Interactive Refactoring of Web Service Interfaces Using Computational Search

Hanzhang Wang, Marouane Kessentini, and Ali Ouni

Abstract— 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 aggregating many non-cohesive and semantically unrelated operations. Thus, the service interface becomes unnecessarily complex for users to find relevant operations to be used by their services-based systems. In this paper, we propose an interactive recommendation approach, based on evolutionary algorithms, that dynamically adapts and suggests a possible remodularization 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 Non-dominated Sorting Genetic Algorithm (NSGA-II), to find a set of good design interface modularization solutions. These solutions provide a trade-off between improving several interface design quality metrics (e.g. coupling, cohesion, number of port types, and number of antipatterns) and fix Web services design antipatterns, maximizing the satisfaction of 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 in terms of generating well-designed Web services interface for users.

Index Terms— Web services, design, quality, user interface.

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 guaranteeing 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]. Web services interfaces could be provided by different service providers such as FedEx, Google, PayPal and Google, and represent 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 concatenating many non-cohesive operations that are semantically unrelated, and thus make it unnecessarily complex for users to find relevant operations to be used in their services-based systems. An example of well-known interface design antipattern is the 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 high unavailability to end users because it is overloaded. Indeed, the choice of how operations should be exposed through a service interface can have an impact on the performance, popularity and reusability of the service [7] and it is not a trivial task. On one hand, Web services interface exposing a high number of operations allow their clients to invoke their interfaces many times which significantly deteriorate the service performance. On the other hand, aggregating several operations of an interface into one large operation will reduce the reusability of the service.

Despite its importance, very few studies focused on improving the design of Web service interfaces for the users/subscribers [4][5]. The majority of existing work [3][4][5][6][7] addressed the problem of the detection of design antipatterns 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 antipattern using combinations of mainly quantitative metrics. For each possible interface design antipattern, 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 antipatterns. In fact, the identification of these interface design antipatterns is ultimately a 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.

Recent work [4][5] addressed the problem of fixing these design antipatterns by automatically decomposing Web services interface based only on the cohesion metric. Indeed, 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 Web service interface could be important to understand the dependency between the operations and generate a well-designed interface [1]. However, these aspects related to the users' feedback, when improving the quality of services interface, were not considered by existing studies.

In this paper, we propose a recommendation approach that dynamically adapts and interactively suggests a possible modularization, also called refactoring [15], of the Web services interface to developers and takes their feedback into consideration. Our approach uses an interactive multi-criteria decision-making

- *Hanzhang Wang and Marouane Kessentini are with the Computer and Information Science Department, University of Michigan, MI, 48126, USA. E-mail: wanghanz@umich.edu, marouane@umich.edu.*
- *Ali Ouni is with the Department of Computer Science, ETS, QC, Canada. E-mail: ali.ouni@etsmtl.ca*

xxxx-xxxx/0x/$xx.00 © 200x IEEE Published by the IEEE Computer Society

algorithm, based on interactive non-dominated sorting genetic algorithm (NSGA-II) [14], to find a set of good design interface modularization solutions that provide a trade-off between (1) improving several interface design quality metrics (e.g. coupling, cohesion, number of portTypes and number of antipatterns), (2) maximizing the satisfaction of the interaction constraints learnt from the user feedback during the execution of the algorithm, while (3) minimizing the deviation from the initial design. To find a trade-off between these different conflicting objectives, there is no single possible modularization solution but a set of optimal, i.e., non-dominated, solutions, so-called Pareto front [14]. The challenge at this step is how to choose one solution from this front to present to the Web service's user or developer? The traditional approach is to seek a 'knee point' [14] from the front that presents the maximum trade-off between the different objectives. However, this may ignore the preferences of the user. To address this issue, we propose to analyze and explore the Pareto front of possible remodularization solutions interactively and implicitly with the developer.

Our algorithm starts by finding the most frequently-occurring remodularization operations among the set of non-dominated solutions. Based on this analysis, a complete interface remodularization solution is chosen from the front that best matches the most frequently-occurring operations, i.e., the solution that best represents the entire front. The recommended modularization operations are then ranked and suggested to the developer one by one. The developer can approve, modify or reject each suggested modularization such as moving operations between port types, or merging/splitting port types. Each action by the developer participates to guide the search process towards a desired solution. For example, if the user rejects to apply a modularization operation, the search process will subsequently avoid to reconsider it when creating new solutions. NSGA-II will continue to execute in the new modified context to repair and evolve the set of good remodularization solutions based on the feedback received from the Web services developer.

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 paper can be summarized as follows:

1. The paper introduces a novel interactive way to modularize and improve the quality of Web services using interactive dynamic multi-objective optimization. The proposed technique supports the adaptation of interface design solutions based on the user feedback while 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 modularized Web services interface.
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 a benchmark of 22 real-world services. The paper also evaluates the relevance and usefulness of the suggested interface design improvements for Web service users.

The remainder of this paper is as follows: Section 2 presents the relevant background and a motivating example for the presented work; Section 3 describes the search algorithm; an evaluation of the algorithm is explained and its results are discussed in Section 4; Section 5 is dedicated to related work. Finally, concluding remarks and future work are provided in Section 6.

2 Background and Challenges

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 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][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 antipatterns are defined as bad design choices that can have a negative impact on the interface quality such as maintainability, changeability, and comprehensibility which may impact 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 antipatterns [3][7]. In our experiments, we focus on the seven following Web service antipattern 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.

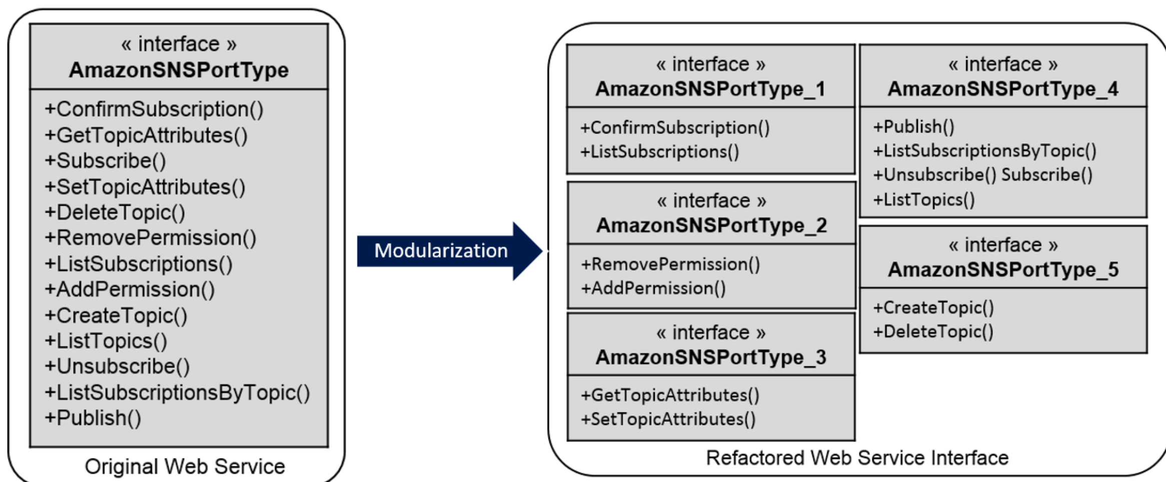


Fig. 1. Restructuring the design of a Web service Interface example (Amazon Simple Notification Service)

– *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 antipattern types in our interactive interface design tool because they are the most frequent and hard to detect [18], cover different interface design issues, due to the availability of antipattern examples and could be detected using a tool proposed in our previous work [3][12]. Our approach supports high-level refactorings: Interface Decomposition (to split an interface into multiple port types), Interface Merging (to merge multiple interfaces) and Move Operation (to move an operation between different interfaces). Of course, these high-level refactorings are composed by low-level ones such as delete and add operations that are also supported by our approach. In addition, the use of cosine similarity, as highlighted later in the fitness functions section, can be used to identify inconsistencies related to the name of operations. A God Object Web Service could be fixed mainly using the Interface Decomposition refactoring while Fine Grained and Chatty antipatterns could be addressed by Interface Merging. To fix redundant PortTypes, our approach uses a high-level refactoring which is Interface Merging to merge the two redundant PortTypes into one. This high-level refactoring includes, automatically, the deletion of redundant operations (as low-level operation) as part of the merging where there is a constraint that the operations within a merged PortType are not redundant. However, we also give the opportunity to the user to delete an operation manually for situations where he created manually a new portType that introduced some redundancies. Both Ambiguous and CRUDy interface can addressed using a combination of Move Operation and Interface Decomposition guided mainly by the fitness function including cosine similarity to distribute the behavior or remove ambiguities.

2.2 Problem Statement

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 have 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 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 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 antipatterns rather than the modularity of the interface [4][5]. 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 antipattern. Another issue is related to the definition of thresholds when dealing with quantitative information. For example, the *GOWS* antipattern 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.

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 port type 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.

The main motivations to refactor Web services are not just to fix antipatterns. Developers want to take control of changes introduced to the interface since they are not interested to fix all possible antipatterns but they may have preferences to improve some quality metrics than others. Thus, it is important to consider the developer in the loop when refactoring services, not because it is impossible to fix antipatterns automatically but mainly due to the fact that 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 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 should be considered such as coupling, number of portTypes, cohesion, number of design antipatterns, etc. Thus, it is critical to find a trade-off between these different metrics based on the preferences of the user as discussed by Coscia et al. [24] . 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.

In this paper, we propose 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 remodularizations. The next section describes the proposed interactive Web services design restructuring technique.

3 Interactive Search Algorithm For the Remodularization of Web Services

In this section, we first detail some required background information to understand the technique proposed in this paper, then we present an overview of our approach and finally we provide the details of our problem formulation and the solution approach.

3.1 Interactive and Dynamic Evolutionary Multi-Objective Optimization

In this section, we give a brief overview about two important aspects in the Evolutionary Multi-Objective Optimization (EMO) paradigm related to the: (1) Interaction with the user and (2) Dynamicity of the problem.

Interacting with the human user means allowing the user to inject his/her preferences into the computational search algorithm and then using these preferences to guide the search process. In most of existing studies [13][14], the user's preferences are expressed and handled in the objective space. It is important to highlight that one of the original aspects of our work in this paper, as detailed later, is to allow the user to express his preferences in the decision space and then handling these preferences to help the user finding the most desired refactoring solution. Moreover, our approach helps the user in eliciting his preferences, which is important for preference-based EMO algorithm. These preferences are introduced implicitly by moving between the Pareto front of non-dominated solutions after obtaining feedback from the user about just few parts of the solution to better understand his preferences. This implicit exploration of the Pareto front will be detailed in the next section.

The integration of user preferences is challenging due the changes introduced to some of the generated solutions based on the interaction feedback. Applying evolutionary algorithms (EAs) to solve Dynamic Multi-Objective Problems (DMOPs) has received great attention from researchers based on the adaptive behavior of evolutionary computation methods. A DMOP consists of minimizing or maximizing an objective function vector under some constraints over time. Its general form is the following [14]:

$$\begin{cases} Min\ f(x,t) = [f_1(x,t), f_2(x,t), ..., f_M(x,t)]^T & \\ g_j(x,t) \geq 0 & j = 1,...,P; \\ h_k(x,t) = 0 & k = 1,...,Q; \\ x_i^L \leq x_i \leq x_i^U & i = 1,...,n. \end{cases}$$

where M is the number of objective functions, t is the time instant, P is the number of inequality constraints, Q is the number of equality constraints, x_i^L and x_i^U correspond respectively to the lower and upper bounds of the variable x_i.

A solution x_i satisfying the (P+Q) constraints is said to be *feasible*, and the set of all feasible solutions defines the feasible search space denoted by Ω. The resolution of an 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 *PF*. Hence, the resolution of a MOP consists in approximating the entire PF. In the following, we provide some background definitions related to multi-objective optimization. These definitions remain valid for the case of DMOPs.

Definition 1: Pareto optimality

A solution $x^* \in \Omega$ is Pareto optimal if $\forall\, x \in \Omega$ and $I = \{1,...,M\}$ either $\forall\ m \in I$ we have $f_m(x) = f_m(x^*)$ or there is at least one $m \in I$ such that $f_m(x) > f_m(x^*)$.

The definition of Pareto optimality states that x^* is Pareto optimal if no feasible vector x exists that would improve some objectives without causing a simultaneous worsening in at least one other objective.

Definition 2: Pareto dominance

A solution $u = (u_1, u_2, ..., u_n)$ is said to dominate another solution $v = (v_1, v_2, ..., v_n)$ (denoted by $f(u) \prec f(v)$) if and only if $f(u)$ is partially less than $f(v)$. In other words, $\forall\, m \in \{1,...,M\}$ we have $f_m(u) \leq f_m(v)$ and $\exists\, m \in \{1,...,M\}$ where $f_m(u) < f_m(v)$.

Definition 3: Pareto optimal set

For a given MOP $f(x)$, the Pareto optimal set is $P^* = \{ x \in \Omega \mid \neg\exists\, x' \in \Omega,\ f(x') \prec f(x) \}$.

Definition 4: Pareto optimal front

For a given MOP $f(x)$ and its Pareto optimal set P^*, the Pareto front is $PF^* = \{ f(x), x \in P^* \}$.

3.2 Approach Overview

The goal of our approach is to propose a new dynamic interactive way for users to refactor their Web services interface design. The general structure of our approach is sketched in Figure 2.

Our technique comprises two main components. The first component consists of an offline phase. It is executed first in the background when the developer uploads the WSDL file to analyze. During this phase, the multi-objective algorithm, NSGA-II [14], is executed for several iterations to find the non-dominated solutions balancing the three following objectives:

- *Objective 1* maximizes the interface design quality, which corresponds to minimize the number of design antipatterns and improve design quality metrics (coupling and cohesion),
- *Objective 2* maximizes the satisfaction of the constraints

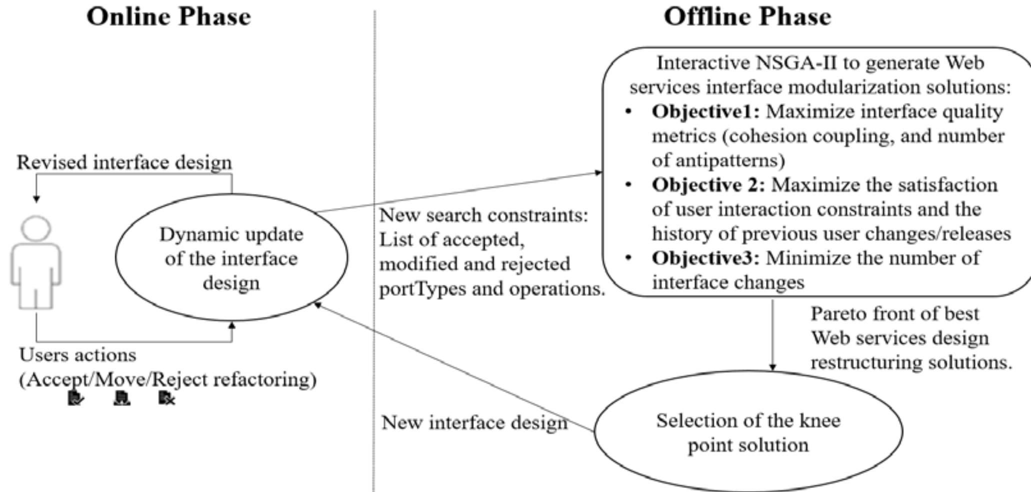


Fig. 2. Approach overview

learnt from the user interaction,

- *Objective 3* minimizes the number of introduced changes to modify the Web service design and port types.

The output of this first step of the offline phase is a set of Web services remodularization solutions that optimize the above three objectives. As explained in Algorithms 1 and 2, the second step of the offline phase explores this Pareto front in an intelligent manner using our algorithm to rank recommended changes based on the common features between the non-dominated solutions. In our adaptation, we assume true the hypothesis that the most frequently occurring remodularization operations in the non-dominated solutions are the most relevant ones for developers and can fix several antipattern types. Thus, the output of this second step of the offline phase is a set of ranked solutions based on this frequency score.

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 accept/apply or modify or reject some of the suggested design changes. Thus, the goal is to guide, *implicitly*, the exploration of the search space of possible Web services 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 accepts a port type then the operations of that port type should stay together in the next interactions of the algorithm but new operations could be moved to that port type. 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. The interaction algorithm (Algorithm 2) will be explained later in Section 3.3.4 in more details.

After several interactions, users may have modified or rejected a high number of suggested design changes or have introduced several new changes manually. Whenever the users stop the Web service 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 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 the new constraints summarizing the feedback of the user. In fact, we consider the rejected port types or operations by the developer as constraints to avoid generating solutions containing similar port types in the next iterations to avoid putting together again the operations of that rejected port types 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 next iterations 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 Web service anymore. The outcome of the proposed approach that consists of the modularization of the Web service interface should have an impact on the implementation of the operations as well. In fact, the operations that are grouped together into one sub-interface may give an indication that they should be implemented within the same module. Thus, the proposed interface modularization could help the services developer to improve the cohesion and coupling of their implementation of services operation.

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

Input
Sys: Web service interface 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, ...) ← 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
$P_{t+1} \leftarrow \bigcup_{j=1}^{l-1} F_j$;
/*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);
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); /* execution of Algorithm 2 */
Threshold ← Threshold + t+1;
End If
End

Algorithm 2. The ranking procedure to manage the interactions with the developer (online phase)

Input
RNS: Ranked Non-dominated SolutionSet
Output
M: Map of refactorings along with their occurrences.
Begin
Applied-Refactorings ← Ø;
Rejected-Refactorings ← Ø;
For *i=1* **to** |*RNS*| **do**
ref[i] ← 0;
End for
/* Main loop to suggest refactorings one by one to the user*/
While |Rejected-Refactorings| < α **do**
/* Select index of the the solution with highest rank*/
index ← Max-Rank(*RNS*);
d ← User-Decision($RNS_{index,ref[index]}$);
/* If the user has applied or modified the operation*/
If (*d* = True) **then**
Applied-Refactorings ← Applied-Refactorings ∪ $RNS_{index,ref[index]}$;
/* If the user has rejected the operation*/
else
Rejected-Refactorings ← Rejected-Refactorings ∪ $RNS_{index,ref[index]}$;
End if
ref[index] ← ref[index] + 1;
/* Update solutions indexes */
For *i=1* **to** |*RNS*| **do**
Update-Rank(RNS_i; Applied-Refactorings,Rejected-Refactorings)
End While
End

3.3 Solution Approach

We describe in the following subsections the details of the various components of our framework.

3.3.1 Interactive NSGA-II

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 paper, we adapted one of the widely used multi-objective search algorithms called NSGA-II [14] and integrated our 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 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.

3.3.2 Solution Representation

PortType2	PortType1	PortType1	PortType1	PortType2
Op1	Op2	Op3	Op4	Op5

Fig. 3. Example of a solution representation

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

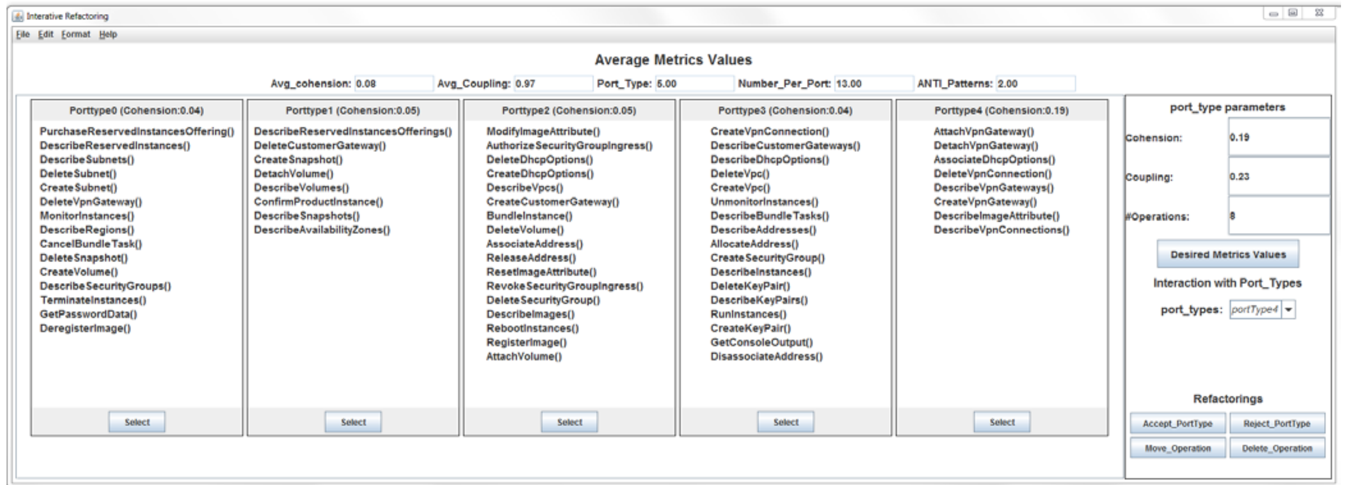


Fig. 4. The proposed Web services design modularization tool.

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. As output, a vector representation is automatically translated by our tool into a graphical interface as described in Figure 4.

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 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) performance indicator 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.

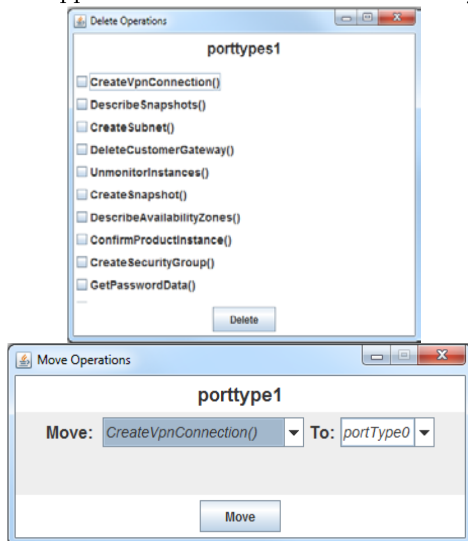


Fig. 5. The user can specify some desired metrics value

3.3.3 Fitness Functions

$$f_1 = \frac{\#antipatterns_After_Modularization}{\#antipatterns_Before_Modularization} + Cohesion - Coupling$$

Objective 1: Maximize the Web services 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 paper, the definition of cohesion defined by [4] which is based on communicational and textual similarities between the operations within the same port type 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 port type and the number of calls to other operations in different port types [5]. The reason of not treating quality objectives separately are related to redcuing the execution time and the number of non-dominated solutions (especially for an interactive approach), and also the performance of NSGA-II when the number of objectives becomes high.

Objective 2: Maximize the interaction-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 4. 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 5 and 6, that correspond to *accept a portType, reject a portType, move operation(s)* and *delete operation(s)*. Each of these user actions will generate a set of constraints for the exploration of the search space. When a port type is accepted, the list of operations in that port type should stay together in the next iterations but new operations could be added to the port type. When a port type is rejected by the user, a constraint is generated to avoid regrouping together again these operations into the same port type. The application of a move operation action will generate a constraint to keep the

moved operation in the targeted port type in the next iterations. When an operation is deleted, a constraint will be generated to avoid putting again that operation in the source port type in the next iterations. Formally, the second fitness function to minimize is defined as follows:

$$f_2 = \frac{1}{\sum_{i=1}^{k} |MDesired_i - M_i|} + \frac{\#Statisfied_Histroy_Constraints}{\#Constraints}$$

This second fitness function is composed by two components. The first component is to minimize the distance between the desired metrics value specified by the user (e.g. coupling, cohesion, number of portTypes, etc.) and the actual values of the solution to evaluate. The second component is to maximize the number of satisfied interaction constraints over the total number of learnt constraints.

Objective 3: Minimize 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 define the fitness function as the following:

$$f_3 = \#designChanges$$

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 port types compared to the initial design.

3.3.4 Interactive Recommendations

The first step of the interactive component is executed as described in Algorithm 2, to investigate if there are some common patterns among the generated non-dominated refactoring solutions. The algorithm checks if the optimal refactoring solutions have some common features such as similar refactoring operations among most or all the solutions, and a specific common order/sequence in which to apply the refactorings. Such information will be used to rank the suggested refactorings for developers using the following formula:

$$Rank(R_{x,y}) = \frac{\sum_{j=0}^{n} \sum_{i=0}^{size(Sj)} [Ri,j = Rx,y]}{MAX(\sum_{j=0}^{n} \sum_{i=0}^{size(Sj)} [Ri,j = Rx,y])} \in [0..1]$$

where $R_{x,y}$ is the refactoring operation number x (index in the solution vector) of solution number y, and n is the number of solutions in the front. S_i is the solution of index i. All the solutions of the Pareto front are ranked based on the score of this measure applied to every solution.

Once all Pareto front solutions are ranked, the second step of the interactive process is executed as described in Algorithm 3. The refactorings of the best solution, in terms of ranking, are recommended to the developer based on their order in the vector. Then, the ranking score of the solutions is updated automatically after every feedback (interaction) with the developer. Our interactive algorithm proposes three levels of interaction as described in Figure 2. The developer can check the ranked list of refactorings and then *apply*, *modify* or *reject* the refactoring. If the developer prefers to modify the refactoring, then our algorithm can help them during the modification process as described in Figures 5 and 6. In fact, our tool proposes to the developer a set of recommendations to modify the refactoring based on the history of changes applied in the past and the semantic similarity between the port types and operations. For example, if the developer wants to modify a move operation refactoring then, having specified the source port type to move, our interactive algorithm automatically suggests a list of possible target port types ranked based on the history of changes and semantic similarity. This is an interesting feature of our approach since developers often know which operation to move, but find it hard to determine a suitable target port type [12]. The same observation is valid for the remaining refactoring types. Another action that the developers can select is to reject/delete a refactoring from the list. After every action selected by the developer, the ranking is updated based on the feedback using the following formula:

$$Rank(S_i) = \sum_{k=1} Rank(R_{k,i}) + (RO \cap AppliedRefactoringsList) \\ -(RO \cap RejectedRefactoringsList) + 0.5 * (RO \cap ModifiedRefactoringsList)$$

Where S_i is the solution to be ranked, the first component consists of the sum of the ranks of its operations as explained previously and the second component will take the value of 1 if the recommended refactoring operation was applied by the developer, or -1 if the refactoring operation was rejected or 0.5 if it was partially modified by the developer. We selected 0.5 as a threshold since most of the operations have very few parameters (up-to two parameters) that could be modified. The recommended refactorings will be adjusted based on the updated ranking score.

It is important to note that we calculate the ranking score for each non-dominated solution using our ranking measure and then the solution with the highest score is presented refactoring by refactoring to the developer. In fact, refactorings tend to be dependent on one another, thus it is important to ensure the coherence of the recommended solution. After several modified or rejected refactorings, the generated Pareto front of refactoring solutions by NSGA-II needs to be updated since the original interface was modified. Thus, the ranking of the solutions will change after every interaction. If many refactorings are rejected, the NSGA-II algorithm will continue to execute while taking into consideration all the feedback from developers as constraints to satisfy during the search. The rejected refactorings should not be considered as part of the newly generated solutions and the new Web service interfaces after refactoring will be considered in the input of the next iteration of the NSGA-II.

In a non-interactive Web services refactoring approach, the set of refactorings, suggested by the best-chosen solution, needs to be fully executed to reach the solution's promised results. Thus, any changes applied to the set of refactorings such as changing or skipping some of them could deteriorate the resulting design quality. In this context, the goal of this work is to cope with the above-mentioned limitation by granting to the developer's the possibility to customize the set of suggested refactorings either by accepting, modifying or rejecting them. The novelty of this work is the approach's ability to consider the developer's interaction, in terms of introduced customization to the existing solution, by conducting a local search to locate a new solution in the Pareto Front that is nearest to the newly introduced changes. We believe that our approach may narrow the gap that exists between automated and manual Web services refactoring techniques. It allows the developer to select the refactorings that best match his/her design preferences.

3.3.5 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. We considered the widely used changes operator adaptation used for discrete problems [26]. 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 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 port types.

4 VALIDATION

To evaluate the ability of our interactive Web services modularization framework to generate a good design quality, we conducted a set of experiments based on 22 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.

4.1 Research Questions and Evaluation Metrics

We defined three research questions that address the applicability, performance in comparison to existing fully-automated Web services 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 Web services design improvements?

RQ2: How does our interactive formulation perform compared to fully-automated Web services restructuring techniques [4][5]?

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 after interactions with the developers as described in the previous section. 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:

$$\text{PR}precision = \frac{\text{suggested_portTypes} \cap \text{expected_portTypes}}{\text{suggested_portTypes}} \in [0,1]$$

$$\text{RC}recall = \frac{\text{suggested_portTypes} \cap \text{expected_portTypes}}{\text{expected_portTypes}} \in [0,1]$$

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 [12]. Formally, *NF* is defined as:

$$NF = \frac{\#\text{fixeddesingantipatterns}}{\#\text{designantipatterns}} \in [0,1]$$

For the qualitative validation, we asked groups of potential users of our Web services refactoring tool to evaluate, manually, whether the suggested interface design refactorings 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 refactorings divided by the total number of recommended refactorings by our tool. The *MC* metric is computed after the user interaction is completed. In fact, the number of correct refactorings includes the number of design refactorings applied by developers when using our tool, since they can either apply, modify or reject a refactoring recommendation (e.g. created port type). *MC* is given by the following equation:

$$MC = \frac{\#\text{coherentappliedmodularizationoperations}}{\#\text{proposedmodularizationoperations}}$$

To avoid the computation of the *MC* metric being biased by the developer's feedback, we asked the developers to manually evaluate the correctness of the recommended refactorings on the Web services that they did not refactor using our tool.

We considered also some other useful metrics to answer RQ1 that count the percentage of Web service refactorings that were accepted (*NAC*) or rejected (*NRE*) or applied with some modifications (*NMO*). Formally, these metrics are defined as:

$$NRE = \frac{\#\text{rejectedmodularizationoperations}}{\#\text{recommendedmodularizationoperations}} \in [0,1]$$

To answer RQ2, we compared our approach to two other existing fully-automated Web services decomposition techniques

$$NMO = \frac{\#\text{modifiedmodularizationoperations}}{\#\text{recommendedmodularizationoperations}} \in [0,1]$$

$$NAC = \frac{\#\text{acceptedmodularizationoperations}}{\#\text{recommendedmodularizationoperations}} \in [0,1]$$

[4][5]. Ouni et al. [5] proposed an approach to decompose Web services using graph partitioning to improve cohesion. Similarly, Athanasopoulos et al. [4] 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. Thus, we used the metrics *PR, RC,* 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 interactive tool and the automated framework proposed by Ouni et al. [5] 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. [5]. Thus, we can check whether the "online phase" 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. Then, we compared between the outcomes of the survey questions for both interactive and fully automate techniques.

4.2 Experimental Setting

We used a benchmark of 22 well-known Web services as detailed in Table 1. All studied services are widely used in different contexts and provided by Amazon and Yahoo, two major Web service providers. We selected these Web services for our validation because they range from medium to large-sized projects, which have been actively developed and changed over several years. Our study involved 24 participants from the University of Michigan to use and evaluate our tool. Participants include 16 master students in Software Engineering and 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. 11 out of the 24 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 twelve 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.

Table 1. Studied Web service interfaces

	Service interface	Provider	#operations
i1.	AutoScalingPortType	Amazon	13
i2.	MechanicalTurkRequesterPort-Type	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

As described in Table 2, we formed 4 groups. Each of the four 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 re-modularization 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. We performed a cross-validation between the groups to avoid the evaluation will be biased by the developer's feedback. Thus, the subjects within the same group evaluated only the desing obtained with the feedback of individual of other groups.

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). To evaluate the fixed Web services design antipatterns, we focus on the ones defined in Section 2. We choose these types in our experiments because they are the most frequent and hard to fix based on several studies [2][3][7]. In the second scenario, we asked the users to manually evaluate the last recommended solution by our algorithm after the interaction with the user. 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.

Table 2. Survey organization

Groups	Web Services	Algorithms / Approaches
Group 1	i1-i5	Interactive approach Ouni et al. [4] Athanasopoulos et al. [5]
Group 2	i6-i10	
Group 3	i11-i16	
Group 4	i17-i22	

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, 50, 100 and 200. The stopping criterion was set to 50,000 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 = 0.6; mutation probability = 0.3 where the probability of gene modification is 0.2; stopping criterion = 50,000 evaluations. 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 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 [4][5] 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 using the Wilcoxon rank sum test with a 95% confidence level ($\alpha < 5\%$) as detailed in the next section.

The Wilcoxon rank sum test verifies whether the results are statistically different or not; however, it does not give any idea about the difference in magnitude. To this end, we used the Vargha-Delaney A measure which is a non-parametric effect size measure. In our context, given the different performance metrics (such as *PR*, *RC*, *NF*, *MC*, etc.), the A statistic measures the probability that running an algorithm B1 (interactive NSGA-II) yields better performance than running another algorithm B2 (such as [4].). If the two algorithms are equivalent, then A = 0.5. In our experiments, we have found the following results: a) On small Web services our approach is better than all the other algorithms based on all the performance metrics with an A effect size higher than 0.91; and b) On large Web services, our approach is better than all the other algorithms with an A effect size higher than 0.84.

4.3 Results and Discussions

Results for RQ1. As described in Figures 7 and 8, we found that a considerable number of proposed port types, 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 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 9 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 lower number of antipatterns than others.

We reported the results of our empirical qualitative evaluation in Figure 6 (*MC*). As reported in Figure 6, 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, 89% 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. The highest MC score is 94% and was achieved for the Web service *GeographicalDictionary*, while the lowest score was 79% for *AmazonVPCPortType*. Thus, this finding indicates that the results are independent of the size of the Web services and the number of recommended changes to the initial design.

Since the manual correctness *MC* metric just evaluates the correctness and not the relevance of the recommended solutions, we also compared the proposed modularization changes with some expected ones defined manually by the different groups for the different Web services. Figures 7 and 8 summarize our findings. We found that a considerable number of proposed port types, with an average of more than 84% in terms of precision and recall, were already created by the users manually (expected port types). The recall scores are higher than precision ones since we found that the port types suggested manually by developers could be further decomposed, if necessary. This was confirmed by the qualitative evaluation (MC). In addition, 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.

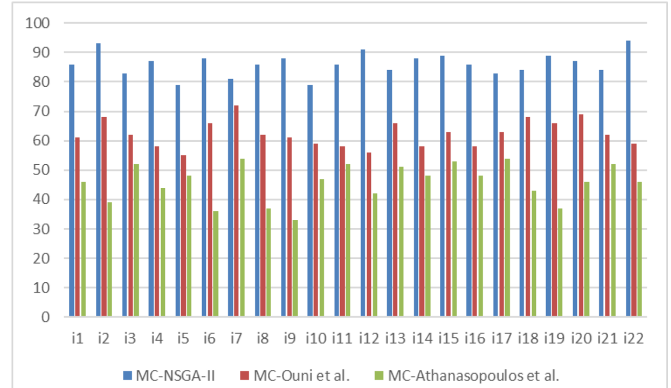


Fig. 6. Median manual correctness (MC) value over 30 runs on all 22 Web services using the different modularization techniques with a 95% confidence level ($\alpha < 5\%$).

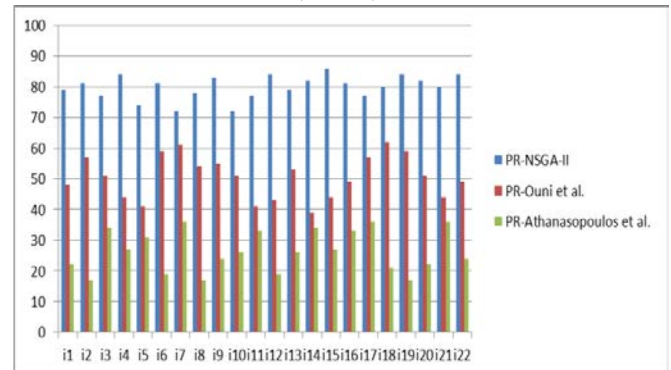


Fig. 7. Median precision (PR) value over 30 runs on all 22 Web services using the different modularization techniques with a 95% confidence level ($\alpha < 5\%$).

We evaluated also the ability of our approach to fix several types of design antipatterns and to improve the service interface design quality as described in Figure 9 that depicts the percentage of fixed antipatterns (*NF*). It is higher than 83% on all the 22 Web services, which is an acceptable score since developers may reject or modify some design changes that fix some antipatterns because they do not consider some of them as very important (their goal is not to fix all design antipatterns in the Web service interface) or because they wanted to focus on improving the cohesion and minimize coupling. Some Web service interfaces, such as *AmazonFWSInboundPortType*, have a higher percentage of fixed code smells with an average of more than 90%. This can be explained by the fact that these Web services include a higher number of design antipatterns than others. We have also considered three other evaluation metrics *NMO* (percentage of modified portTypes), *NRE* (percentage of rejected portTypes) and *NAC* (percentage of accepted portTypes) to evaluate the efficiency of our

interactive approach. We collected this data using a feature that we implemented in our tool to record all the actions performed by the developers during the remodularization sessions. Figure 10 shows that, on average, more than 81% of the recommended portTypes were accepted by the developers. In addition, an average of 9% of the recommended refactorings were modified by the developers, while 11% of the suggested refactorings were rejected by the developers. Thus, our recommendation tool successfully suggested a good set of design changes to apply.

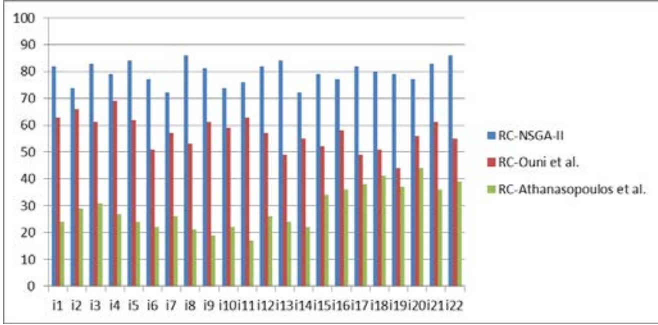


Fig. 8. Median recall (RE) value over 30 runs on all 22 Web services using the different modularization techniques with a 95% confidence level (α < 5%).

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 and improve the quality of all the 22 Web services.

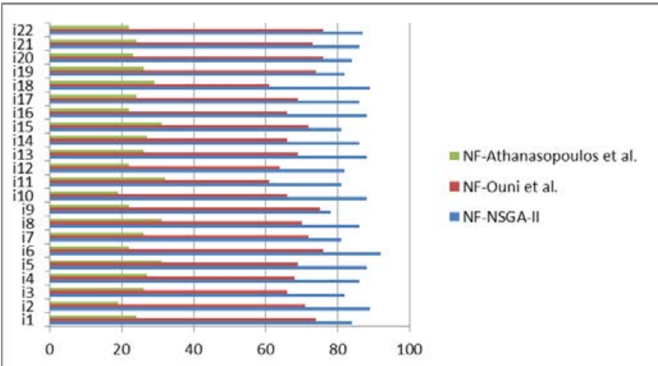


Fig. 9. Median number of fixed Web service antipatterns (*NF*) value over 30 runs on all 22 Web services using the different modularization techniques with a 95% confidence level (α < 5%).

Results for RQ2. Figures 6,7,8 and 9 confirm the average superior performance of our interactive approach compared to the two existing fully automated Web service modularization techniques [4][5]. Figure 6 shows that our approach provides significantly higher manual correctness results (*MC*) than all other approaches having *MC* scores respectively between 48% and 61%, on average as MC scores on the different Web services. The same observation is valid for the precision and recall as described in Figures 8 and 9. The outperformance of our technique in terms of percentage of fixed antipatterns, as described in Figure 9, can be explained by the fact that the main goal of existing studies is not to mainly fix these antipatterns (not considered in the fitness function by the work of Ouni et al. [5]).

Overall the superior performance of our interactive approach can be explained by several factors. First, existing studies [3][4][5][6]use only structural indications (quality metrics) to evaluate the remodularization solutions and thus a high number of changes may lead to a semantically incoherent Web services design. Our approach reduces the number of semantic incoherencies when suggesting refactorings and during the interaction with the developers. Second, the ranking component of our approach improved the quality of the suggested refactoring solutions by using an interactive approach as compared to a regular NSGA-II where the developers need to select one solution from the Pareto front that cannot be updated dynamically. Third, existing work are mainly limited to the cohesion metric which may not be sufficient to guide the modularization of Web services.

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

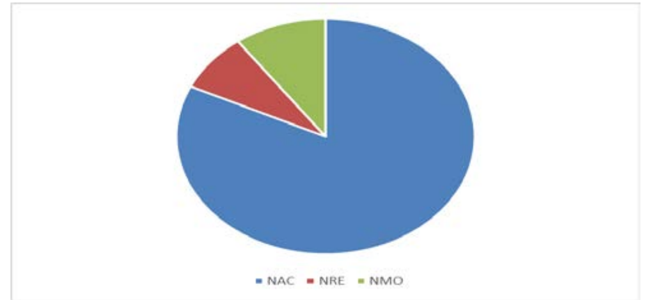


Fig. 10. Median percentage of accepted (NAC), modified(NMO) and rejected(NRE) portTypes over 30 runs on all 22 Web services using our interactive approach with a 95% confidence level

Results for RQ3. To further analyze the obtained results, we have 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: (a) The interactive dynamic interface modularization recommendations are a desirable feature to improve the quality of Web services interface. (b) 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 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.

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.

4.4 Threats to Validity

Conclusion validity is concerned with the statistical relationship between the treatment and the outcome. The parameter tuning of the different computational search algorithms used in our experiments creates another internal threat that we need to evaluate in our future work. The parameters' values used in our experiments are found by trial-and-error. However, it would be an interesting perspective to design an adaptive parameter tuning strategy for our approach so that parameters are updated during the execution to provide the best possible performance. In addition, our multi-objective formulation treats the different types of quality metrics such as coupling and cohesion with the same weight in terms of complexity when calculating one of the fitness functions. However, some quality metrics can be more important than others when evaluating a Web service design but we considered both coupling and cohesion as equally important. The same observation is valid for the different types of considered design antipatterns. Another threat is related to the use of our previous work [3] to detect antipatterns which may include few false positive. However, this threat may not have a high impact on the validity of the results since the different proposed refactorings were manually validated by the participants but some of the rejected recommendations by the developer are related to the detected antipatterns.

Construct validity is concerned with the relationship between theory and what is observed. The different developers involved in our experiments may have divergent opinions about the recommended modularizations in terms of correctness and readability. We considered in our experiments the majority of votes from the developers. For the selection threat, the participant diversity in terms of experience could affect the results of our study. We addressed the selection threat by giving a lecture and examples of Web services modularization already evaluated with arguments and justification.

5 Related Work

Web Services Design Quality: 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. [23] 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 [9]. Furthermore, Rodriguez et al. [21] 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 [17], 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. [18] have proposed an interesting approach towards generating WSDL documents with less antipatterns using text mining techniques. Coscia et al. [33] discussed the importance of finding a trade-off between several conflicting quality metrics when improving the design of Web services interface. In our previous work [12], 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[35][36][37]. However, the proposed approach can deal only with Web service interface metrics and cannot consider all Web service antipattern symptoms.

Software Remodularization: Several studies addressed the problem of clustering and remodularization of object oriented (OO) applications in terms of packages organization[27][28][29][30][31]. Harman et al. [19] used a genetic algorithm to improve subsystems decomposition by combining several quality metrics including coupling, cohesion, and complexity. Similarly, Recently, we proposed in our previous work [13] 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[32][33][34]. Praditwing et al. [16] have recently formulated the software clustering problem as a multi-objective optimization problem. Their work aim at maximizing the modularization quality measurement, minimizing the inter-package dependencies, increasing intra-package dependencies, maximizing the number of clusters having similar sizes and minimizing the number of isolated clusters. Despite these advances in OO systems modularization, still this problem is not widely explored in the context of Web service interfaces.

6 Conclusion and Future Work

We proposed, in this paper, 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.

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 paper, on the recommendation of interface design changes. We plan to extend the approach by considering multiple service interfaces instead of one interface for services composition. In addition, we will consider the importance of interface antipatterns during the correction step using previous invocations, interface complexity, etc. We are also planning to consider the different quality objectives sepretaly by adapting a many-objective optimization algorithm to support a high number of objectives.

References

[1] 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.

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

[13] 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.

[14] 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.

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

[16] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," TSE, vol. 37, no. 2, pp. 264–282, March 2011.

[17] 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.

[18] 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.

[19] 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.

[20] H. Abdeen, S. Ducasse, H. Sahraoui, and I. Alloui, "Automatic package coupling and cycle minimization," in 16th WCRE. IEEE, 2009, pp. 103–112.

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

[22] José Luis Ordiales Coscia, Cristian Mateos, Marco Crasso, and Alejandro Zunino. 2013. Anti-pattern free code-first web services for state-of-the-art Java WSDL generation tools. Int. J. Web Grid Serv. 9, 2 (May 2013), 107-126. DOI=http://dx.doi.org/10.1504/IJWGS.2013.054108

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

[24] Coscia, J. L. O., Mateos, C., Crasso, M., & Zunino, A. (2014). Refactoring code-first Web Services for early avoiding WSDL anti-patterns: Approach and comprehensive assessment. Science of Computer Programming, 89, 374-407.

[25] Noor TH, Sheng QZ, Ngu AH, Dustdar S. Analysis of web-scale cloud services. IEEE Internet Computing. 2014 Jul;18(4):55-61.

[26] Talbi, El-Ghazali. Metaheuristics: from design to implementation. Vol. 74. John Wiley & Sons, 2009

[27] Sahin, Dilan, Marouane Kessentini, Slim Bechikh, and Kalyanmoy Deb. "Code-Smell Detection as a Bilevel Problem", ACM Transactions on Software Engineering and Methodology, 2014.

[28] Mansoor, Usman, Marouane Kessentini, Manuel Wimmer, and Kalyanmoy Deb. "Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm", Software Quality Journal, 2015.

[29] Mkaouer, M.W., Kessentini, M., Bechikh, S., Deb, K., and Ó Cinnéide, M.: 'Recommendation system for software refactoring using innovization and interactive dynamic optimization'. ASE 2014 pp. 331-336

[30] 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.

[31] 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.

[32] 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.

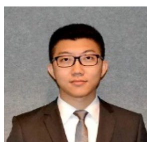
[33] Boussaa, Mohamed, Wael Kessentini, Marouane Kessentini, Slim Bechikh, and Soukeina Ben Chikha. "Competitive coevolutionary code-smells detection." In International Symposium on Search Based Software Engineering, pp. 50-65. Springer, Berlin, Heidelberg, 2013.

[34] 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.

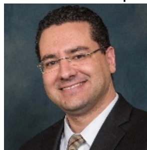
[35] Kessentini, Marouane, Arbi Bouchoucha, Houari Sahraoui, and Mounir Boukadoum. "Example-based sequence diagrams to colored petri nets transformation using heuristic Search." Modelling Foundations and Applications (2010): 156-172.

[36] 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.

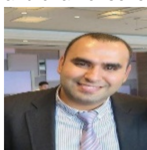
[37] Kessentini, Marouane, Manuel Wimmer, Houari Sahraoui, and Mounir Boukadoum. "Generating transformation rules from examples for behavioral models." In Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications, p. 2. ACM, 2010.



Hanzhang Wang is currently a PhD student in the Search Based Software Engineering Group at the University of Michigan. His PhD project is concerned with the application of SBSE techniques in different areas such as web services, MDE, refactoring and automotive industry. His current research interests are Search-Based Software Engineering, quality of Web services, model-driven engineering and software quality.



Marouane Kessentini is an assistant professor in Software engineering at the University of Michigan. He received his PhD in 2012 from University of Montreal. He is the founding director of the Search-based software engineering research lab in Michigan and he has several collaborations with different industrial companies on studying software engineering problems by computational search and artificial intelligence techniques. Dr. Kessentini has three best paper awards. He published over 100 papers. Dr. Kessentini served as a program committee member and chair of several conferences and associate editor in several journals.



Ali Ouni Ali Ouni is an assistant professor at the department of Software Engineering and IT at ETS Montreal, University of Quebec. He received his Ph.D. degree in computer science from University of Montreal in 2014. His research interests are in software engineering including software maintenance and evolution, refactoring of software systems, software quality, service-oriented computing, and the application of artificial intelligence techniques to software engineering. His work has received several nominations and Best Paper Awards.