Personalized Web Services Interface Design Using Interactive Computational Search

by

FNU Jirigesi

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
(Software Engineering)
in the University of Michigan-Dearborn
2017

Master's Thesis Committee:

Assistant Professor Marouane Kessentini, Chair
Professor Qiang Zhu
Associate Professor Brahim Medjahed

DEDICATION

To My Parents.

ACKNOWLEDGEMENTS

It is with a great joy that I reserve these few lines of gratitude and deep appreciation to all those who directly or indirectly contributed to the completion of this work:

I express my greatest gratitude to Dr. Marouane Kessentini, who dedicated all his wonderful time to collaborate, support and lead me to the end of this piece of work. His advices, dedication, availability, relevant comments, corrections and committeemen led to the success of this work.

I also express my greatest thanks to SBSE members who supported me with valuable feedback and always kindly encouraged me to succeed this project.

I thank all the lecturers of the CIS master degree who have used their valuable time to transmit the knowledge that help in putting this work together me.

Finally, I wish to express my deep gratitude and thank my family who has consistently expressed its unconditional support and encouragement.

All those who contributed in one way or another, to make this work, can be found here, the crowning of their efforts.

Thank you.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Most of successful Web services evolve through a process of continuous change due to several reasons such as improving the quality, fixing bugs and adding new features. However, this evolution process may weaken the design of the Web service's interface by including a large number of non-cohesive operations and make it unnecessarily complex for users to find relevant operations to be used by their services-based systems.

In this thesis, we propose a remodularization recommendation approach that dynamically adapts and interactively suggests a possible modularization of the Web services interface design to users/developers and takes their feedback into consideration. Our approach uses an interactive multi-criteria decision making algorithm, based on interactive NSGA-II, to find a set of good design interface modularization solutions that find a trade-off between improving several interface design quality metrics (e.g. coupling, cohesion, number of portTypes and number of antipatterns), maximizing the reuse of user-interface interaction history patterns identified from previous releases and satisfying the interaction constraints learnt from the user feedback during the execution of the algorithm while minimizing the deviation from the initial design.

We evaluated our approach on a set of 22 real-world Web services, provided by Amazon and Yahoo. Statistical analysis of our experiments shows that our dynamic interactive Web services interface modularization approach performed significantly better than the state-of-the-art modularization techniques.

Key words: Web services, interface design, quality, multi-objective search

CHAPTER 1: Introduction

Web services promote software reuse by providing reusable services to end users who can compose them to implement or update an existing system [2]. One of the main key factors for deploying successful and popular services is assuring a well-designed interface for users (service's subscribers) to find relevant and high-quality operations to implement the features of their service-based systems [6]. The Web services interface is provided by the service providers such as FedEx, Google, PayPal and Google. It is the most critical component in the service-oriented architecture (SOA) since the interface is the only visible component to the users.

The evolution of Web services may have a negative impact on the design quality of the interface by including a large number of non-cohesive operations and make it unnecessarily complex for users to find relevant operations to be used their services-based systems. An example of well-known interface design defect is God object Web service (GOWS) [3] which implements many operations related to different business and technical abstractions in a single service interface leading to low cohesion of its operations and unavailability to end users because it is overloaded. The choice of how operations should be exposed through a service interface can have an impact on the performance, popularity and reusability [7] and it is not a trivial task. On one hand, Web services interface including a high number of operations lead their clients to invoke their interfaces many times which significantly deteriorate the performance. On the other hand, aggregating several operations of an interface into one operation will reduce the reusability of the service.

Despite its importance, very few studies focused on personalizing and improving the design of Web services interface for the users/subscriber [4] [5]. The majority of existing work [3] [11] [12] addressed the problem of the detection of design defects of Web services interface based on declarative rule specification. In these settings, rules are manually defined to identify the key symptoms that characterize an interface design defect using combinations of mainly quantitative metrics. For each possible interface design defect, rules that are expressed in terms of metric combinations need high calibration efforts to find the right threshold value for each metric. Another important issue is that translating symptoms into rules is not obvious because there is no consensual symptom-based definition of design defects. In fact, the identification of these interface design defects is a very subjective process and requires integrating the user in the loop. These difficulties explain a large portion of the high false-positive rates reported in existing research. Very recent work [4][5] addressed the problem of fixing these design defects by fully-automatically decomposing Web services interface based only on the cohesion metric. 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. In addition, the history of interactions between the users and the current interface design could be important to understand the dependency between the operations within an interface and generate a personalized interface. However, these aspects were not considered by existing studies.

In this thesis, we propose a remodularization recommendation approach that dynamically adapts and interactively suggests a possible modularization of the Web services interface design to developers and takes their feedback into consideration. Our approach uses an interactive multi-criteria decision making algorithm, based on interactive NSGA-II[15], to find a set of good design interface modularization solutions that find a trade-off between improving several interface design

quality metrics (e.g. coupling, cohesion, number of portTypes and number of antipatterns), maximizing the reuse of user-interface interaction history patterns identified from previous releases and satisfying the interaction constraints learnt from the user feedback during the execution of the algorithm while minimizing the deviation from the initial design. Based on this analysis, the interface modularization solutions are ranked and suggested to the developer one by one in an interactive fashion. The developer can approve, modify or reject each of the recommended operations or portTypes, and this feedback is then used to update the proposed rankings of recommended interface modularization solutions. After a number of interactions with the developer, the interactive NSGA-II algorithm is executed again on the new modified design interface to repair the set of interface modularization solutions based on the new changes and the feedback received from the users.

We evaluated our approach on a set of 22 real-world Web services, provided by Amazon and Yahoo. Statistical analysis of our experiments shows that our dynamic interactive Web services interface modularization approach performed significantly better than the state-of-the-art modularization techniques [4][5]. The primary contributions of this thesis can be summarized as follows:

1. This work introduces a novel interactive and personalized way to modularize and improve the quality of Web services interface using interactive dynamic multi-objective optimization. The proposed technique supports the adaptation of interface design solutions based on the user feedback while also taking into account other objectives such as the history of previous interactions from multiple releases and improving several quality attributes while minimizing the deviation from the initial design. To the best of our knowledge, we propose the first approach to interactively generate a personalized Web services interface.

2. This work 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 a benchmark of 22 real-world services. This thesis also evaluates the relevance and usefulness of the suggested interface design improvements for Web service users.

The remainder of this thesis is as follows: Chapter 2 presents the relevant background, a motivating example for the presented work and an overview of the related work; Chapter 3 describes the search algorithm; an evaluation of the algorithm is explained and its results are discussed in Chapter 4. Finally, concluding remarks and future work are provided in Chapter 5.

CHAPTER 2: Related Work

We first detail some required background information to understand the problem addressed in this work, then we present a motivating example to illustrate the limitations of existing studies. Finally, we present an overview of existing work.

2.1 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 [6]. A portType 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 such as Amazon EC2 that contains more than 100 operations in some releases. There are few WSDL design improvement tools [4][5] that have emerged to provide basic refactorings on WSDL files however applying these refactorings is fully manual and time consuming as discussed in the next section. These interface design refactorings correspond 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 [12]. They can be also considered as structural characteristics of the interface that may indicate a design problem that makes the service hard to evolve and maintain, and trigger refactoring. To this end, recent studies defined different types of Web services design defects [3]. 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 portTypes 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 [20], cover different interface design issues, due to the availability of defect examples and could be detected using a tool proposed in our previous work [3][12].

2.2 Problem Statement and Related Work

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 portTypes that group together the most cohesive operations to decompose the Amazon Simple Notification Service interface.

Recently, few studies are proposed to restructure the design of the Web services interface [4][5]. We can distinguish two main categories: manual and fully-automated techniques. The manual approaches propose a set of refactorings that the user can select and execute to split an interface, extract an interface and merge two interfaces [8]. However, manual refactoring of the design interface 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 have to accept the entire refactoring 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 [4][5]. Overall, 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. 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 in a given service/community of users could be considered average in another. Thus, it is important to consider the user in the loop when identifying such design violations.

Several possible levels of interaction are not considered by existing Web services interface refactoring techniques. It is easy for developers to identify large interfaces that should be refactored, but they find it is difficult, in general, to locate a target portType when applying a move operation. In addition, existing tools do not update their recommended refactoring solutions based on the user's feedback such as accepting, modifying or rejecting certain refactoring actions. While automation is important, it is essential to understand the points at which human oversight, intervention, and decision-making should impact on automation. Human developers/users might reject changes made by any automated technique. Especially if they feel that they have little control, there will be a natural reluctance to trust and use the automated design restructuring tool.

In addition to the above-mentioned limitations, existing studies propose only few quality metrics such as cohesion to decompose a Web service interface. However, several conflicting metrics have to be considered such as coupling, number of portTypes, cohesion, number of design defects, etc. Thus, it is critical to find a trade-off between these different metrics based on the preferences of the user. Furthermore, the history of the interaction between the users and the Web

service interface (invocations) is not considered by existing work when decomposing Web services design interfaces. In fact, users in general select operations that are related to each other's when implementing a specific feature. Thus, such information could be useful when regrouping operations together into portTypes.

Several studies addressed the problem of clustering and remodularization of object oriented (OO) applications in terms of packages organization. Anquetil et al. [22] used cohesion and coupling of modules within a decomposition of OO systems to evaluate its quality. Maqbool et al. [23]used hierarchical clustering in the context of software architecture recovery and modularization. On the other hand, Mancoridis et al. [24] proposed the first search-based approach to address the problem of software modularization using a single objective approach.

Harman et al. [25] used a genetic algorithm to improve subsystems decomposition by combining several quality metrics including coupling, cohesion, and complexity. Similarly, Seng et al. [26] treated the remodularization task as a single objective optimization problem using genetic algorithm to reduce violations of design principles. Later, Abdeen et al. [27] proposed a heuristic search-based approach for automatically optimizing (i.e., reducing) the dependencies between packages of a software system by moving classes beteween packages.

Recently, Mkaouer et al. [14] have proposed a multi-objective approach to finding optimal remodularization solutions that improve the structure of packages, minimize the number of changes, preserve semantics coherence, and reuse the history of changes. Despite these advances in OO systems modularization [28, 29, 30, 31, 32, 33, 34, 35], still this problem is not widely explored in the context of Web service interfaces.

To address the above-mentioned limitations, we propose in this thesis a new way for users to refactor the design of their Web services interface as a sequence of transformations based on different levels of interaction and dynamic adaptive ranking of the suggested refactorings. The next section describes the proposed interactive, dynamic and personalized Web services interface designing restructuring technique.

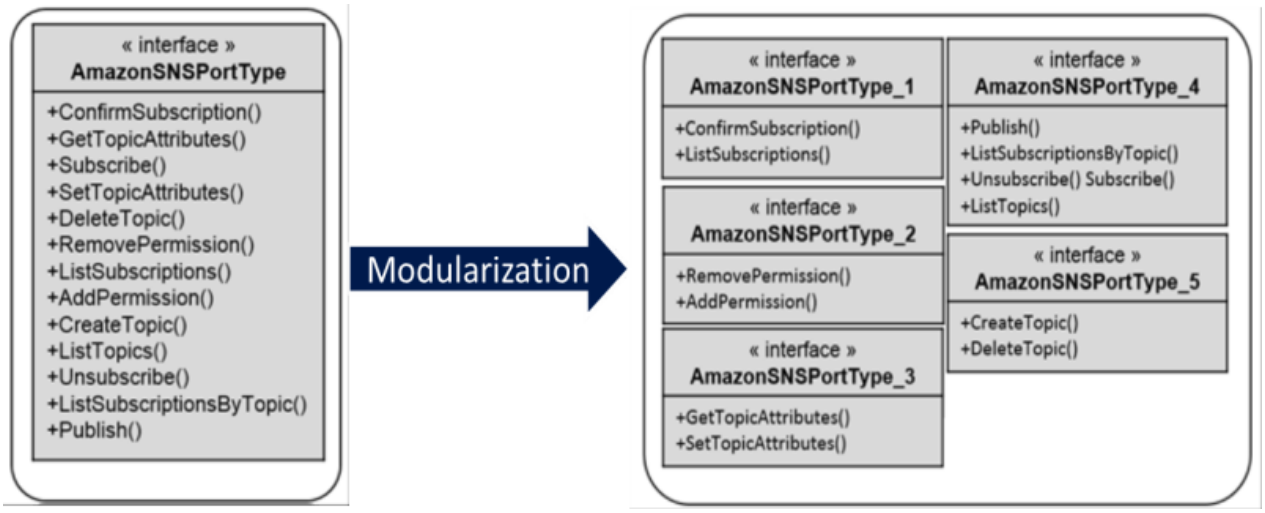

Fig. 1. Restructuring the design of a Web service Interface example (Amazon Simple Notification Service)

CHAPTER 3: Interactive Search Algorithm For Presonalized Design Of Web Services Interface

In this chapter, we present an overview of our approach and then we provide the details of our problem formulation and the solution approach.

3.1 Approach Overview

The goal of our approach is to propose a new dynamic interactive way for users to refactor the Web services interface based on their usage. The general structure of our approach is sketched in Fig. 2.

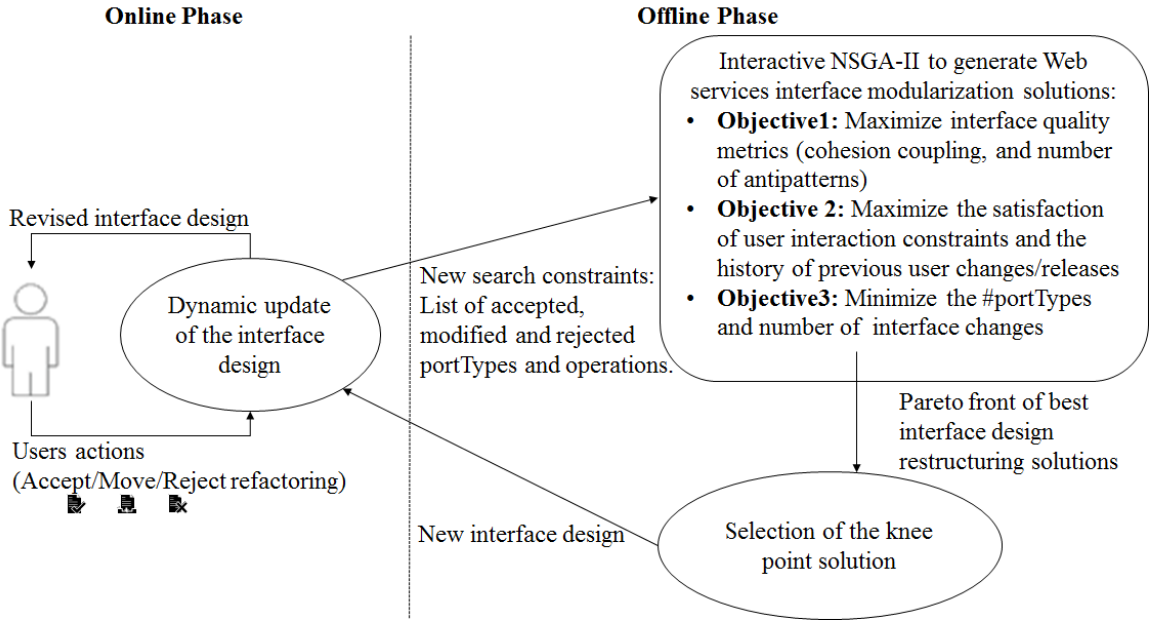

Fig. 2. Approach overview

Our technique comprises two main components. The first component is an *offline phase,* is executed first in the background, when the users are uploading the Web services interface that they want to restructure its design. During this phase, the multi-objective algorithm, NSGA-II [15], is executed for a number of iterations to find the non-dominated solutions balancing the three objectives of improving the interface design quality, which corresponds to minimizing the number of design antipatterns and improving design quality metrics (coupling and cohesion), the second objective of maximize the reuse of the user interaction and changes history from previous releases, and the third objective of minimizing the number of introduced changes and portTypes in the proposed interface design restructuring solutions.

The output of this first step of the offline phase is a set of Pareto-equivalent interface restructuring solutions that optimizes the above three objectives. As explained in Algorithm 1, the second step of the offline phase explores this Pareto front using a knee point strategy [14] . The knee point corresponds to the solution with the maximal trade-off between all fitness functions, i.e., a vector of the best objective values for all solutions. In order to find the maximal tradeoff, we use the trade-off worthiness metric proposed by Rachmawati and Srinivasan [14] to evaluate the worthiness of each solution in terms of objective value compromise. The solution nearest to the knee point is then selected.

The second component of our approach is an *online phase* to manage the interaction with the user. It dynamically updates the list of interaction constraints based on the feedback of the developer. This feedback can be to approve/apply or modify or reject some of the suggested operations location and portTypes in the interface. Thus, the goal is to guide, *implicitly*, the exploration of the search space of possible interface modularization solutions. Since the interactions constraints are updated dynamically, our interactive algorithm allows the implicit

move between non-dominated solutions of the Pareto front. The list of constraints that could be learnt will be discussed in the next section. For example, when a user accept a portType then the operations of that portType have to stay together in the next interactions of the algorithm but new operations could be moved to that portType. Another interaction option for the user is to specify desired values of the different metrics then the multi-objective algorithm will try to restructure the design of the interface to reach these desired values.

After a number of interactions, users may have modified or rejected a high number of suggested interface changes or have introduced several new changes manually. Whenever the users stop the interface design modularization session by closing the suggestions window, the first component of our approach is executed again on the background to update the last set of non-dominated interface modularization solutions by continuing the execution of NSGA-II based on the three objectives defined in the first component as described in Algorithm 1 and also the new constraints summarizing the feedback of the user. In fact, we consider the rejected portTypes or operations by the developer as constraints to avoid generating solutions containing similar portTypes in the next iterations to avoid putting together again the operations of that rejected portTypes in the next iterations of the algorithm. This may lead to reducing the search space and thus a fast convergence to better interface modularization solutions. Of course, the continuation of the execution of NSGA-II takes as input the updated version of the interface after the interactions with users.

The whole process continues until the developers decide that there is no necessity to restructure the interface any further.

Algorithm 1. Dynamic Interactive NSGA-II at generation *t*

Input

Sys: system to evaluate, P_t: parent population

Output

P_{t+1}

Begin

/* Test if any user interaction occurred in the previous iteration */

If UserFeedback = *TRUE* **then**

/* Rejected or Modified portTypes as constraints */

$C_t \leftarrow$ Get-Constraints();

/* Updated interface after applying changes */

Sys ← Get-Remodulazied-Interface();

UserFeedback ← *FALSE;*

End If

$S_t \leftarrow \emptyset, i \leftarrow 1$;

$Q_t \leftarrow$ Variation (P_t);

$R_t \leftarrow P_t \cup Q_t$;

$P_t \leftarrow$ evaluate (P_t, C_t, Sys);

$(F_1, F_2, ...) \leftarrow$ Non-dominationed-Sort (R_t);

Repeat

$S_t \leftarrow S_t \cup F_i$; $i \leftarrow i$+1;

Until $| S_t | \geq N$;

$F_l \leftarrow F_i$; //Last front to be included

If $| S_t | = N$ **then**

$P_{t+1} \leftarrow S_t$;

Else

$P_{t+1} \leftarrow \cup_{j=1}^{l-1} F_j$;

/*Number of points to be chosen from F_l*/

$K \leftarrow N - |P_{t+1}|$;

/*Crowding distance of points in F_l */

Crowding-Distance-Assignment(F_l);

Quick-Sort(F_l);

/*Choose K solutions with largest distance*/

$P_{t+1} \leftarrow P_{t+1} \cup$ Select(F_l, k);

End If

If t+1 = Threshold **then**

UserFeedback ← *TRUE;*

/* Select and rank the best front */

Rank-Solution (F_1); /* based on Algorithm 2 */

Threshold ← Threshold + t+1;

End If

End

3.2 Solution Approach

Most real world optimization problems encountered in practice involve multiple criteria to be considered simultaneously. These criteria, also called objectives, are often conflicting. Usually, there is no single solution that is optimal with respect to all these objectives at the same time, but rather many different designs exist which are incomparable per se. Consequently, contrary to Single-objective Optimization Problems (SOPs) where we look for the solution presenting the best performance, the resolution of a multi-objective optimization (MOP) yields a set of compromise solutions presenting the optimal trade-offs between the different objectives. When plotted in the objective space, the set of compromise solutions is called the Pareto front. The resolution of a MOP yields a set of trade-off solutions, called Pareto optimal solutions or non-dominated solutions, and the image of this set in the objective space is called the Pareto front. Hence, the resolution of a MOP consists in approximating the whole Pareto front.

In this work, we adapted one of the widely used multi-objective algorithms called NSGA-II and integrated the interactive component to it. NSGA-II is a powerful search method stimulated by natural selection that is inspired from the theory of Darwin. Hence, the basic idea of NSGA-II is to make a population of candidate solutions evolve toward the near-optimal solution in order to solve a multi-objective optimization problem. NSGA-II is designed to find a set of optimal solutions, called non-dominated solutions, also Pareto set. A non-dominated solution is the one which provides a suitable compromise between all objectives without degrading any of them. As

described in Algorithm 1, the first step in NSGA-II is to create randomly a population P_0 of individuals encoded using a specific representation. Then, a child population Q_0 is generated from the population of parents P_0 using genetic operators such as crossover and mutation. Both populations are merged into an initial population R_0 of size N. As a consequence, NSGA-II starts by generating an initial population based on a specific representation that will be discussed later, using the exhaustive list of interface operations given as input as mentioned in the previous section. Thus, this population stands for a set of possible solutions represented as sequences of portTypes (including the operations) which are selected and combined. After a number of iterations, the best solution (interface design modularization) will be presented to the user to get his feedback then the algorithm will continue to execute taking into consideration the new learnt interaction constraints.

To summarize, the main NSGA-II loop goal is to make a population of candidate solutions evolve toward the best clustering of interface operations into portTypes, i.e., the sequence that minimizes the coupling, number of antipatterns, number of portTypes and number of interface changes, and maximizes the cohesion and the satisfaction of the interaction constraints. During each iteration t, an offspring population Q_t is generated from a parent population P_t using genetic operators (selection, crossover and mutation). Then, Q_t and P_t are assembled in order to create a global population R_t. Then, each solution S_i in the population R_t is evaluated using our three fitness functions. We describe in the next sections, the different steps of adaption of the interactive NSGA-II algorithm to our problem.

A solution consists of a sequence of n interface operations assigned to a set of portTypes. A portType could contain one or many operations but an operation could be assigned to only one portType. The vector-based representation is used to cluster the different operations of the

interface, taken as input from the WSDL file description, into portTypes. Figure 3 describes an example of 5 operations assigned to two portTypes. A vector representation is automatically translated by our tool into a graphical interface as described in Figure 4.

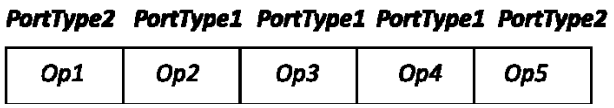

Fig. 3. Example of a solution representation

The initial population is generated by randomly assigning a sequence of operations to a randomly chosen set of portTypes. The size of a solution, i.e. the vector's length corresponds to the number of operations of the Web service interface however the number of portTypes 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 where the goal is to identify the tree size limits. Since the number of required portTypes depends mainly on the size of the target interface design, we performed, for each target design, several trial and error experiments using the HyperVolume (HP) performance indicator [19] to determine the upper bound after which, the indicator remains invariant. For the lower bound, it is arbitrarily chosen. The experiments section will specify the upper and lower bounds used in this study.

The generated solutions are evaluated using three fitness functions as detailed in the following.

Objective 1: Maximize the interface design quality metrics. This fitness function is defined as the average of three measures. The first measure is the number of design antipatterns that can be detected using the rules defined in our previous work [3][12] . The list of antipatterns is

discussed in Section 2. The second measure is the cohesion that corresponds to the degree to which the operations exposed in a service interface conceptually belong together [4] . We used, in this thesis, the definition of cohesion defined by [4] which is based on communicational and textual similarities between the operations within the same portType based on cosine similarity and call-graphs. The third measure is coupling within a service measures the relationships between implementation elements belonging to the same service [5] . Service interface coupling is a measure of how strongly a service interface is connected to or relies on other service interfaces. We used the existing definition of coupling based on the similarity between the operations within the same portType and the number of calls to other operations in different portTypes [5] .

$$f_1 = \frac{\#antipatternsAfterModularization}{\#antipatternsBeforeModularization} + Coupling - Cohesion$$

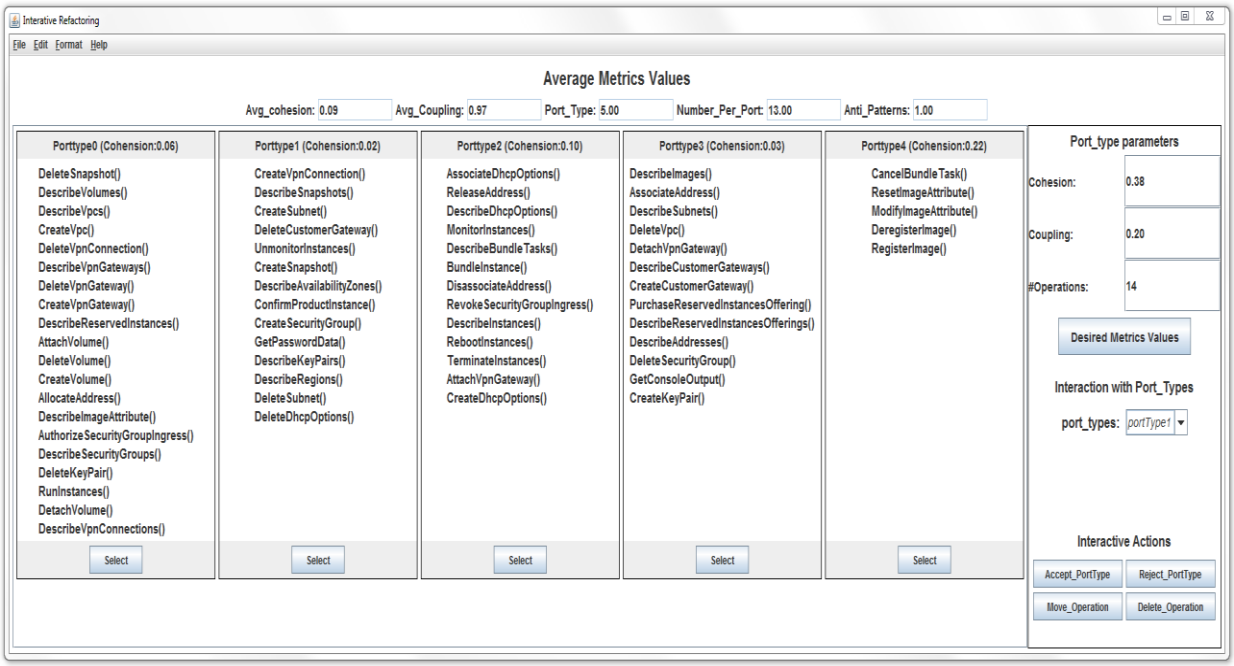

Fig. 4. The proposed interface design modularization tool.

Objective 2: Maximize the interaction and history-based function. This function maximizes the satisfaction of the constraints learnt from the interaction with user or minimizes the distance with the desired metrics, if specified by the user as described in Figure 5. In case that the user did not specify these desired values then we just ignore this component of the fitness function. Furthermore, the user has four other types of interaction, as described in Figures 3 and 6, that correspond to *accept a portType, reject a portType, move operation(s)* and *delete operation(s)*. Every of these user interaction actions will generate a set of constraints. When a portType is accepted, the list of operations in that portType should stay together in the next iterations but new operations could be added to the portType. In the case of a reject of a portType by the user, a constraint is generated to avoid regrouping together again these operations into the same portType. The application of a move operation action will generate a constraint to keep the moved operation in the targeted portType in the next iterations. When an operation is deleted, a constraint will be generated to avoid putting again that operation in the source portType in the next iterations. Another constraint considered by our fitness function is based on the history of previous releases (if available), when two operations were modified together in the same release by developers then a constraint will be generate to put them together in the same portType. Formally, the second fitness function to minimize is defined as follows:

$$f_2 = \frac{1}{\sum_{i=1}^{k} |MDesired_i - M_i|} + \frac{\#statisfiedHistroyConstra\text{int}s}{\#Constra\text{int}s}$$

Objective 3: Minimize the number of portTypes and the number of changes comparing to the initial design. The designer may have some preferences regarding the degree of the deviation

with the initial design of the interface thus we formally defined the fitness function as the following:

$$f_3 = \#\,portTypes + \#\,designChanges$$

The number of design changes is calculated based on the number of differences between the two vector representations of the initial design and the generated one, i.e. the number of operations of the new design assigned to different portTypes than the initial design.

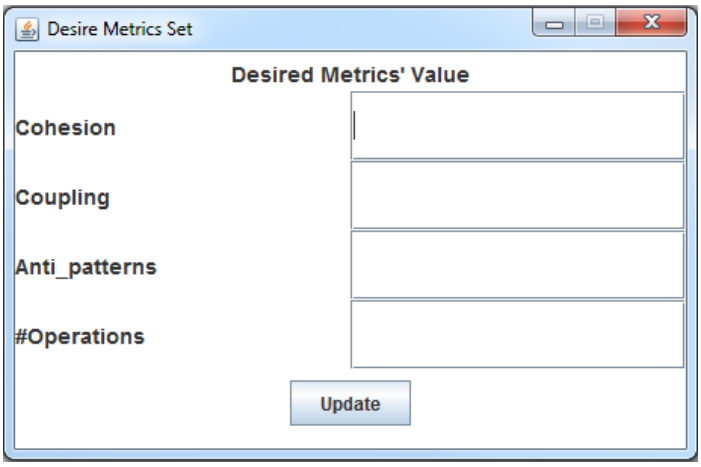

Fig. 5. The user can specify some desired metrics value

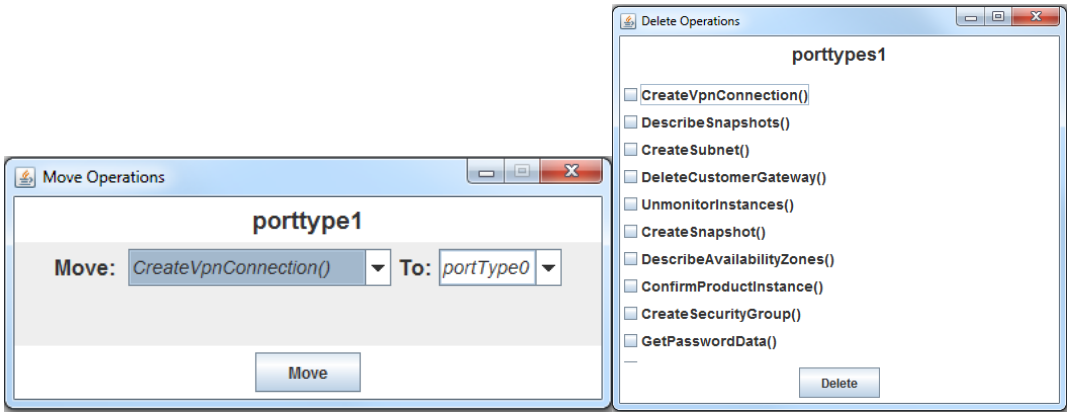

Fig. 6. The user can move or delete an operation

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 in order to have a more efficient search process. For mutation, we use the bit-string mutation operator that picks probabilistically one or more refactoring 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 portTypes.

CHAPTER 4: VALIDATION

To validate the ability of our interactive interface modularization framework to generate a good design quality, we conducted a set of experiments based on 22 real-world web services. 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.

4.1 Research Questions and Evaluation Metrics

We defined three research questions that address the applicability, performance in comparison to existing fully-automated interface modularization approaches [4][5] , and the usefulness of our interactive multi-objective approach. The three research questions are as follows:

RQ1: To what extent can our approach recommend relevant interface design improvements to the users?

RQ2: How does our interactive formulation perform compared to fully-automated Web services interface restructuring techniques [5]?

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

Table 1. Studied Web service interfaces

Service interface	Provider	#operations
i1. AutoScalingPortType	Amazon	13
i2. MechanicalTurkRequesterPortType	Amazon	27
i3. AmazonFPSPorttype	Amazon	27
i4. AmazonRDSv2PortType	Amazon	23
i5. AmazonVPCPortType	Amazon	21
i6. AmazonFWSInboundPortType	Amazon	18
i7. AmazonS3	Amazon	16
i8. AmazonSNSPortType	Amazon	13
i9. ElasticLoadBalancingPortType	Amazon	13
i10. MessageQueue	Amazon	13
i11. AmazonEC2PortType	Amazon	87
i12. KeywordService	Yahoo	34
i13. AdGroupService	Yahoo	28
i14. UserManagementService	Yahoo	28
i15. TargetingService	Yahoo	23
i16. AccountService	Yahoo	20
i17. AdService	Yahoo	20
i18. CompaignService	Yahoo	19
i19. BasicReportService	Yahoo	12

i20. TargetingConverterService	Yahoo	12
i21. ExcludedWordsService	Yahoo	10
i22. GeographicalDictionaryService	Yahoo	10

4.2 Experimental Setting

Parameter setting influences significantly the performance of a search algorithm on a particular problem. The stopping criterion was set to 10,000 evaluations for all algorithms in order to ensure fairness of comparison. The other parameters' values were fixed by trial and error and are as follows: crossover probability = 0.6; mutation probability = 0.4 where the probability of gene modification is 0.2; population size = 50. Regarding the evaluation of fixed interface design antipatterns, we focus on those defined in chapter 2.

Our study involved 19 participants from the University of Michigan to use and evaluate our tool. Participants include 11 master students in Software Engineering, 8 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 2 to 19 years. 7 out of the 19 participants are currently active programmers as well in software industry with a minimum experience of 2 years.

Participants were first asked to fill out a pre-study questionnaire containing four questions. The questionnaire helped to collect background information such as their role within the company, their programming experience, their familiarity with Web services. In addition, all the participants attended one lecture about Web services design quality, modularization and passed five tests to evaluate their performance to evaluate and suggest interface design modularization solutions.

4.3 Results

Results for RQ1. As described in Figures 7 and 8, we found that a considerable number of proposed portTypes, with an average of more than 80% in terms of precision and recall on all the 22 Web services, were already suggested manually (expected refactorings) by the users (software development team). The recall scores are slightly higher, in average, than precision ones since we found that the portTypes suggested manually by developers are incomplete compared to the solutions provided by our approach. In addition, we found that the slight deviation with the expected portTypes is not related to incorrect ones but to the fact that different possible modularization solutions could be correct.

We evaluated also the ability of our approach to fix several types of interface design antipatterns and to improve the quality as described in Figure 8 depicts the percentage of fixed code smells (*NF*). It is higher than 79% on all the 22 Web services, which is an acceptable score since users may not be interested to fix all the antipatterns in the interface. Some Web services, such as AmazonSNSPortType, has a higher percentage of antipatterns with an average of more than 86%. This can be explained by the fact that this Web service interface includes a low number of antipatterns than others.

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

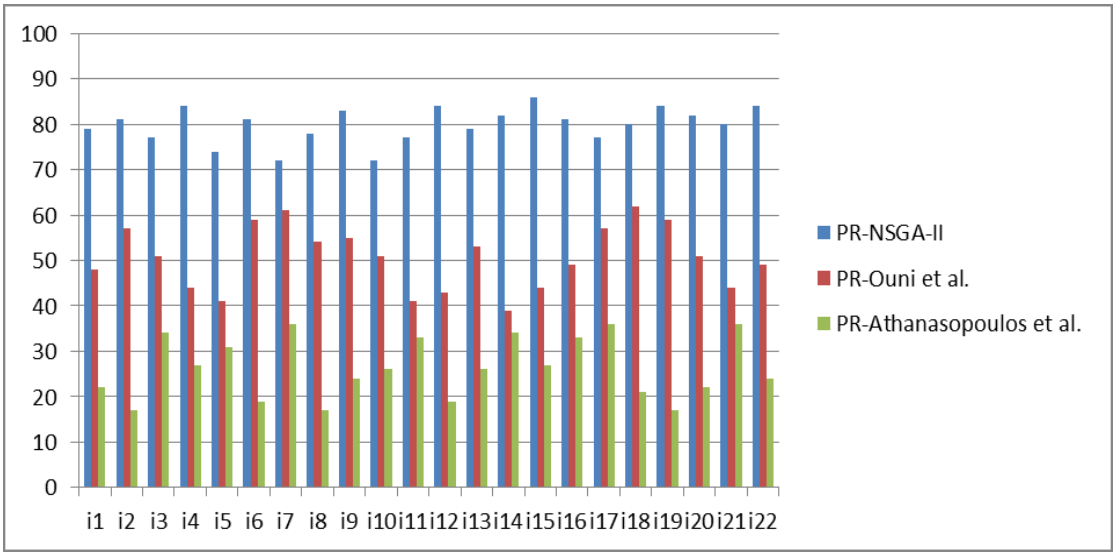

Fig. 7. The precision (PR) results on all the 22 Web services.

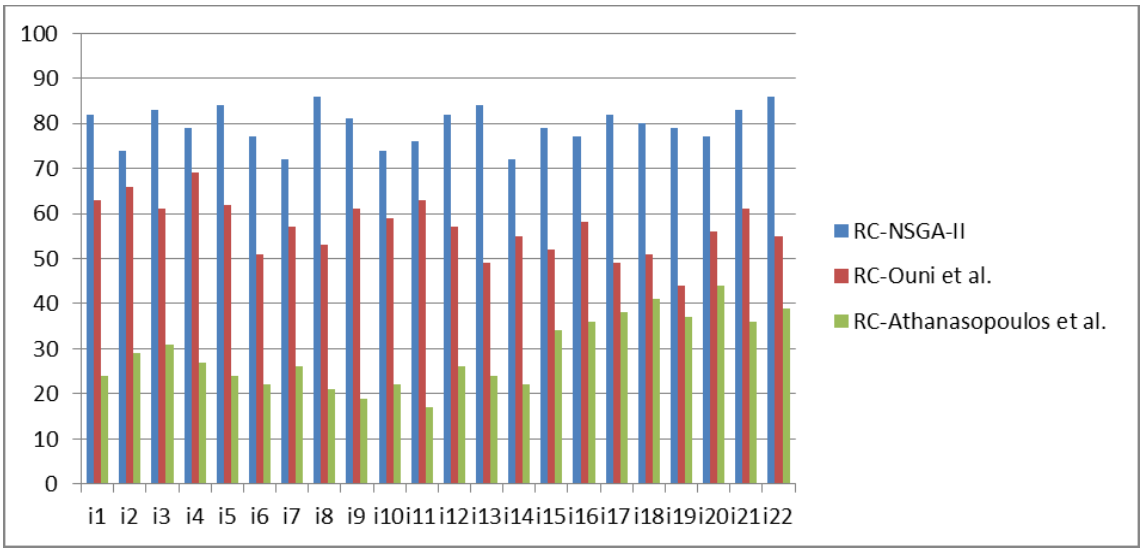

Fig. 8. The recall (RE) results on all the 22 Web services.

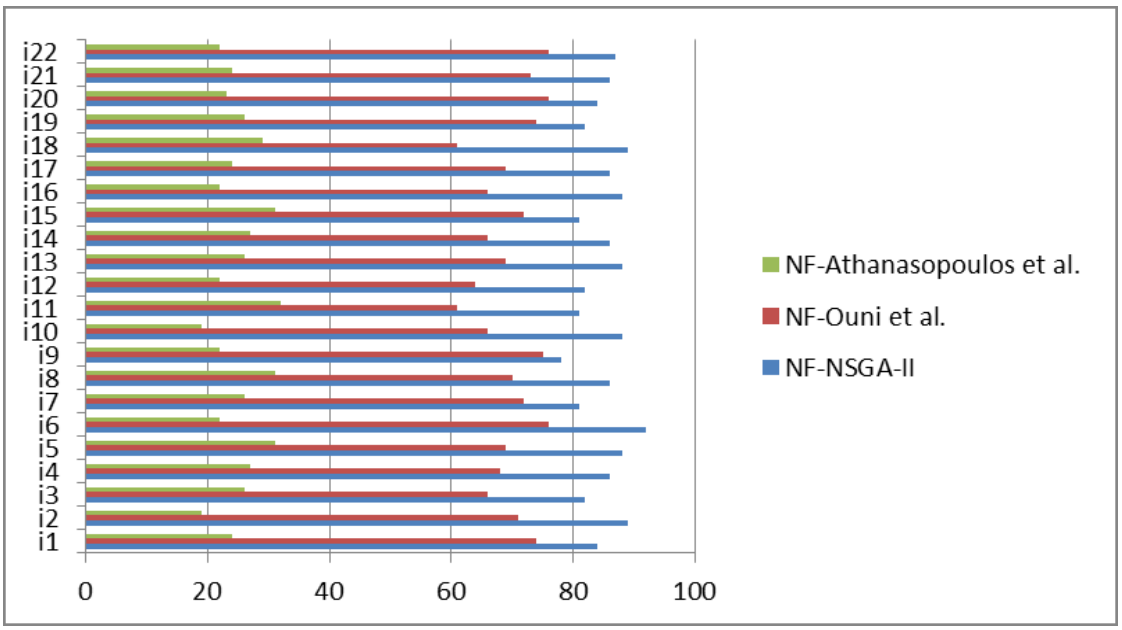

Fig. 9. The number of fixed design antipatterns (NF) results on all the 22 Web services.

Results for RQ2. Figures 7,8 and 9 confirm the average superior performance of our interactive approach compared to the two existing fully-automated interface design decomposition techniques [4][5].

Figure 7 and 8 show that our approach provides significantly higher precision and recall than all other approaches having *PR* and *RC* scores respectively between 17% (minimum) and 74% (maximum), on average on the different Web services. The same observation is valid for the number of fixed antipatterns (*NF*). This is can be explained by the reason that the main goal of these existing approaches to improve only the cohesion metric. In addition, our approach is based on a multi-objective algorithm to find a trade-off between different objectives including the correction of antipatterns.

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

Results for RQ3. 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:

- The interactive dynamic interface modularization recommendations are a desirable feature to improve the quality of Web services interface.
- The interactive manner of recommending modularization solutions by our approach is a useful and flexible way to consider the user perspective compared to fully-automated tools.

The agreement of the participants was 4.9 and 4.6 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 also 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 faster than the manual restructuring of the interface since they spent a long time with manual changes to create portTypes and move operations. Thus, the developers liked the functionality of our tool that helps them to modify a portType based on the recommendations.

Another important feature that the participants mention is that our interactive approach allows them to take the advantages of using multi-objective optimization without the need to learn anything about optimization and exploring explicitly the Pareto front to select one "ideal" solution. The implicit exploration of the Pareto front in an interactive fashion represents an important advantage of our tool along with the dynamic update of the recommended design. 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.

CHAPTER 5: CONCLUSION

We proposed, in this thesis, an interactive recommendation tool for Web services interface design modularization that dynamically adapts and suggests design changes to developers based on their feedback and three objective functions. Our interactive approach allows users to benefit from search-based tools without explicitly involving any knowledge about optimization and multi-objective optimization algorithms. In fact, the exploration of the non-dominated refactoring solutions is implicitly performed based on the interaction with the users. The feedback received from the users is used to reduce the search space and converge to better design modularization solutions.

To evaluate the effectiveness of our tool, we conducted a human study on a set of users (software developers) who evaluated the tool and compared it with the state-of-the-art interface design modularization techniques. Our evaluation results provide strong evidence that our tool improves the applicability of interface modularization techniques, and proposes a novel way for software developers to refactor their interfaces design interactively.

Future work involves validating our technique with additional interfaces and APIs in order to conclude about the general applicability of our methodology. Furthermore, we only focused, in this work, on the recommendation of interface design changes. We plan to extend the approach by

considering multiple service interfaces instead of one interface for the purpose of services composition. In addition, we will consider the importance of interface antipatterns during the correction step using previous invocations, interface complexity, etc.

REFERENCES

1. 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.

2. 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. ACM Transactions on Embedded Computing Systems, Vol. V, No. N, Article A, Publication date: January YYYY. A:16

3. 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.

4. 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.

5. 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.

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

7. 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.

8. D. Romano and M. Pinzger, "A genetic algorithm to find the adequate granularity for service interfaces," in Services (SERVICES), 2014 IEEE World Congress on. IEEE, 2014, pp. 478–485.

9. R. Haesen, M. Snoeck,W. Lemahieu, and S. Poelmans, "On the definition of service granularity and its architectural impact," in Advanced Information Systems Engineering. Springer, 2008, pp. 375–389.

10. M. Perepletchikov, C. Ryan, K. Frampton, and H. Schmidt, "Formalising service-oriented design," Journal of software, vol. 3, no. 2, pp. 1–14, 2008.

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

12. H. Wang, A. Ouni, M. Kessentini, B. R. 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, San Francisco, CA, USA, June 27 - July 2, 2016, 2016, pp. 586–593.

13. H. Masoud and S. Jalili, "A clustering-based model for class responsibility assignment problem in object-oriented analysis," Journal of Systems and Software, vol. 93, pp. 110–131, 2014.

14. W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software remodularization using nsga-iii," ACM Trans. Softw. Eng. Methodol., vol. 24, no. 3, pp. 17:1–17:45, May 2015.

15. K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," Evolutionary Computation, IEEE Transactions on, vol. 6, no. 2, pp. 182–197, 2002.

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

17. D. Athanasopoulos and A. Zarras, "Fine-grained metrics of cohesion lack for service interfaces," in IEEE International Conference on Web Services (ICWS), July 2011, pp. 588–595.

18. K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," IEEE Transactions on Software Engineering, vol. 37, no. 2, pp. 264–282, March 2011.

19. E. R. Hruschka, R. J. Campello, A. Freitas, A. C. De Carvalho et al., "A survey of evolutionary algorithms for clustering," Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on, vol. 39, no. 2, pp. 133–155, 2009.

20. 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.

21. 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.

22. N. Anquetil and T. C. Lethbridge, "Experiments with clustering as a software remodularization method," in 6th Working Conference on Reverse Engineering. IEEE, 1999, pp. 235–255.

23. O. Maqbool and H. A. Babri, “Hierarchical clustering for software architecture recovery,” IEEE Transactions on Software Engineering, vol. 33, no. 11, pp. 759–780, 2007.

24. S. Mancoridis, B. S. Mitchell, C. Rorres, Y.-F. Chen, and E. R. Gansner, “Using automatic clustering to produce high-level system organizations of source code.” in IWPC, vol. 98. Citeseer, 1998, pp. 45–52.

25. M. Harman, R. M. Hierons, and M. Proctor, “A new representation and crossover operator for search-based optimization of software modularization.” in GECCO, vol. 2, 2002, pp. 1351–1358.

26. O. Seng, M. Bauer, M. Biehl, and G. Pache, “Search-based improvement of subsystem decompositions,” in Proceedings of the 7th annual conference on Genetic and evolutionary computation. ACM, 2005, pp. 1045–1051.

27. H. Abdeen, S. Ducasse, H. Sahraoui, and I. Alloui, “Automatic package coupling and cycle minimization,” in 16th Working Conference on Reverse Engineering. IEEE, 2009, pp. 103–112

28. Kalboussi S, Bechikh S, Kessentini M, Said LB. Preference-based many-objective evolutionary testing generates harder test cases for autonomous agents. In International Symposium on Search Based Software Engineering 2013 Aug 24 (pp. 245-250). Springer, Berlin, Heidelberg.

29. Ouni, Ali, Marouane Kessentini, and Houari Sahraoui. "Search-based refactoring using recorded code changes." In Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on, pp. 221-230. IEEE, 2013.

30. Bechikh, Slim, Marouane Kessentini, Lamjed Ben Said, and Khaled Ghédira. "Chapter four-preference incorporation in evolutionary multiobjective optimization: A survey of the state-of-the-art." Advances in Computers 98 (2015): 141-207.

31. Boussaa, Mohamed, Wael Kessentini, Marouane Kessentini, Slim Bechikh, and Soukeina Ben Chikha. "Competitive coevolutionary code-smells detection." In International Symposium on Search Based Software Engineering, pp. 50-65. Springer, Berlin, Heidelberg, 2013.

32. Kessentini, Marouane, Houari Sahraoui, Mounir Boukadoum, and Manuel Wimmer. "Search-based design defects detection by example." In International Conference on Fundamental Approaches to Software Engineering, pp. 401-415. Springer, Berlin, Heidelberg, 2011.

33. Kessentini, Marouane, Arbi Bouchoucha, Houari Sahraoui, and Mounir Boukadoum. "Example-based sequence diagrams to colored petri nets transformation using heuristic Search." Modelling Foundations and Applications (2010): 156-172.

34. Kessentini, Marouane, Philip Langer, and Manuel Wimmer. "Searching models, modeling search: On the synergies of SBSE and MDE." In Proceedings of the 1st International Workshop

on Combining Modelling and Search-Based Software Engineering, pp. 51-54. IEEE Press, 2013.

35. Kessentini, Marouane, Manuel Wimmer, Houari Sahraoui, and Mounir Boukadoum. "Generating transformation rules from examples for behavioral models." In Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications, p. 2. ACM, 2010.