PR, RC, T* and *NF* to perform the comparisons.

To answer RQ3, we used a post-study questionnaire that collects the opinions of Web service developers on our tool as described in the next section. Thus, we asked these participants to use both our tool and the automated framework proposed by Ouni et al. [1] on different sets of Web services. The participants were asked to make changes, when appropriate, to the final solution of the automated approach of Ouni et al. [1]. Thus, we can check whether the interactive component of the proposed interactive approach makes a real contribution, or whether the same effect can be attained by just fixing the output of the automated remodularization approaches. We measured the time spent by the developers on using our interactive approach and the automated techniques. Then, we compared between the outcomes of the survey questions for both interactive and fully automate techniques.

TABLE I. WEB SERVICE STATISTICS

Web Service Provider	#services	#operations (min, max)
FedEx	19	(13, 36)
Amazon	16	(16, 93)
Yahoo	18	(11, 41)
Ebay	12	(13, 37)
Microsoft	17	(11, 59)

B. Research Questions

We extracted a set of 82 well-known Web services from an existing benchmark [1][5] as detailed in Table 1. All studied services are widely used in different contexts and provided by Amazon, FedEx, Ebay, Microsoft and Yahoo, five major Web service providers. We selected these Web services for our validation because they range from medium to large-sized interfaces, which have been actively developed and changed over several years. Our study involved 36 participants from the University of Michigan to use and evaluate our tool. Participants include 27 master students in Software Engineering and 9 Ph.D. students in Software Engineering. All the participants are volunteers and familiar with Web services and refactoring in general. The experience of these participants on programming ranged from 3 to 17 years. 19 out of the 36 participants are currently active programmers as well in software industry with a minimum experience of 3 years. Participants were first asked to fill out a pre-study questionnaire containing nine questions. The questionnaire helped to collect background information such as their role within the company, their programming

experience, their familiarity with Web services. As part of the Software Quality Assurance graduate course, all the participants attended two lectures about Web services design quality, modularization and passed five tests to evaluate their performance to evaluate and suggest interface design modularization solutions.

As described in Table 2, we formed 6 groups. Each of the 6 groups is composed by 6 participants. Table 2 summarizes the survey organization including the list of Web services and the algorithms evaluated by each of the groups. The groups were formed based on the pre-study questionnaire and the tests result to make sure that all the groups have almost the same average skills. Consequently, each group of participants who accepted to participate in the study received a questionnaire, a manuscript guide to help them to fill the questionnaire, the tools and results to evaluate the Web services design. Since the application of remodularization solutions is a subjective process, it is normal that not all the developers have the same opinion. In our case, we considered the majority of votes to determine if suggested solutions are correct or not.

TABLE II. SURVEY ORGANIZATION

Groups	Web Services
Group 1	FedEx
Group 2	Amazon
Group 3	Yahoo
Group 4	Ebay
Group 5	Microsoft, Ebay
Group 6	FedEx, Yahoo

We executed three different scenarios. In the first scenario, we asked every participant to manually modularize a set of Web services. As an outcome of the first scenario, we calculated the differences between the recommended modularizations and the expected ones (manually suggested by the users/developers). The evaluate the fixed Web services design antipatterns, we focus on the ones defined in Section 2. In the second scenario, we asked the users to manually evaluate the recommended solution by our algorithm. We performed a cross-validation between the groups to avoid the computation of the *MC* metric being biased by the developer's feedback. In the third scenario, we collected their opinions of the participants based on a post-study questionnaire that will be detailed before in this section. The participants were asked to justify their evaluation of the solutions and these justifications are reviewed by the organizers of the study.

Parameter setting influences significantly the performance of a search algorithm. For this reason, for each algorithm and for each Web service, we perform a set of experiments using several population sizes: 20, 30 and 50. We limited the interaction with the user in our approach to a maximum of 30. The stopping criterion was set to 1000 evaluations for all algorithms to ensure fairness of comparison. The other parameters' values were fixed by trial and error and are as follows: (1) crossover probability = 0.5; mutation probability = 0.2 where the probability of gene modification is 0.1. Each algorithm is executed 30 times with each configuration and then the comparison between the configurations is done using the Wilcoxon test. To achieve significant results, for each couple (algorithm, Web service), we use the trial and error method to obtain a good parameter configuration.

Since metaheuristic algorithms are stochastic optimizers, they can provide different results for the same problem instance from one run to another. For this reason, our experimental study is based on 30 independent simulation runs for each problem instance of the automated approaches and the obtained results are statistically analyzed by using the Wilcoxon rank sum test with a 95% confidence level (α = 5%). The latter tests the null hypothesis, H0, that the obtained results of two algorithms are samples from continuous distributions with equal medians, against the alternative that they are not, H1. The p-value of the Wilcoxon test corresponds to the probability of rejecting the null hypothesis H0 while it is true (type I error). A p-value that is less than or equal to α (≤ 0.05) means that we accept H1 and we reject H0. However, a p-value that is strictly greater than α (> 0.05) means the opposite. In fact, for each problem instance, we compute the p-value obtained by comparing existing studies [5][2] results with our approach ones. In this way, we determine whether the performance difference between our technique and one of the other approaches is statistically significant or just a random result. The results presented were found to be statistically significant on 30 independent runs of the fully-automated approaches using the Wilcoxon rank sum test with a 95% confidence level ($\alpha < 5\%$) as detailed in the next section.

C. Results and Discussions

Results for RQ1. As described in Figures 4 and 5, we found that a considerable number of proposed port types, with an average of more than 81% in terms of precision and recall on all the 82 Web services, were already suggested manually (expected refactorings) by the users (software development team). The achieved recall scores are slightly higher, in average, than the precision ones since we found that some of the port types suggested manually by developers do not exactly match the solutions provided by our approach. In addition, we found that the slight deviation with the expected port types is not related to incorrect ones but to the fact that different possible modularization solutions could be optimal.

We evaluated the ability of our approach to fix several types of interface design antipatterns and to improve the quality. Figure 6 depicts the percentage of fixed code smells (*NF*). It is higher than 82% on all the Web services, which is an acceptable score since users may not be interested to fix all the antipatterns in the interface. We reported the results of our empirical qualitative evaluation in Figure 7 (*MC*). As reported in Figure 7, most of the Web services modularization solutions recommended by our interactive approach were correct and approved by developers. On average, for the different Web services, 88% of the created port types and applied changes to the initial design are

considered as correct, improve the quality, and are found to be useful by the software developers of our experiments. Thus, we found that the slight deviation with the expected design is not related to incorrect changes but to the fact that the developers have different scenarios/contexts in using the different operations.

To summarize and answer RQ1, the experimentation results con-firm that our interactive approach helps the participants to re-structure their Web service interface design efficiently by finding the relevant portTypes and improve the quality of all the 22 Web services.

Results for RQ2. Figures 4,5,6 and 7 confirm the average superior performance of our interactive learning GA approach compared to the two existing fully automated Web service modularization techniques [5][2]. Figure 7 shows that our approach provides significantly higher manual correctness results (*MC*) than all other approaches having *MC* scores respectively between 41% and 62%, on average as MC scores on the different Web services. The same observation is valid for the precision and recall as described in Figures 4 and 5. The outperformance of our technique in terms of percentage of fixed defects, as described in Figure 6, can be explained by the fact that the main goal of existing studies is not to mainly fix these defects (not considered in the fitness function by the work of Ouni et al. [2]).

In conclusion, our interactive approach provides better results, on average, than all existing fully-automated Web services modularization techniques (answer to RQ2).

Results for RQ3. To further analyze the obtained results, we have also asked the participants to take a post-study questionnaire after completing the different validation and tasks using our interactive approach and the two techniques considered in our experiments. The post-study questionnaires collected the opinions of the participants about their experience in using our approach compared to fully-automated tools. The post-study questionnaire asked participants to rate their agreement on a Likert scale from 1 (complete disagreement) to 5 (complete agreement) with the following statements: (1) The interactive interface modularization recommendations using our predictive modeling approach are a desirable feature to improve the quality of Web services interface. (2) The interactive manner of recommending modularization solutions by our GA learning approach is a useful and flexible way to consider the user perspective compared to fully-automated tools.

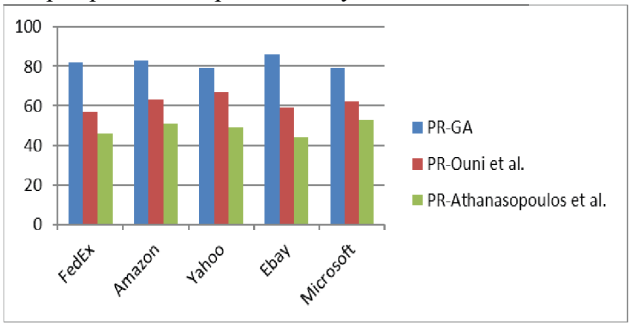

Figure 4. Median precision (PR) value over 30 runs on all Web services using the different modularization techniques with a 95% confidence level ($\alpha < 5\%$).

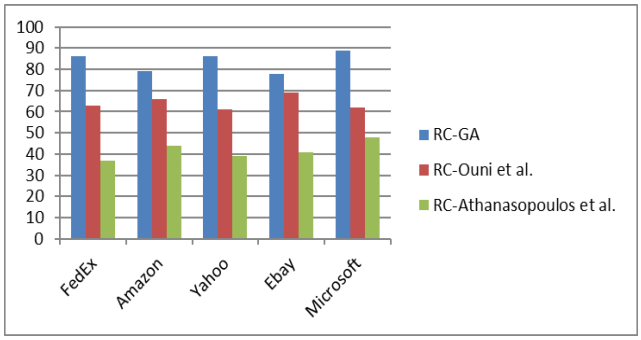

Figure 5. Median recall (RC) value over 30 runs on all Web services using the different modularization techniques with a 95% confidence level ($\alpha < 5\%$).

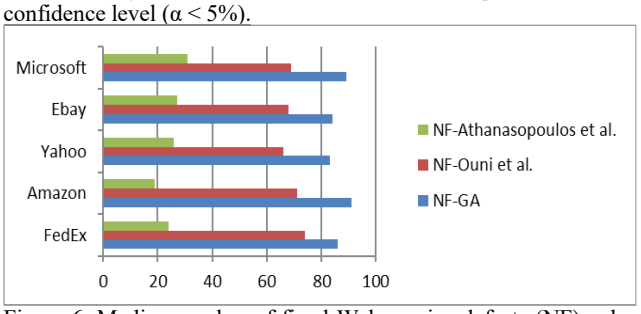

Figure 6. Median number of fixed Web service defects (NF) value over 30 runs on all Web services using the different modularization techniques with a 95% confidence level ($\alpha < 5\%$).

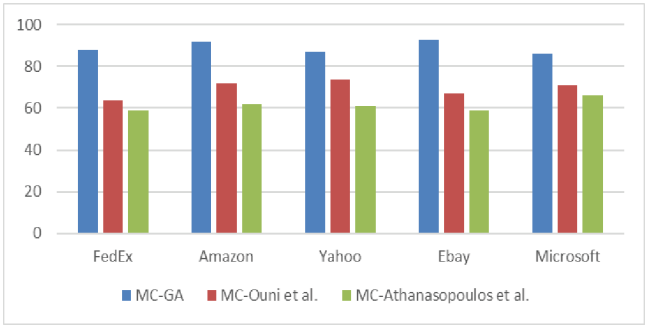

Figure 7. Median manual correctness (MC) value over 30 runs on all Web services using the different modularization techniques with a 95% confidence level ($\alpha < 5\%$).

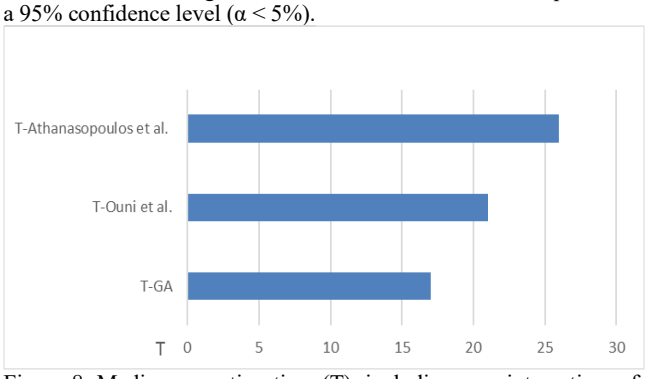

Figure 8. Median execution time (T), including user interaction, of the different tools over 30 runs on all Web services with a 95% confidence level ($\alpha < 5\%$).

The agreement of the participants was 4.6 and 4.2 for the first and second statements respectively. This confirms the

usefulness of our approach for the users of our experiments. The remaining questions of the post-study questionnaire were about the benefits and the limitations (possible improvements) of our interactive approach.

We summarize in the following the feedback of the users. Most of the participants mention that our interactive approach is much faster and easy to use compared to the manual restructuring of the interface since they spent a long time with manual changes to create port types and move operations. Thus, the developers liked the functionality of our tool that helps them to modify a port type based on the recommendations. The participants also suggested some possible improvements to our interactive approach. Some participants believe that it will be very helpful to extend the tool by adding a new feature to decompose multiple services into interfaces based on the dependency between them. Another possibly suggested improvement is to consider the users invocation data to restructure the interface.

In our evaluation, we considered measuring the time spent by the different developers to use our interactive tool and automated Web services modularization techniques [5][2]. We allowed the user to fix the solutions proposed by the automated tools to reach an acceptable design. Figure 8 shows the average results of the execution time of the different tools per Web service including the interaction time. The developers found that automated techniques generate solutions that require a lot of effort to inspect and manually adjust the proposed design. All developers expressed a high interest in the idea of the interactive tool that can incorporate their preferences by evaluating manually very few solutions. Furthermore, the execution time results confirm that few number of interactions are required with the user and that the generate solutions do not require a lot of changes to meet the developers' preferences.

Threats to validity. In our experiments, construct validity threats are related to the absence of similar work that uses interactive learning techniques for Web services modularization. Thus, we compare our proposal with several other fully-automated techniques. Another construct threat can be related to the corpus of manually applied design changes since developers do not all agree if a candidate is a code smell or not. We will ask some other experts to extend the existing corpus and provide additional feedback regarding the suggested solutions. In addition, the parameter tuning of the different algorithms can be another threat related to our experiments that should be addressed in the future by further experiments with different parameters.

V. Conclusion and Future Work

This paper presented a novel interactive search-based learning Web services modularization approach that does not require the definition of a fitness function. The user is asked to evaluate manually few modularization solutions suggested by a Genetic Algorithm (GA) for few iterations then these examples are used as a training set for the ANNs to evaluate the solutions of the GA in the next iterations. We evaluated our approach on a benchmark of Web services. We report the results on the efficiency and effectiveness of our approach, compared to existing approaches [5][2].

In future work, we are planning to investigate an empirical study to consider additional Web services and larger set of refactorings in our experiments. We are also planning to extend our approach to include the detection of refactoring opportunities in Web services using our interactive heuristic-based learning approach.

References

[1] 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, 2015.

[2] A. Ouni, Z. Salem, K. Inoue, and M. Soui, "SIM: an automated approach to improve web service interface modularization," in IEEE International Conference on Web Services, ICWS 2016, San Francisco, CA, USA, June 27 - July 2, 2016, 2016, pp. 91–98.

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

[4] 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.

[5] D. Athanasopoulos, A. V. Zarras, G. Miskos, and V. Issarny, "Cohesion-Driven Decomposition of Service Interfaces Without Access to Source Code," IEEE Transactions on Services Computing, vol. 8, no. JUNE, pp. 1–18, 2015.

[6] D. Romano and M. Pinzger, "Analyzing the evolution of web services using fine-grained changes," in IEEE International Conference on Web Services (ICWS), June 2012, pp. 392–399.

[7] Dilan Sahin, Marouane Kessentini, Manuel Wimmer, Kalyanmoy Deb: Model transformation testing: a bi-level search-based software engineering approach. Journal of Software: Evolution and Process 27(11): 821-837 (2015)

[8] Hanzhang Wang, Marouane Kessentini, Ali Ouni: Bi-level Identification of Web Service Defects. ICSOC 2016: 352-368

[9] Hanzhang Wang, Marouane Kessentini, Ali Ouni: Bi-level Identification of Web Service Defects. ICSOC 2016: 352-368

[10] Idri, A., Khoshgoftaar, T.M., Abran, A.: Can neural networks be easily interpreted in software cost estimation. In Proc. of the IEEE International Conference on Fuzzy Systems, pp. 1162–1167 (2002).

[11] J.S. Bridle, M. Perepletchikov, C. Ryan, and K. Frampton, "Cohesion metrics for predicting maintainability of service-oriented software," in the 7th International Conference on Quality Software, Oct 2007, pp. 328–335.

[12] Jayalakshmi, T., Santhakumaran, A.: Statistical normalization and back propagation for classification. International Journal of Computer Theory and Engineering, 3(1):1793–8201 (2011).

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

[14] M. Crasso, J. M. Rodriguez, A. Zunino, and M. Campo. Revising WSDL Documents: Why and How. Internet Computing, IEEE, (5):48{56.

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

[16] 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.

[17] Marouane Kessentini, Rim Mahouachi, Khaled Ghédira: What you like in design use to correct bad-smells. Software Quality Journal 21(4): 551-571 (2013)

[18] Mitchell M. An introduction to genetic algorithms. MIT press; 1998.

[19] Rafi Almhana, Wiem Mkaouer, Marouane Kessentini, Ali Ouni: Recommending relevant classes for bug reports using multi-objective search. ASE 2016: 286-295

[20] Usman Mansoor, Marouane Kessentini, Bruce R. Maxim, Kalyanmoy Deb: Multi-objective code-smells detection using good and bad design examples. Software Quality Journal 25(2): 529-552 (2017)