Noname manuscript No.
(will be inserted by the editor)

Refactoring Web Services Interface Using Many-Objective Search

Hanzhang Wang · Ali Ouni · Marouane Kessentini · Wiem Mkaouer · Kalyanmoy Deb

Received: date / Accepted: date

Abstract Service-oriented computing (SOC) has changed the way of how software applications are designed, delivered and used. Web services is the widely used technology to implement service-oriented architectures (SOA). The Web service interface is a critical component in an SOA since it is the only source of interactions with the user to adopt the services in real-world applications. Although several studies focused on the quality of services (QoS), very few work addressed the problem of improving the design quality of Web service interface. A common bad service design practice is to group together several operations implementing different abstractions into a single interface. In this paper, we propose a many-objective search-based approach to automatically refactor service interfaces. A refactoring solution consists of a sequence of interface changes that optimizes five objectives of cohesion, coupling, number of interfaces, the interfaces size deviation, and the number of service antipatterns in the generated interfaces. We validated our approach on a benchmark of 22 real-world Web services provided by Amazon and Yahoo. Our results provide evidence that the produced interfaces are able to improve the service design quality and achieve results similar, on average, to manually performed refactorings by developers at 81% of precision and 80% of recall.

Hanzhang Wang
University of Michigan, Dearborn, MI, USA. E-mail: wanghanz@umich.edu

Ali Ouni
Osaka University, Osaka, Japan. E-mail: ali@ist.osaka-u.ac.jp

Marouane Kessentini
University of Michigan, Dearborn, MI, USA. E-mail: marouane@umich.edu

Wiem Mkaouer
University of Michigan, Dearborn, MI, USA. E-mail: mkaouer@umich.edu

Kalyanmoy Deb
Michigan State University, MI, USA. E-mail: kdeb@egr.msu.edu

Keywords Web services · interface design · SOA · refactoring · modularization · search-based software engineering.

1 Introduction

Service-oriented computing (SOC) has emerged as a paradigm that changed the way software applications are designed, delivered and consumed. Within the SOC paradigm, the composition of loosely coupled and coarse-grained pieces of software, called *services*, drives the low-cost development of distributed applications in heterogeneous environments namely service-based systems (SBSs) [1]. According to a recent statistics from seekda.com[1], a major service provider, there are 28,606 Web services available on the Web, offered by over 7,739 different providers.

The interfaces are the main source of interactions with the user to adopt and reuse the proposed services in real-world applications. Thus, it is important to ensure that the interface is easy to understand, maintain and use which may lead to a better usability and popularity of the proposed services [2,3,4,5,6,7,8,9,10,11,12]. However, existing work in the area of the quality of services (QoS) focus on providing solutions to improve the quality from the service providers perspective, such as availability, response time and security, but not from the service users perspective. Thus, very few studies addressed quality issues related to the services interface design level [13,14,15].

The structure of a service interface is critical in SOA. However, developers tend to take little care of their service WSDL descriptions as several researchers have pointed out [13,14,15,16]. Most of these existing descriptions are designed in only one interface grouping all the operations together. To this end, Web service bad design practices and antipatterns have been recently studied, and different approaches found that several of existing Web service interfaces are suffering from bad design practices and proposed solutions to detect them [14,15,17,18].

In this paper, we propose an approach, namely MOWSIR (Many-Objective Web Service Interface Refactoring), that aims at improving the design quality of service interfaces by improving its structure and modularization while fixing existing antipatterns. MOWSIR generates a refactoring solution which is a sequence of changes to restructure the list of operations provided by a Web service through appropriate interfaces, i.e., port types in terms of size and design. In fact, the number of possible refactoring combinations to explore is exponentially high, leading to a large and complex search space. The best refactoring solution should optimize 5 objectives: (1) maximize interfaces cohesion; 2) minimize interfaces coupling; (3) maximize the number of interfaces; and (4) minimize the interface size deviation; and (5) minimize the number of service antipatterns in the generated interfaces. We, thus, consider the Web service interface refactoring task as a many-objective optimization problem using the

[1] `https://seekda.com/about`

new many-objective non dominated sorting genetic algorithm (NSGA-III) [19]. NSGA-III was recently used to address the problem of the modularization of object-oriented programs using a set of quality attributes[20]. However, the problem addressed in this paper is different from [20] since the modularization process is just limited to regrouping classes in terms of packages. In addition, the used objectives and inputs/outputs are different from [20].

To validate our approach, we conducted an empirical study on a benchmark of 22 real-world services, provided by Amazon and Yahoo. The study aimed to evaluate the potential that a design quality improvement of an interface refactoring solution proposed by our approach will achieve. We then compared our approach to the only existing service refactoring approach in the current literature by Athanasopoulos et al. [13]. The approach proposed by [13] is based on a greedy algorithm for the refactoring of service interfaces based on cohesion metrics, using one type of refactorings (interface partitioning), without consideration of antipatterns and did not use intelligent exploration of the search space such as metaheuristic search algorithms. The obtained results confirm that our many-objective formulation improved the quality of the interfaces and the generated refactoring solutions are similar to manually performed refactorings by developers at 81% of precision and 80% of recall, on average.

The rest of this paper is organized as follows. Section 2 provides the necessary background along with a motivating example. Our approach is discussed in Section 3. Our empirical study design is described in Section 4, while the obtained results are presented and discussed in Section 5. Threats to validity are analyzed in Section 6, while the related work is discussed in Section 7. Finally, we conclude and describe future research directions in Section 8.

2 Background and Motivation

In this section, we present the necessary background for Web service interface refactoring and antipatterns, along with a real-world motivating example for our problem.

2.1 Definitions

A Web Service is defined according to the W3C[2] (World Wide Web Consortium), as “a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artefacts. Its interface is described as a WSDL (Web service Description Language) document that contains structured information about the Web service’s location, its offered operations, the input/output parameters, etc.”

A Web service interface corresponds to a WSDL port type, which is the most important WSDL element. A Web service has at least one interface. This

[2] `http://www.w3.org/TR/wsdl20/`

WSDL element describes a Web service, the operations that can be performed, and the messages that are involved. It can be compared to a function library (or a module or a class) in a traditional programming language.

Modularity. Service interface *modularity* can be defined as the degree to which the operations of a service belong together and well partitioned into cohesive interfaces. A good *modularization* of a design leads to a service which is easier to use, design, develop, test, maintain, and evolve. The importance of design modularity was best articulated by David et al. [21]: "*perhaps the most widely accepted quality objective for design is modularity*". Although modularity tends to be a subjective concept, measuring the degree of modularization of a software design can be achieved through two quality measures: cohesion and coupling [22].

Antipatterns are symptoms of poor design and implementation practices that describe bad solutions to recurring design problems. They often lead to software which is hard to maintain and evolve [23].Different types of antipatterns presenting a variety of symptoms have been recently studied with the intent of improving their detection and suggesting improvements paths [15] [17] [24]. Common Web service interface antipatterns include:

- *God object Web service interface (GOWS)*: exposes a multitude of methods related to different business and technical abstractions in a single interface. Consequently the service is often unavailable to end users because it is overloaded [24].
- *Fine grained Web service interface (FGWS)*: is a too fine-grained service interface whose overhead (communications, maintenance, and so on) outweighs its utility. This antipattern refers to a small Web service interface with few operations implementing only a part of an abstraction[24].
- *Chatty Web service interface (CWS)*: represents an antipattern where a high number of operations, typically attribute-level setters or getters, are required to complete one abstraction[24]. This antipattern may have many fine-grained operations, which degrades the overall performance with higher response time .
- *Data Web service interface (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[15].
- *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. Interfaces designed in that way might be chatty because multiple operations need to be invoked to achieve one goal[24].

In this paper, we focus mainly on these five antipattern types as they are the antipatterns that occur most frequently in SBSs based on recent studies [15,17,25,26,27].

Refactoring. Software refactoring is defined by Fowler [28] as "*the process of changing the internal structure of a software to improve its quality without altering the external behavior*". Refactoring is recognized as an essen-

Table 1: Web service interface refactorings, amd their parameters, pre- and post-conditions.

Refactoring	ID	Parameters	Pre and Post-conditions
Interface Partitioning (si1,si2,ops[ ])	IP	si1: SourcePortTypye si2: TargetPortTypye ops[ ]: Operations to be extracted to si2	**Pre:** $\exists si1 \wedge isPortType(si1) \wedge$ $\nexists si2 \wedge si1.size \geqslant 2 \wedge ops.lengh \geqslant 1$ **Post:** $\exists si2 \wedge isPortType(si2) \wedge si1.size \geqslant 1 \wedge$ $si2.size \geqslant 1$
Interface Consolidation (si1,si2)	IC	si1: SourcePortTypye si2: TargetPortTypye	**Pre:** $\exists si1 \wedge isPortType(si1) \wedge \exists si2 \wedge isPortType(si2)$ **Post:** $\exists si1 \wedge isPortType(si1) \wedge \nexists si2$
Move Operation (op,si1,si2)	MO	op: SourceOperation si1: SourcePortTypye si2: TargetPortTypye	**Pre:** $\exists si1 \wedge isPortType(si1) \wedge \exists si2 \exists isPortType(si2) \wedge$ $op.interface = si1 \wedge \exists op \wedge \neg declares(si2, op) \wedge$ $si1.size \geqslant 2$ **Post:** $si1.size \geqslant 1 \wedge \neg declares(si1, op) \wedge declares(si2, op)$

tial practice to improve software quality. Dudney et al. [24] have defined an initial catalog of refactoring operations for Web services including *Interface Partitioning*, *Interface Consolidation*, *Bridging Schemas or Transforms* and *Web Service Business Delegate*. Despite being commonly used in the Object-Oriented Programming (OOP) paradigm and widely supported by OOP integrated development environments (IDEs), refactoring is still unexplored in the context of service-oriented computing (SOC). In fact, SOC refactoring is not a trivial case of recoding existing OOP refacoring techniques.

There are few WSDL refactoring tools[3] that have emerged to provide basic operations, however these tools do not support developers in decision making with advanced WSDL refactoring techniques. Our approach proposed in this paper, defines and supports three WSDL refactoring operations based on the literature [24, 28, 27]:

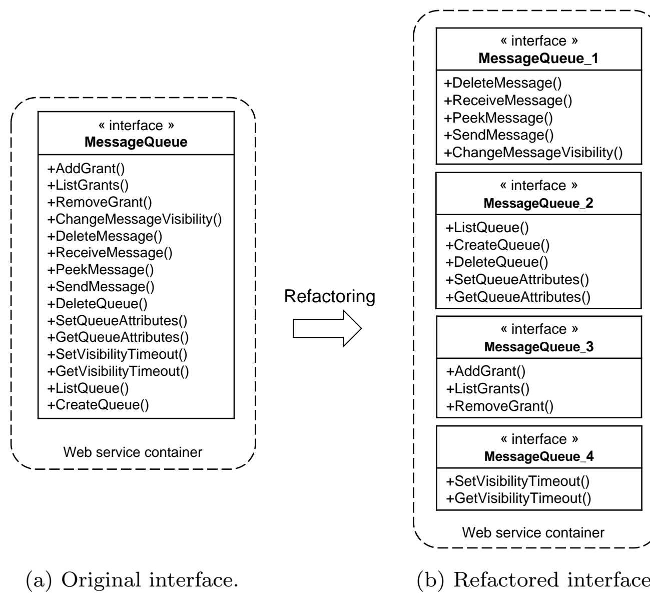
- *Interface Partitioning*: This refactoring decompose a large, multi-abstraction interface into multiple interfaces that each represents a distinct abstraction.
- *Interface Consolidation*: This refactoring merges a set of interfaces that collectively implement a complete, single abstraction. Different service interfaces that operate against the same abstraction are merged into one interface that represents a single cohesive abstraction.
- *Move Operation*: This refactoring moves an operation from one interface to another one. It implies deciding what the core abstraction should be, and moving the operations that do not fit that abstraction to some other interface(s).

Table 1 presents the set of parameters, pre and post-conditions required for each of our adopted refactorings[29].

2.2 Motivating Example

To illustrate some of the Web service interface design anomalies, let us consider an example of real-world Web service provided by Amazon, namely the

[3] `https://www.soapui.org/soap-and-wsdl/wsdl-refactoring`



(a) Original interface. (b) Refactored interface.

Fig. 1: The Amazon `MessageQueue`[4] service interface.

Amazon Message Queue Service (SQS)[4]. SQS is a scalable queue service that enables communication via message queues, allocated on the Amazon infrastructure. Figure 1a depicts `MessageQueue` v2007, one of the main interfaces of the SQS service providing a large number of operations, which enable deleting a queue, getting/setting certain queue attributes/timeouts, adding/ removing/listing access permissions for a particular queue, sending messages to a queue, receiving messages from a queue and changing the visibility of messages.

One can clearly notice that the `MessageQueue` interface exposes various functionalities that do not belong together including queue attributes management, access rights management, message exchange operations. This design makes the service harder to understand and reuse in business processes. Potential developers who aims at implementing a queue client to exchange messages through this interface should understand the whole API documentation and the WSDL specification of SQS. However, only 4 operations are actually related to the exchange of messages.

A better SOA design practice could consider partitioning the `MessageQueue` interface into appropriately-sized, cohesive and loosely coupled interfaces that relate to the management of queue attributes, the management of access rights and the exchange of messages as depicted in Figure 1b. This would simplify the comprehension of the functionalities that the developer actually needs . Indeed, in the 2008 release of the SQS service, the access rights management

[4] `http://docs.aws.amazon.com/AWSSimpleQueueService/2007-05-01/SQSDeveloperGuide/`

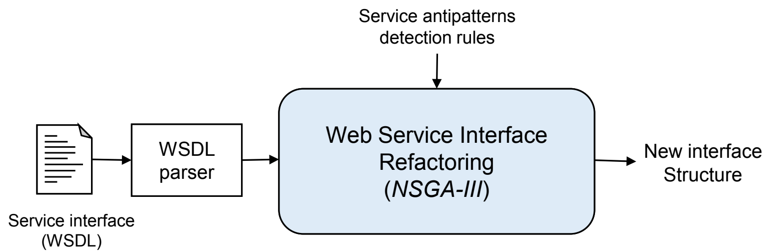


Fig. 2: Approach overview.

operations (`ListGrants()`, `AddGrant()`, `RemoveGrant()`) were removed from the original `MessageQueue` interface as an attempt to improve the service reusability and performance.

3 Web Services Refactoring as a Search Problem

In this section, we describe our approach for Web service interface refactoring problem using NSGA-III.

3.1 Approach overview

A refactoring solution consists of a sequence of change operations to restructure the list of operations provided by a Web service within appropriate interfaces, i.e., port types. The search space is determined not only by the number of possible refactoring operations combinations, but also by the number of provided operation and the order in which they are applied. A heuristic-based optimization method is used to generate refactoring solutions. The best refactoring solution should optimize 5 objectives: (1) maximize interfaces cohesion; (2) minimize interfaces coupling; (3) maximize the number of interfaces; (4) minimize the interface size difference (standard deviation of number of operations per interface); and (5) minimize the number of service antipatterns in the generated interfaces.

Figure 2 illustrates the overall architecture of our approach to the Web service interface refactoring problem. Our approach takes as input a Web service interface WSDL file/url to be refactored and Web service antipatterns detection rules [14,18]. Then, it parses the WSDL sources by tree walking up the XML hierarchy. It then analyses the parsed WSDL through structural and semantic analysis in order to extract structural and semantic relationships among operations. The extracted information will be used in an optimization process based on the recent non-dominated sorting genetic algorithm (NSGA-III) [19] to generate the suitable refactoring solutions. The output of our algorithm is a set of optimal refactoring solutions that should find the best trade-off between the five objectives defined above.

3.2 Service interface Cohesion and Coupling

3.2.1 Cohesion

Cohesion is a widely used metric in SOC to measure how strongly related are the operations of a service interface [13] [30] [31] . Our approach employs three commonly used interface cohesion metrics that will drive the refactoring search process: sequential, communicational, and conceptual cohesion. Our cohesion metrics focus on interface-level relations, as service implementation is typically not provided by the service providers.

Lack of sequential cohesion (LoC_{seq}). The sequential similarity S_{seq} between two operations quantifies the sequential category of cohesion [31]. Two operations are deemed to be connected by a sequential control flow if the output from an operation is the input for the second operation, or vice versa. Formally, let $op_1, op_2 \in si$, two operations belonging to an interface si, then S_{seq} is defined as follows:

$$S_{seq}(op_1, op_2) = \frac{MS(I(op_1), O(op_2)) + MS(O(op_1), I(op_2))}{2} \tag{1}$$

where $I(op)$ and $O(op)$ refer to the input and output messages of the operation op, respectively; and $MS(I(op_1), O(op_2))$ is the function that returns the message similarity between two messages $I(op_1)$ and $O(op_2)$.

To calculate message similarity (MS), we consider two messages as similar if they have common parameters, or similar types of parameters. To calculate MS of two messages m_1 and m_2, our approach is based on the number of common primitive types which corresponds to the Jaccard similarity between m_1 and m_2, i.e., the ratio of common primitive types in m_1 and m_2, divided by the union of primitive types of m_1 and m_2. The more two messages share common primitive types, the more they are likely to be related.

The Lack of sequential cohesion LoC_{seq} of an interface si is defined as the complement of the average S_{seq} of all pairs of operations belonging to the interface si [30]. Formally, LoC_{seq} is defined as follows:

$$LoC_{seq}(si) = 1 - \frac{\sum\limits_{\substack{\forall(op_i, op_j) \in si \\ op_i \neq op_j}} S_{seq}(op_i, op_j)}{\frac{|si| \times (|si|-1)}{2}} \tag{2}$$

Lack of communicational cohesion (LoC_{com}). The Communicational Similarity S_{com} between two operations quantifies the communicational category of cohesion [31]. Two service operations are deemed to be connected by a communication similarity, if they share (or use) common parameter and return types, i.e., both operations are related by a reference to the same set of input and/or output data. Formally, let m_1 and m_2, two operations, then S_{com} is defined as follows:

$$S_{com}(op_1, op_2) = \frac{MS(I(op_1), I(op_2)) + MS(O(op_1), O(op_2)}{2} \quad (3)$$

where $I(op)$ and $O(op)$ refer to the input and output messages of the operation op, respectively; and $MS(I(op_1), I(op_2))$ is the function that returns the message similarity between two messages $I(op_1)$ and $I(op_2)$.

The LoC_{com} of an interface si is defined as the complement of the average S_{com} of all pairs of operations belonging to the interface si [30]. Formally, LoC_{com} is defined as follows:

$$LoC_{com}(si) = 1 - \frac{\sum\limits_{\substack{\forall(op_i, op_j)\in si \\ op_i \neq op_j}} S_{com}(op_i, op_j)}{\frac{|si| \times (|si|-1)}{2}} \quad (4)$$

Lack of semantic cohesion (LoC_{sem}). The Semantic Similarity S_{sem} between two operations quantifies the conceptual category of cohesion. We define a concrete refinement of the conceptual cohesion, as it is regarded as the strongest cohesion metric [32].

S_{sem} is based on the meaningful semantic relationships between two operations, in terms of some identifiable domain level *concept*. We expand the existing definition to get more meaningful sense of the semantic meanings embodied in the operation names. To this end, we perform a lexical analysis on the operations signature. Our lexical analysis consists of the four following steps: (1) we first tokenize all operation names using a camel case splitter, (2) then we use a stop word list to cut-off and filter out all common English words[5] from the extracted tokens, (3) then we lemmatize all extracted terms using the Stanford's CoreNLP[6] to reduce each term to its basic form in order to group together the different inflected forms of a basic word so they can be analyzed as a same word, (4) we used `WordNet`[7], a widely used lexical database that groups words into sets of cognitive synonyms, each representing a distinct concept. We use `WordNet` to enrich and add more informative sense to the extracted bag of words for each operation. For example, the word *customer* can be used with different synonyms (e.g., *client*, *purchaser*, etc.), but pertaining to a common domain concept.

To capture semantics or textual similarity between two bags of words A and B extracted from two operations op_1 and op_2 respectively, we use the cosine of the angle between both vectors representing A and B in a vector space using *tf-idf* (term frequency-inverse document frequency) model. We interpret term sets as vectors in the n-dimensional vector space, where each dimension corresponds to the weight of the term (tf-idf) and thus n is the overall number of terms. Formally, the S_{sem} between op_1 and op_2 corresponds

[5] `http://www.textfixer.com/resources/common-english-words.txt`

[6] `nlp.stanford.edu/software/corenlp.shtml`

[7] `wordnet.princeton.edu`

to the cosine similarity of their two weighted vectors **A** and **B** and defined as follows:

$$S_{sem}(op_1, op_2) = cosine(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\| \times \|\mathbf{B}\|} \quad (5)$$

The LoC_{sem} of an interface si is defined as the complement of the average S_{sem} of all pairs of operations belonging to the interface si. Formally, LoC_{sem} is defined as follows:

$$LoC_{sem}(si) = 1 - \frac{\sum_{\substack{\forall (op_i, op_j) \in si \\ op_i \neq op_j}} S_{sem}(op_i, op_j)}{\frac{|si| \times (|si|-1)}{2}} \quad (6)$$

Lack of cohesion (*LoC*) The *LoC* metric covers all possible aspects of service interface cohesion as captured by the previously defined metrics LoC_{seq}, LoC_{com} and LoC_{sem}. Thus, it quantifies the total (overall) cohesion of a service interface. *LoC* of an interface si is defined as follows:

$$LoC(si) = w_{seq} * LoC_{seq}(si) + w_{com} * LoC_{com}(si) + w_{sem} * LoC_{sem}(si) \quad (7)$$

where $w_{seq} + w_{com} + w_{sem} = 1$ and their values denote the weight of each similarity measure.

3.2.2 Coupling

We define the *Coupling* metric between two service interfaces si_1 and si_2 as the average similarity between all possible pairs of operations from si_1 and si_2. Formally, the coupling, *Cpl*, is defined as follows:

$$Cpl(si_1, si_2) = \frac{\sum_{\forall op_i \in si_1, \forall op_j \in si_2} Sim(op_i, op_j)}{| si_1 | \times | si_2 |} \quad (8)$$

where $| si_1 |$ denotes the number of operations in the interface si_1, and $Sim(op_i, op_j)$ is defined as the weighted sum of the different operations similarity measures defined in the previous section:

$$Sim(op_i, op_j) = w_{seq} * S_{seq}(op_i, op_j) + w_{com} * S_{com}(op_i, op_j) + w_{sem} * S_{sem}(op_i, op_j) \quad (9)$$

where $w_{seq} + w_{com} + w_{sem} = 1$ and their values denote the weight of each similarity measure.

In the following, we describe how coupling, cohesion and other objectives are considered in our many-objective formulation.

3.3 Many-objective formulation

3.3.1 NSGA-III

NSGA-III is a recent many-objective algorithm proposed by Deb et al. [19]. The basic framework remains similar to the original NSGA-II algorithm with significant changes in its selection mechanism. Algorithm 3 gives the pseudo-code of the NSGA-III procedure for a particular generation t. First, the parent population P_t (of size N) is randomly initialized in the specified domain, and then the binary tournament selection, crossover and mutation operators are applied to create an offspring population Q_t. Thereafter, both populations are combined and sorted according to their domination level and the best N members are selected from the combined population to form the parent population for the next generation.

Input: H structured reference points Z_s, population P_t
Output: population P_{t+1}
1: $S_t \leftarrow \emptyset, i \leftarrow 1$
2: $Q_t \leftarrow$ Variation(P_t)
3: $R_t \leftarrow P_t \cup Q_t$
4: $(F_1, F_2, \ldots) \leftarrow$ Nondominated_Sort(R_t)
5: **repeat**
6: $S_t \leftarrow S_t \cup F_i$
7: $i \leftarrow i + 1$
8: **until** $|S_t| \geq N$
9: $F_l \leftarrow F_i$
10: **if** $|S_t| = N$ **then**
11: $P_{t+1} \leftarrow S_t$
12: **else**
13: $P_{t+1} \leftarrow \bigcup_{j=1}^{l-1} F_j$
14: $K \leftarrow N - |P_{t+1}|$
15: Normalize(F^M, S_t, Z^r, Z^s)
16: $[\pi(s), d(s)] \leftarrow$ Associate(S_t, Z^r)
17: $p_j \leftarrow \sum_{s \in S_t/F_l} ((\pi(s) = j)\ ?\ 1 : 0)$
18: Niching($K, p_j, \pi(s), d(s), Z^r, F_l, P_{t+1}$)
19: **end if**

Fig. 3: NSGA-III procedure at generation t.

The fundamental difference between NSGA-II and NSGA-III lies in the way the niche preservation operation is performed. Unlike NSGA-II, NSGA-III starts with a set of reference points Z^r. After non-dominated sorting, all acceptable front members and the last front F_l that could not be completely accepted are saved in a set S_t. Members in S_t/F_l are selected right away for the next generation. However, the remaining members are selected from F_l such that a desired diversity is maintained in the population. Original NSGA-II uses the crowding distance measure for selecting well-distributed set of points, however, in NSGA-III the supplied reference points (Z^r) are used to select these remaining members as described in Figure 4. To accomplish this, ob-

jective values and reference points are first normalized so that they have an identical range. Thereafter, orthogonal distance between a member in S_t and each of the reference lines (joining the ideal point and a reference point) is computed. The member is then associated with the reference point having the smallest orthogonal distance. Next, the niche count p for each reference point, defined as the number of members in S_t/F_l that are associated with the reference point, is computed for further processing. The reference point having the minimum niche count is identified and the member from the last front F_l that is associated with it is included in the final population. The niche count of the identified reference point is increased by one and the procedure is repeated to fill up population P_{t+1}.

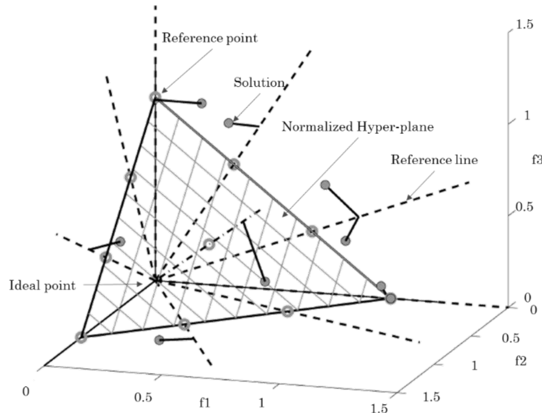


Fig. 4: Normalized reference plane for a three-objective case

It is worth noting that a reference point may have one or more population members associated with it or need not have any population member associated with it. Let us denote this niche count as p_j for the j-th reference point. We now devise a new niche-preserving operation as follows. First, we identify the reference point set $J_{\min} = \{j : \text{argmin}_j(p_j)\}$ having minimum p_j. In case of multiple such reference points, one ($j^* \in J_{\min}$) is chosen at random. If $p_{j*} = 0$ (meaning that there is no associated P_{t+1} member to the reference point j^*), two scenarios can occur. First, there exists one or more members in front F_l that are already associated with the reference point j^*. In this case, the one having the shortest perpendicular distance from the reference line is added to P_{t+1}. The count p_{j*} is then incremented by one. Second, the front F_l does not have any member associated with the reference point j^*. In this case, the reference point is excluded from further consideration for the current generation. In the event of $p_{j*} \geq 1$ (meaning that already one member associated with the reference point exists), a randomly chosen member, if exists, from front F_l that is associated with the reference point F_l is added to P_{t+1}. If such a member exists, the count p_{j*} is incremented by one. After p_j counts

are updated, the procedure is repeated for a total of K times to increase the population size of P_{t+1} to N.

3.3.2 Solution Approach

Our solution approach uses Search-based Software Engineering (SBSE) [33, 34] that provides best practice for formulation software engineering problems as search problems. We describe our SBSE formulation in terms of solution representation, fitness function and change operators.

Solution Representation. To represent a candidate service refactoring solution (individual), we use a vector representation. Each vector's dimension represents a refactoring operation. Thus, a solution is defined as a sequence of operations applied to the service description document (WSDL) to improve its structure. A randomly generated solution is created as follows. First, we generate the solution length randomly between the lower ($minSize$) and upper ($maxSize$) bounds of the solution length. Thereafter, for each chromosome dimension, we generate a number x between 1 and the total number of possible operations, then we assign the ith operation to the considered dimension. For each operation, the parameters, described in Table 1, are randomly generated from the list of WSDL elements. Moreover, for each refactoring operation, it is important to check that it satisfies its pre and post-conditions to guarantee that it is feasible and that it can be legally applied.

An example of a refactoring solution applied to the MessageQueue interface depicted in Figure 1 consists of three consequent `Interface Partitioning` refactorings to extract three additional interfaces, each representing a core abstraction.

Objective functions. Our approach aims at finding the suitable refactoring solutions that optimize five objectives.

- ***Objective 1: Cohesion.*** The cohesion objective function is a measure of the overall cohesion of a candidate refactoring solution. This objective function corresponds to the complement of the average cohesion score of each interface in the service description document (WSDL) and is computed as follows:

$$Cohesion(WS) = 1 - \frac{\sum\limits_{\forall si \in WS} LoC(si)}{\mid WS \mid} \quad (10)$$

  where $LoC(si_i)$ denotes the total interface lack of cohesion given by equation 9, and $\mid WS \mid$ is the total number of interfaces in the service WS.
- ***Objective 2: Coupling.*** The coupling objective function measures the overall coupling among interfaces in a service WS. This objective function corresponds to the average coupling score between all possible pairs interfaces in a the refactored service WS and is calculated as follows:

$$Coupling(WS) = \frac{\sum\limits_{\substack{\forall (si_i, si_j) \in WS \\ si_i \neq si_j}} Cpl(si_i, si_j)}{\frac{|WS| \times (|WS| - 1)}{2}} \quad (11)$$

where $Cpl(si_i, si_j)$ denotes the coupling between the interfaces si_i and si_j given by equation 8, and $\mid WS \mid$ is the total number of interfaces in the service WS.

Typically, coupling among service interfaces should be minimized as this indicates that each interface covers separate functionality aspects.

- ***Objective 3: Number of antipatterns.*** The presence of antipatterns in a service interface makes it hard to understand, maintain and resue. A good refactoring solution should reduce the number of antipatterns in the refactored interface. To this end, we calculate the number of Service antipatterns after applying the suggested refactorings. The number of antipatterns (NA) is calculated as follows:

$$NA(WS) = \sum_{\forall si_i \in WS} isAntipattern(si_i) \quad (12)$$

where the function $isAntipattern(si_i)$ return the number of antipatterns in the interface si_i using specific service antipatterns detection rules for each antipattern type described in Section 2. An antipattern detection rule is a combination of metrics/thresholds to be used as indicators for the existence of an antipattern [14,18]. Indeed, antipatterns are in conflicting consideration with cohesion and coupling, for instance fixing a god object service interface by applying the Interface Partitioning refactoring can result in an increase in the overall coupling of the Web service.

- ***Objective 4: Number of interfaces.*** Our refactoring goal is to maximize the number of interfaces in order to avoid having all operations in a single large interface. This objective function refers to the total number of interfaces in the WSDL document of WS.

$$NI(WS) = \mid WS \mid \quad (13)$$

- ***Objective 5: Standard deviation of number of operations per interface.*** The standard deviation of number operations per interface (STDOI) in a WSDL document of a Web service is ought to be minimized to aim at appropriately, equal-sized interfaces. STDOI is computed as follows:

$$STDOI(WS) = \sqrt{\frac{\sum\limits_{\forall si_i \in WS} (\mid si_i \mid - \mu)^2}{\mid WS \mid}} \quad (14)$$

where $\mid si_i \mid$ is the count of number of operations in the interface si_i; μ is the mean value of the number of operations in all the service interfaces $si_i \in WS$; and $\mid si_i \mid$ is the count of the number of interfaces in the service WS.

Variation operators. In order to evolve NSGA-III individuals, we use crossover and mutation operators. For *crossover*, we used a single-point crossover with a probability p_{cross}. Given two parents, a single-point crossover cuts-off them, i.e., their vector, a random position k, and then all genes after position k are exchanged to produce two offspring solutions. For *mutation*, we use the bit-string mutation operator with a probability p_{mut}. Mutation operator picks probabilistically one or more operations from its or their associated sequence and replaces them by other operations and other parameters from the initial list of possible operations as described in Table 1.

4 Empirical study design

In this section, we present the design of the experiment conducted to evaluate our approach. The purpose of this study is to investigate how well MOWSIR provides refactoring solutions and compare it with available state-of-the-art approach [13].

All the materials used in our study as well as the raw results are publicly available in a comprehensive replication package [35].

4.1 Research questions

We designed our experiments to address the following research questions.

- **RQ1.** *To what extent can* MOWSIR *improve the service interface design quality?*
  This RQ aims at evaluating the impact of the suggested refactorings by our approach on the interface design quality in terms of cohesion, coupling, modularity and antipatterns.
- **RQ2.** *To what extent is* MOWSIR *able to provide appropriate automated refactoring changes?*
  This RQ aims at comparing the refactored service interfaces generated by MOWSIR with those manually performed by developers in terms of precision and recall. The goal is to see if our approach can actually achieve results similar to what developers did manually.
- **RQ3.** *Does* MOWSIR *improve the Web service interfaces design from a developer's point of view?*
  This RQ aims at evaluating our approach with independent developers to assess how well is the Web services interface design generated by MOWSIR, and give more qualitative feedback.
- **RQ4.** How does the proposed approach compared to random search and other popular metaheuristic search methods?
  This RQ is a 'sanity check' to investigate whether our formulation is adequate or not. If an intelligent search method does not outperform a random search then there is a problem with the problem formulation [33].

4.2 Context selection

Objects. To evaluate our approach, we conducted our experiment on a benchmark of 22 real-world services provided by Amazon[8] and Yahoo[9]. We selected services with interfaces exposing at least 10 operations. We chose these Web services because their WSDL interfaces are publicly available, and they were previously studied in the literature [13,36]. Table 2 presents our used benchmark.

Subjects. In the context of **RQ2** and **RQ3**, our evaluation involved 14 independent volunteer participants including 6 industrial developers and 8 graduate students. In particular, 3 senior developers from *Browser Kings*[10], 3 developers from *Accunet Web Services*[11], 3 MSc and 5 PhD candidates in Software Engineering. We first gathered information about the participant's background. All participants are familiar with service-oriented development and SOAP Web services with an experience ranging from 4 to 9 years. The participants were unaware of the techniques MOWSIR and Athanasopoulos et al. neither the particular research questions, in order to guarantee that there will be no bias in their judgement.

Baseline approach. We compare our results with a recent state-of-the art approach by Athanasopoulos et al. [13], a Web service refactoring approach based on a greedy algorithm to refactor and split Web service interfaces based on different cohesion measures.To the best of our knowledge, currently there is no significant established state-of-the-art in terms of Web service interface refactoring. We, thus, attempt to compare our approach against Athanasopoulos et al. as a baseline approach.

4.3 Method Analysis

To answer **RQ1**, we evaluate the design improvement that a candidate refacroring solution suggested by MOWSIR will bring to the service. Moreover, we compare our results with a recent state-of-the art approach by Athanasopoulos et al. [13], a Web service refactoring approach based on a greedy algorithm to split Web service interfaces based on different cohesion measures.

To this end, we use four metrics (i) cohesion gain (CohG), (ii) coupling gain (CplG), (iii) modularity gain (MG), and (iv) number of antipatterns (NA). For CohG and CplG, we calculate the difference between the service cohesion (cf. Equation 10) and coupling (cf. 10 of the services after and before refactoring. Modularity evaluates the balance between coupling and cohesion by combining them into a single measurement. The aim is to reward increased cohesion with a higher modularity score and to punish increased coupling with a lower modularity score. It has been proved that the higher the value

[8] `http://aws.amazon.com/`
[9] `developer.searchmarketing.yahoo.com/docs/V6/reference/`
[10] `http://www.browserkings.com`
[11] `http://www.accunet.us`

Table 2: Amazon and Yahoo benchmark overview.

Service interface	Provider	ID	#operations	LoC
AutoScalingPortType	Amazon	I1	13	0,91
MechanicalTurkRequesterPortType	Amazon	I2	27	0,87
AmazonFPSPorttype	Amazon	I3	27	0,94
AmazonRDSv2PortType	Amazon	I4	23	0,82
AmazonVPCPortType	Amazon	I5	21	0,90
AmazonFWSInboundPortType	Amazon	I6	18	0,87
AmazonS3	Amazon	I7	16	0,87
AmazonSNSPortType	Amazon	I8	13	0,92
ElasticLoadBalancingPortType	Amazon	I9	13	0,87
MessageQueue	Amazon	I10	13	0,92
AmazonEC2PortType	Amazon	I11	87	0,96
KeywordService	Yahoo	I12	34	0,89
AdGroupService	Yahoo	I13	28	0,81
UserManagementService	Yahoo	I14	28	0,95
TargetingService	Yahoo	I15	23	0,81
AccountService	Yahoo	I16	20	0,93
AdService	Yahoo	I17	20	0,85
CompaignService	Yahoo	I18	19	0,88
BasicReportService	Yahoo	I19	12	0,94
TargetingConverterService	Yahoo	I20	12	0,72
ExcludedWordsService	Yahoo	I21	10	0,69
GeographicalDictionaryService	Yahoo	I22	10	0,81

of modularity, the better is the design quality [37]. The modularity metric is computed as the average of the overall cohesion and coupling, whereas the MG refers to the difference of the modularity after and before refactoring.

To answer **RQ2**, we asked our group of 14 independent developers described in Section 4.2, to manually refactor each of the studied Web service (cf. Table 2) in order to improve their interface readability and understandability.The resulted interfaces by the developers were considered as the ground truth, allowing the calculation of the precision and recall of our approach. We compute he precision and recall scores as follows:

$$Precision = \frac{TP}{TP + FP} \tag{15}$$

$$Recall = \frac{TP}{TP + FN} \tag{16}$$

where TP (*True Positive*) corresponds to an interface identified by the independent developer and also by the proposed approach; FP (*False Positive*) corresponds an interface identified by the proposed developer, but not by the independent expert; FN (*False Negative*) corresponds to an interface identified by the independent developer, but not by the proposed approach.

Note that we computed TP, FP and FN at a fine-grained level, meaning that the interface identified by the proposed approach and by the independent developer should match with a Jaccard similarity of at least 80% in terms of their operations.

To answer **RQ3**, we asked our 14 participants to evaluate the relevance of three refactoring interfaces, for each of the 22 services: (i) the interfaces re-

sulted by MOWSIR, (*ii*) the interfaces resulted by Athanasopoulos et al., and (*iii*) random refactored interfaces using random search. The random refactoring option is considered as a 'sanity check' to make sure whether participants have seriously answered this study, as a random refactoring does not make sense.

To this end, we used a survey hosted in *eSurveyPro*[12], an online Web application. Specifically, for each refactored interface, we provide a high-level description of each service interfaces before and after refactoring using UML classes (as represented in Figure 1). Then, the participants was asked to answer the following question for each refactoring solution:

> "*Does the new refactored interfaces improve the understandability of the service?*"

Possible answers follow a five-point Likert scale [38] to express their level of agreement: 1: *Strongly disagree*, 2: *Disagree*, 3: *Neutral*, 4: *Agree*, 5: *Fully agree*. Note that the Web application used for our survey allowed our participants to save and complete the study in multiple rounds within a maximum of 7 days available to respond. At the end of the 7 days we collected the 14 complete questionnaires.

To draw statistically sound conclusions, we compared the participants evaluations of MOWSIR, Athanasopoulos et al. and Random refactorings using the Wilcoxon rank sum test in a pairwise fashion [39] in order to detect significant efficiency differences between MOWSIR and Athanasopoulos et al.. Moreover, to assess the efficiency difference magnitude, we studied the effect size based on the Cliff's Delta statistic [40] that will we detail in Section 4.4.

To answer **RQ4**, we compare our NSGA-III formulation against random search (RS) [41] in terms of search space exploration. The goal is to make sure that there is a need for an intelligent method to explore our huge search space of possible refactoring solutions. In addition, to justify the adoption of NSGA-III, we compared our approach against two other popular search algorithms namely Indicator-Based Evolutionary Algorith (IBEA) [42], and the Multi-Objective Evolutionary Algorithm based on Decomposition (MOEA/D) [43]. RQ4 serves the role of a *sanity check* and standard 'baseline' question asked in any attempt at an SBSE formulation [33].

Unlike mono-objective search algorithms, multi and many-objective evolutionary algorithms return as output a set of *non-dominated* (also called *Pareto optimal*) solutions obtained so far during the search process. A number of performance metrics for multi-objective optimization have been proposed and discussed in the literature, which aim to evaluate the performance of multi-objective evolutionary algorithms. Most of the existing metrics require the obtained set to be compared against a specified set of Pareto optimal reference solutions. In this study, the generational distance (GD)[44] and inverted generational distance (IGD) [45] are used as the performance metrics since thay have been shown to reflect both the diversity and convergence of the obtained non-dominated solutions.

[12] http://www.esurveyspro.com

- *Generational Distance (GD)*: computes the average distance between the set of solutions, S, from the algorithm measured and the reference set RS. The distance between S and RS in an n objective space is computed as the average n-dimensional Euclidean distance between each point in S and its nearest neighbouring point in RS. GD is a value representing how "far" S is from RS (an error measure) [44].
- *Inverted Generational Distance (IGD)*: is used as a performance indicator since it has been shown to reflect both the diversity and convergence of the obtained non-dominated solutions [45]. The IGD corresponds to the average Euclidean distance separating each reference solution set (RS) from its closest non-dominated one S. Note that for each system we use the set of Pareto optimal solutions generated by all algorithms over all runs as reference solutions.

Moreover, we use the Modularity Gain (MG) and Number of Antipatterns (NA) measures as performance indicators to evaluate the efficiency of each algorithm.

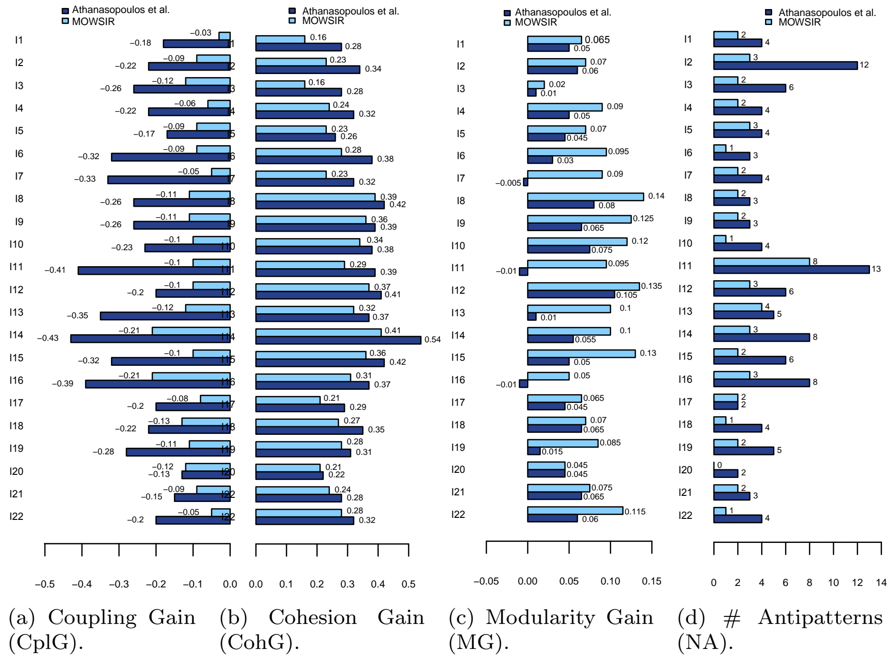
4.4 Inferential Statistical Tests Used

Due to the stochastic nature of the employed algorithms, they may produce different results when applied to the same problem instance over different runs. In order to cope with this stochastic nature, the use of statistical testing is essential to provide support and draw statistically sound conclusions derived from analyzing such data [46]. To this end, we used the Wilcoxon rank sum test in a pairwise fashion [47, **?**] in order to detect significant performance differences between the algorithms under comparison. The Wilcoxon test does not require that the data sets follow a normal distribution since it operates on values' ranks instead of operating on the values themselves. We set the confidence limit, α, at 0.05. In these settings, each experiment is repeated 31 times, for each algorithm and for each system. The obtained results are subsequently statistically analyzed with the aim to compare our NSGA-III approach with NSGA-II, MOEA/D and random search (RS).

While the Wilcoxon rank sum test verifies whether the results are statistically different or not, it does not give any idea about the difference magnitude. To this end, we investigate the effect size using Cliff's delta statistic [40]. The effect size is considered: (1) 'negligible' if $\mid d \mid < 0.147$, (2) 'small' if $0.147 \leq \mid d \mid < 0.33$, (3) 'medium' if $0.33 \leq \mid d \mid < 0.474$, or (4) 'large' if $\mid d \mid \geq 0.474$.

4.5 Parameter Tuning and Setting

we set the different parameter values of our algorithm by trial-and-error method, which is commonly used by the SBSE community [46, 48]. The initial population/solution of NSGA-III is completely random. The stopping criterion is



(a) Coupling Gain (CplG). (b) Cohesion Gain (CohG). (c) Modularity Gain (MG). (d) # Antipatterns (NA).

Fig. 5: Quality improvements achieved by MOWSIR and Athanasopoulos et al. in terms of CohG, CplG, MG, and NA.

when the maximum number of fitness evaluations, set to 350,000, is reached. After several trial runs of the simulation, the parameter values of the algorithm is fixed to 100 solutions per population ($popSize$) and 3,500 iterations. For the variation operators, we set crossover rate, p_{cross}, is set to 0.9 and mutation, p_{mut}, at 0.4 probability. We used a high mutation rate to ensure the diversity of the population and avoid premature convergence to occur [49]. After several trial runs of the simulation, these parameter values are fixed. Indeed, there are no general rules to determine these parameters, and thus we set the combination of parameter values by trial-and-error, a method that is commonly used by the SBSE community [46, 48].

5 Study results

This subsection presents the results obtained for RQs 1–4 defined in Section 4.1.

5.1 Results for RQ1

Figure 5 presents the results achieved by both MOWSIR and Athanasopoulos et al. in terms of CohG, CplG, MG and NA. We expected an increase of

cohesion (desired effect) due to the split of different operations exposed in the original interface. However, we also expected an increase of coupling (side effect), since splitting an interface into several interfaces typically results in an increment of the total dependencies between these interfaces. For these reasons coupling and cohesion should be measured together to make a proper judgment on the complexity and quality of the refactored interfaces (MG).

As depicted in Figure 5b, for almost all the services, the cohesion is significantly improved by both approaches. In particular, the improvement achieved by Athanasopoulos et al. is better than MOWSIR. However, Figure 5a shows the coupling results which is decreased compared to the initial coupling. Indeed, this is natural as most of the original services has a single interface (thus its coupling = 0). Consequently, any interface restructuring will result in some connections between interfaces due to the semantic similarity that is unlikely to be equals to zero and due to some shared (primitive) data types in operation messages. As reported in figure 5b, MOWSIR has remarkably achieved low coupling for all the services. Indeed, improvement of cohesion usually comes at the expense of increase in coupling and vice versa.

Overall, we assume that a candidate refactoring is a good design solution if the improvement of cohesion is significantly greater than the deterioration of coupling. This balance is captured by the MG metric as reported in Figure 5c. For both Amazon and Yahoo services, interesting modularity improvements was achieved by MOWSIR with an average of 0.07, while Athanasopoulos et al. approach turns out to be less effective with an average of 0.04. Furthermore, as reported in Figure 5c, Athanasopoulos et al. produced 3 out of 22 services with deteriorated modularity due to the high coupling resulted in the new interfaces, specifically for `AmazonS3` (I7), `AmazonEC2` (I11) and `AccountService` (I16).

On the other hand, Figure 5d reports the number of resulted antipatterns in the refactored services by each approach. On average, MOWSIR generated refactored interfaces with 2.3 antipatterns per service, while 5.1 are resulted by Athanasopoulos et al.. Indeed, although antipatterns are sometimes unavoidable in service interfaces, reducing them is necessary to improve the service reusability, extendibility and performance. Moreover, we observed that most of detected antipatterns in MOWSIR are related to the chatty/CRUDy service antipattern. For instance, in the `AmazonEC2`, most of the operations are already setters, getters or CRUDy operations leading to several instances of chatty and CRUDy interfaces.

5.2 Results for RQ2

Table 3 reports the results for RQ2 in terms of precision and recall of each of MOWSIR and Athanasopoulos et al.. For the 22 services, MOWSIR achieve an average precision of 81% and an average recall of 80% comparing the manual refactoring performed by developers. We consider that these values of precision and recall are high since a deviation between the proposed solution and the manual one may not be an indication of some wrong recommendations but

it could be just another possible good solution. In fact, there is no single good design solutions of Web service interface but multiple ones. On the other hand, we noticed that Athanasopoulos et al. tends to produce more interface splits generating several fine-grained interfaces. Indeed, fine-grained interfaces tend to have higher cohesion. This resulted in low precision and recall with an average of 27% and 33%, respectively. We noticed that, Athanasopoulos et al. generated in many cases, several interfaces with only one operation. Such fine-grained interfaces will make the service more complex and severely limit its reusability as core abstractions will be split into several small and scattered interfaces.

We noticed that all fourteen participants were able to identify more independent, and sometimes completely disconnected interfaces from the original interface. These interfaces are usually the best candidates for split since they present core abstractions and do not bare strong dependency from the rest of the original interface.

Another interesting observation was that MOWSIR successfully identified design solutions with 100% of precision and recall in 7 out of the 22 services while Athanasopoulos et al. succeeded to do so only twice. On the other hand, Athanasopoulos et al. turns out to completely fail in identifying appropriate design in 4 cases out of 22 with 0% of precision and recall.

Finally, we identify a main drawback of the Athanasopoulos et al.'s approach [13] from our perspective. Diving Web service interface refactoring with only cohesion metrics would not be enough, and coupling, size of interfaces, and number of antipatterns are as important as cohesion for good service interface design.

5.3 Results for RQ3

Table 4 reports the results achieved by our study for the developers assessment. We observed that for all the studied services, the participants rated the MOWSIR interfaces with an average score of 3.86, an average of 2.59 for the Athanasopoulos et al. approach, while an average of 1.21 was obtained for the random refactorings. This provides evidence that the our solutions are more adjusted to developers needs than those of Athanasopoulos et al.. In addition, as reported in table 4, the rating results of MOWSIR and Athanasopoulos et al. was statistically different with a 'large' effect size, except for the services I13 and I20, the effect size was negligible, as the refactoring results was similar with 100% of precision and recall (Table 3).

It is worth to note that during the evaluation, we discovered some common operations provided by different Amazon services. For example, we found that `AmazonVPCPortType` and `AmazonEC2PortType` have several common operations including `CreateVpc()`, `DescribeVpcs()`, `DeleteVpc()`, `DeleteVpnConnection()`, `CreateVpnGateway()` and `DeleteVpnGateway()`. More interestingly, some generated interfaces from both `AmazonVPCPortType` and `AmazonEC2PortType` expose exactly the same operations. Although this redundancy can be related to

Table 3: Comparison results of MOWSIR and Athanasopoulos et al. in terms of precision and recall.

Web service	MOWSIR		Athanasopoulos et al.	
	Precision	Recall	Precision	Recall
AutoScalingPortType	75%	75%	17%	33%
MechanicalTurkRequester	100%	100%	0%	0%
AmazonFPSPorttype	80%	89%	27%	30%
AmazonRDSv2PortType	83%	83%	20%	20%
AmazonVPCPortType	100%	100%	0%	0%
AmazonFWSInboundPortType	67%	67%	40%	33%
AmazonS3	75%	60%	17%	20%
AmazonSNSPortType	80%	67%	17%	20%
ElasticLoadBalancingPortType	50%	67%	0%	0%
MessageQueue	75%	67%	50%	60%
AmazonEC2PortType	82%	90%	14%	18%
KeywordService	78%	88%	11%	14%
AdGroupService	100%	100%	100%	100%
UserManagementService	100%	100%	36%	71%
TargetingService	83%	83%	13%	20%
AccountService	100%	100%	7%	17%
AdService	80%	57%	25%	17%
CompaignService	67%	50%	14%	25%
BasicReportService	100%	100%	57%	80%
TargetingConverterService	100%	100%	100%	100%
ExcludedWordsService	67%	67%	33%	50%
GeographicalDictionaryService	33%	50%	0%	0%
Average	**81%**	**80%**	**27%**	**33%**

some business constraints/factors, best design practice in SOA suggests that common core abstractions could be implemented in separate service interfaces, making them easier to maintain, evolve and reuse.

An interesting point here was that the participants confirmed that the interfaces refactored by MOWSIR tend to be more appropriately sized and describe distinct abstractions with less overlap. We asked one of the participants to comment on his decision for the generated Amazon EC2 interfaces, he claimed: "*This new interface structure is more understandable to me, it should allow the service to be reused and maintained more effectively*".

Moreover, we noticed that in most of the cases, Athanasopoulos et al. approach tend to split core abstractions into many interfaces. For instance, in `AmazonEC2`, operations related to image management was dispersed through many other interfaces: operations `RegisterImage()` and `DescribeImages()` are assigned to a new interface, `DescribeImageAttribute()` is in another interface, `CreateImage()` is in another interface, `ResetImageAttribute()`, `DeregisterImage()` and `ModifyImageAttribute()` are in another interface along with other unrelated operations [13]. We asked another participant comment on this new design, his answer was: "*Such scattered abstractions will result in several connections between interfaces for no benefit as a large number of resulted interfaces are not representing core abstractions*". On the other hand, we noticed that due to the high resulted coupling, most of the identified

Table 4: Developer's evaluation of the resulted Web service interfaces design for MOWSIR (M), Athanasopoulos et al. (A), and random refactorings (R).

Service ID	**Ratings**			**Statistical tests**			
				M^* vs $A^{\ddagger}$		M^* vs R^+	
	M^*	$A^{\ddagger}$	R^+	p-value	effect size	p-value	effect size
I1	3,85	2,57	1,21	<0,01	0.765 (large)	<0,01	0.928 (large)
I2	4,00	2,43	1,07	<0,01	0.826 (large)	<0,01	0.928 (large)
I3	3,92	2,43	1,21	<0,01	0.877 (large)	<0,01	0.928 (large)
I4	3,79	2,50	1,29	<0,01	0.729 (large)	<0,01	0.928 (large)
I5	3,79	2,50	1,36	<0,01	0.866 (large)	<0,01	0.928 (large)
I6	4,00	2,29	1,07	<0,01	0.87 (large)	<0,01	0.928 (large)
I7	4,00	2,21	1,21	<0,01	0.915 (large)	<0,01	0.928 (large)
I8	3,64	2,14	1,36	<0,01	0.94 (large)	<0,01	0.928 (large)
I9	3,92	2,21	1,21	<0,01	0.906 (large)	<0,01	0.928 (large)
I10	3,93	2,36	1,36	<0,01	0.746 (large)	<0,01	0.928 (large)
I11	3,71	1,71	1,14	<0,01	0.775 (large)	<0,01	0.928 (large)
I12	3,93	2,50	1,21	<0,01	0.639 (large)	<0,01	0.928 (large)
I13	3,71	3,71	1,07	No. diff	-0,23 (neglig.)	<0,01	0.928 (large)
I14	3,79	2,64	1,14	<0,01	0.81 (large)	<0,01	0.928 (large)
I15	4,07	2,57	1,14	<0,01	0.75 (large)	<0,01	0.928 (large)
I16	3,71	2,71	1,35	<0,01	0.751 (large)	<0,01	0.928 (large)
I17	3,85	3,00	1,29	<0,01	0.562 (large)	<0,01	0.928 (large)
I18	4,00	2,93	1,29	<0,01	0.58 (large)	<0,01	0.928 (large)
I19	3,93	2,64	1,00	<0,01	0.775 (large)	<0,01	0.928 (large)
I20	4,00	4,00	1,36	No. diff	-0,23 (neglig.)	<0,01	0.928 (large)
I21	3,64	2,36	1,07	<0,01	0.873 (large)	<0,01	0.928 (large)
I22	3,64	2,50	1,15	<0,01	0.843 (large)	<0,01	0.785 (large)
Avg.	**3,86**	**2,59**	**1,21**				

* MOWSIR ‡ Athanasopoulos et al. $^+$ Random refactorings

Athanasopoulos et al. interfaces expose operations related to different core abstractions. For instance, for the same Amazon EC2 service, a suggested interface by Athanasopoulos et al. contains `DetachVolume()`, `AttachVolume()` and `DescribeInstanceAttribute()`. Results show that this design is unlikely to be desirable for developers [13]. Moreover, the obtained results suggest that reducing coupling and antipatterns is as important metric as cohesion to drive Web service interface refactoring.

5.4 Results for RQ4

Figure 6 and Table 6 present the results of the performance indicators *GD*, *IGD*, *MG* and *NA* and the results of the statistical significance and effect size tests, respectively. For each pair of algorithms, we have eight experiments (four performance indicators × two categories of Web services, Amazon and Yahoo). We observe that NSGA-III clearly outperforms RS in all the studied Amazon and Yahoo services with a Cliff's delta effect size of 'large' in the four performance indicators *GD*, *IGD*, *MG*, and *NA*. This is mainly due to

Table 5: Statistical significance p-value (α=0.05) and Cliff's d effect size comparison results of NSGA-III against IBEA, MOEA/D and RS in terms of Generational Distance (GD), Inverted Generational Distance (IGD), Modularity Gain (MG), and Number of Anti-patterns (NA). A statistical difference is accepted at $p \leqslant 0.05$.

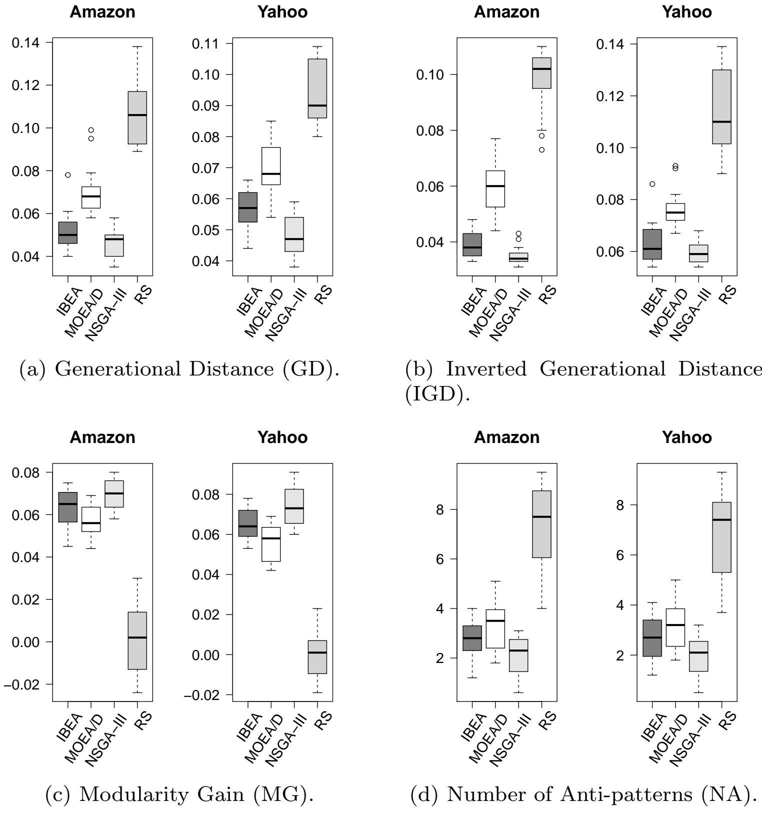
Pairs of Algorithms	**Quality measures**	**Amazon**		**Yahoo**	
		p-value	*d*	*p-value*	*d*
NSGA-III vs IBEA	GD	0.028	-0.347 (medium)	9.773e-05	-0.661 (large)
	IGD	0.0001	-0.587 (large)	0.04691	-0.229 (small)
	MG	0.0035	0.408 (medium)	0.001	0.481 (large)
	NA	0.0006	-0.452 (medium)	0.001	-0.42 (medium)
NSGA-III vs MOEA/D	GD	1.229e-06	-0.993 (large)	1.821e-06	-0.959 (large)
	IGD	1.227e-06	-1 (large)	1.226e-06	-0.993 (large)
	MG	2.249e-05	0.758 (large)	2.595e-06	0.802 (large)
	NA	0.0001	-0.612 (large)	1.762e-05	-0.637 (large)
NSGA-III vs RS	GD	1.224e-06	-1 (large)	1.231e-06	-1 (large)
	IGD	1.23e-06	-1 (large)	1.23e-06	-1 (large)
	MG	1.229e-06	0.967 (large)	1.233e-06	0.967 (large)
	NA	1.232e-06	-1 (large)	1.231e-06	-1 (large)

the large search space to explore to find suitable combinations of refactoring operations. This requires a heuristic-based search rather than random search.

In more detail, Figure6 and Table 6 report our results for RQ4. We observe that over 31 runs, NSGA-III outperforms IBEA in 3 out of 8 experiments with 'large' effect size. In the cases of Amazon, the effect size was 'medium' in terms of *GD*, *MG*, and *NA*. For the Yahoo services, NSGA-III achieved a 'small' effect size with IBEA in terms of *IGD*, and 'medium' effect size in terms of *NA*. Interestingly, NSGA-III significantly outperforms MOEA/D with a 'large' effect size in all the 8 experiments. We also noticed that NSGA-III tends to achieve relatively better *MG* and *NA* results for Amazon Web services which are larger on average than the Yahoo services, compared to IBEA and MOEA/D.

Furthermore, we can also get a more informative sense of the distributions of results for the three competitive algorithms from the boxplots shown in Figure 6. From these boxplots, we can see that the variance in the results from NSGA-III is lower for both *GD* and *IGD* than the IBEA, MOEA/D and RS. The obtained results suggest that this is because there are simply fewer solutions that converge toward the last generation of NSGA-III. In addition, because *GD* and *IGD* metrics combine the information of convergence and diversity, the results indicate that NSGA-III has the best overall performance.

To conclude, the obtained results provide evidence that NSGA-III is the best search technique for the Web service remodularization problem. Consequently, we can conclude that there is empirical evidence that our formulation passes the sanity check (RQ4).



(a) Generational Distance (GD).

(b) Inverted Generational Distance (IGD).

(c) Modularity Gain (MG).

(d) Number of Anti-patterns (NA).

Fig. 6: Boxplots for the quality measures GD, IGD, MG, and NA results over 31 independent simulation runs of NSGA-III, IBEA, MOEA/D, and RS.

6 Threats to validity

In this section, we identify factors that may affect the validity of our study. A possible threat to construct validity can be related to the set of ground truth to calculate precision and recall with refactorings performed manually by developers.

An external threat can be related to the studied services. Although we used 22 real-world Web services provided by Amazon and Yahoo, from different application domains and ranging from 10 to 87 operations, we can not generalize our results to other services and other technologies, e.g., REST services. As part of our future work, we plan to test our approach with an extended benchmark of Web services.

An internal threats to validity can be related to the knowledge and expertise of the human evaluators. Inadequate knowledge could lead to limited ability to assess the quality of an interface. We mitigate this threat by se-

lecting participants having from 4 to 9 years experience with service-oriented development and familiar with SOAP Web services. Moreover, to avoid bias in the experiment none of the authors have been involved in this evaluation. In addition, we randomized the ordering in which the MOWSIR, Athanasopoulos et al. and random refactorings were shown to participants, to mitigate any sort of learning or fatigue effect.

7 Related Work

This section reviews the related literature in three different areas (*i*) Web service antipatterns, (*ii*) software refactoring, and (*iii*) software modularization.

Web service antipatterns. Detecting antipatterns, i.e., bad design and implementation practices, in Web services and SOA is a relatively new field. The first book in the literature was written by Dudney et al. [24] and provides informal definitions of a set of Web service antipatterns. More recently, Rotem-Gal-Oz described the symptoms of a range of SOA antipatterns [27]. Furthermore, Král et al. [17] listed seven "popular" SOA antipatterns that violate accepted SOA principles. Recently, Ouni et al. [14,18] proposed a search-based approach to automatically detect Web service antipatterns including the god object Web service, fine-grained Web service, ambiguous Web service and semantically unrelated operations in port types. The proposed approach uses genetic programing identify Web service interfaces that present symptoms of poor design practices. Moha et al. [50] have proposed a rule-based approach called SODA for SCA systems (Service Component Architecture). Later, Palma et al. [15] extended this work for Web service antipatterns in SODA-W. The proposed approach relies on declarative rule specification using a domain-specific language (DSL) to specify/identify the key symptoms that characterize an antipattern using a set of WSDL metrics. In other work [51], the authors presented a repository of 45 general antipatterns in SOA, to support developers to work with clear understanding of patterns in phases of software development and so avoid many potential problems.

Software refactoring. Software refactoring have been practiced for more many years [28]. It is widely recognized that, if applied well, refactoring brings a lot of advantages to improve software readability, maintainability and extendibility. One of the first attempts to address service interface refactoring was by Athanasopoulos et al. [13]. Although their approach was able to improve cohesion, it is not perfectly adjusted to the developers' needs [13]. Ouni et al. [52] proposed a graph partioning-base approach namely SIM for spliting service APIs based on their cohesion and semantic inofrmation. The proposed approach aims at identified chains of operations that are strongly connected. Another work by Romano et al. [53,54] addressed the problem of fat interfaces clustering based on the Interface Segregation Principle in order to split interfaces based on methods invoked by groups of clients. However, the most notable limitation of these existing works is that coupling between interfaces and antipatterns are not considered which resulted in many cohesive but highly

coupled interfaces. Our approach addresses explicitly this issue limitation to improve the modularization quality. In another study, Rodriguez et al. [55, 56] and Mateos et al. [57] 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.

A lot of efforts has been devoted to refactoring of object-oriented (OO) applications. Our approach is more closely similar to *Extract Class* refactoring in OO systems, which employs metrics to split a large class into smaller, more cohesive classes [28]. Bavota et al. [58, 59] have proposed an extract class approach to split a large class into smaller cohesive classes using structural and semantic similarity measures. Fokaefs et al. [60] proposed an automated extract class refactoring approach based on a hierarchical clustering algorithm to identify cohesive subsets of class methods and attributes. However, the *Extract Class* refactoring is not applicable in the context of Web services as typically the Web service source code is not publicly available, and the development paradigm, used technologies and metrics are different.

Software modularization. Several studies addressed the problem of clustering and remodularization of OO applications in terms of packages organization. Anquetil et al. [61] used cohesion and coupling of modules within a decomposition of OO systems to evaluate its quality. Maqbool et al. [62] used hierarchical clustering in the context of software architecture recovery and modularization. On the other hand, Mancoridis et al. [63] proposed the first search-based approach to address the problem of software modularization using a single objective approach. Harman et al. [64] used a genetic algorithm to improve subsystems decomposition by combining several quality metrics including coupling, cohesion, and complexity. Recently, Mkaouer et al. [20] 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, still this problem is not widely explored in the context of service interfaces.

8 Conclusion and Future Work

In this paper, we have proposed a many-objective search-based approach, namely MOWSIR to refactor Web service interfaces and validated it on 22 real-world Web services. Our approach used NSGA-III to improve the structure of Web service interfaces while fixing detected antipatterns. Our results provide evidence that MOWSIR performs significantly better than state-of-the art techniques in terms of design quality improvement. Moreover, we evaluated our approach with 14 developers since we strongly believe that an automated approach must suggest refactoring solutions that fit with developers expectations. The obtained results show that our generated refactoring solutions are similar to manually performed refactorings by developers at 81% of precision and 80% of recall, on average.

As future work, we plan to extend our work to consider additional Web-services in our validation to generalize our findings, and by giving weights to the different antipattern types. In addition, we are planning to extend our refactoring types and balance between refactorings in the code and interface levels. Furthermore, we will consider the programmer in-the-loop when identifying the refactoring solutions.

References

1. M. P. Singh and M. N. Huhns, *Service-oriented computing - semantics, processes, agents*. Wiley, 2005.
2. M. Fleck, J. Troya, M. Kessentini, M. Wimmer, and B. Alkhazi, "Model transformation modularization as a many-objective optimization problem," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1009–1032, 2017.
3. A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. German, and K. Inoue, "Search-based software library recommendation using multi-objective optimization," *Information and Software Technology*, vol. 83, pp. 55–75, 2017.
4. H. Wang, M. Kessentini, and A. Ouni, "Bi-level identification of web service defects," in *International Conference on Service-Oriented Computing*. Springer, Cham, 2016, pp. 352–368.
5. M. W. Mkaouer, M. Kessentini, M. Ó. Cinnéide, S. Hayashi, and K. Deb, "A robust multi-objective approach to balance severity and importance of refactoring opportunities," *Empirical Software Engineering*, vol. 22, no. 2, pp. 894–927, 2017.
6. M. Kessentini, R. Mahaouachi, and K. Ghedira, "What you like in design use to correct bad-smells," *Software Quality Journal*, vol. 21, no. 4, pp. 551–571, 2013.
7. A. ben Fadhel, M. Kessentini, P. Langer, and M. Wimmer, "Search-based detection of high-level model changes," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2012, pp. 212–221.
8. M. Kessentini, H. Sahraoui, M. Boukadoum, and M. Wimmer, "Search-based design defects detection by example," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, Berlin, Heidelberg, 2011, pp. 401–415.
9. M. Kessentini, H. Sahraoui, and M. Boukadoum, "Example-based model-transformation testing," *Automated Software Engineering*, vol. 18, no. 2, pp. 199–224, 2011.
10. M. Kessentini, M. Wimmer, H. Sahraoui, and M. Boukadoum, "Generating transformation rules from examples for behavioral models," in *Proceedings of the Second International Workshop on Behaviour Modelling: Foundation and Applications*. ACM, 2010, p. 2.
11. U. Mansoor, M. Kessentini, M. Wimmer, and K. Deb, "Multi-view refactoring of class and activity diagrams using a multi-objective evolutionary algorithm," *Software Quality Journal*, vol. 25, no. 2, pp. 473–501, 2017.
12. U. Mansoor, M. Kessentini, B. R. Maxim, and K. Deb, "Multi-objective code-smells detection using good and bad design examples," *Software Quality Journal*, vol. 25, no. 2, pp. 529–552, 2017.
13. 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.
14. 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.
15. F. Palma, N. Moha, G. Tremblay, and Y.-G. Guéhéneuc, "Specification and detection of soa antipatterns in web services," in *Software Architecture*. Springer, 2014, pp. 58–73.
16. M. Crasso, J. M. Rodriguez, A. Zunino, and M. Campo, "Revising WSDL Documents: Why and How," *Internet Computing, IEEE*, no. 5, pp. 48–56.
17. J. Král and M. Zemlicka, "Popular SOA Antipatterns," in *Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns*, 2009, pp. 271–276.

18. A. Ouni, R. Gaikovina Kula, M. Kessentini, and K. Inoue, "Web service antipatterns detection using genetic programming," in *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference*, ser. GECCO'15. ACM, 2015, pp. 1351–1358.
19. K. Deb and H. Jain, "An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: Solving problems with box constraints," *IEEE Transactions on Evolutionary Computation*, vol. 18, no. 4, pp. 577–601, 2014.
20. W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software remodularization using nsga-iii," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 3, pp. 17:1–17:45, May 2015.
21. D. N. Card and R. L. Glass, *Measuring Software Design Quality.* Prentice-Hall, Inc., 1990.
22. D. Budgen, *Software Design.* Addision Wesley, 1999.
23. M. V. Mäntylä and C. Lassenius, "Subjective evaluation of software evolvability using code smells: An empirical study," *Empirical Softw. Engg.*, vol. 11, no. 3, pp. 395–431, Sep. 2006.
24. B. Dudney, J. Krozak, K. Wittkopf, S. Asbury, and D. Osborne, *J2EE Antipatterns.* John Wiley; Sons, Inc., 2003.
25. J. Král and M. Žemlička, "Crucial service-oriented antipatterns," *International Journal On Advances in Software*, vol. 2, no. 1, pp. 160–171, 2009.
26. C. Mateos, M. Crasso, A. Zunino, and J. L. O. Coscia, "Detecting WSDL bad practices in codefirst Web Services," *International Journal of Web and Grid Services*, vol. 7, no. 4, pp. 357–387, 2011.
27. A. Rotem-Gal-Oz, *SOA Patterns.* Manning Publications, 2012.
28. M. Fowler, *Refactoring: improving the design of existing code.* Addison-Wesley Longman Publishing Co., Inc., 1999.
29. D. Webster, P. Townend, and J. Xu, "Interface refactoring in performance-constrained web services," in *15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, 2012, pp. 111–118.
30. 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.
31. 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.
32. M. Perepletchikov, C. Ryan, K. Frampton, and H. Schmidt, "Formalising service-oriented design," *Journal of software*, vol. 3, no. 2, pp. 1–14, 2008.
33. M. Harman and B. F. Jones, "Search-based software engineering," *Information and Software Technology*, vol. 43, no. 14, pp. 833–839, 2001.
34. M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys*, vol. 45, no. 1, pp. 1–61, 2012.
35. "Replication package. available at: http://sel.ist.osaka-u.ac.jp/people/ali/MOWSIR."
36. M. Fokaefs, R. Mikhaiel, N. Tsantalis, E. Stroulia, and A. Lau, "An empirical study on web service evolution," in *IEEE International Conference on Web Services (ICWS)*, July 2011, pp. 49–56.
37. K. J. Stewart, D. P. Darcy, and S. L. Daniel, "Opportunities and challenges applying functional data analysis to the study of open source software evolution," *Statistical Science*, pp. 167–178, 2006.
38. P. M. Chisnall, "Questionnaire design, interviewing and attitude measurement," *Journal of the Market Research Society*, vol. 35, no. 4, pp. 392–393, 1993.
39. J. Cohen, *Statistical power analysis for the behavioral sciences.* Academic press, 1988.
40. N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions." *Psychological Bulletin*, vol. 114, no. 3, p. 494, 1993.
41. D. C. Karnopp, "Random search techniques for optimization problems," *Automatica*, vol. 1, no. 2, pp. 111–121, 1963.
42. E. Zitzler and S. Künzli, "Indicator-based selection in multiobjective search," in *Parallel Problem Solving from Nature-PPSN VIII.* Springer, 2004, pp. 832–842.

43. Q. Zhang and H. Li, "Moea/d: A multiobjective evolutionary algorithm based on decomposition," *IEEE Transactions on Evolutionary Computation*, vol. 11, no. 6, pp. 712–731, 2007.
44. D. A. Van Veldhuizen and G. B. Lamont, "Multiobjective evolutionary algorithm research: A history and analysis," 1998.
45. C. A. C. Coello and N. C. Cortés, "Solving multiobjective optimization problems using an artificial immune system," *Genetic Programming and Evolvable Machines*, vol. 6, no. 2, pp. 163–190, 2005.
46. A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Software Engineering (ICSE), 2011 33rd International Conference on.* IEEE, 2011, pp. 1–10.
47. J. Demšar, "Statistical comparisons of classifiers over multiple data sets," *The Journal of Machine Learning Research*, vol. 7, pp. 1–30, 2006.
48. A. E. Eiben and S. K. Smit, "Parameter tuning for configuring and analyzing evolutionary algorithms," *Swarm and Evolutionary Computation*, vol. 1, no. 1, pp. 19–31, 2011.
49. K. Deb and T. Goel, "Controlled elitist non-dominated sorting genetic algorithms for better convergence," in *Evolutionary Multi-Criterion Optimization.* Springer, 2001, pp. 67–81.
50. N. Moha, F. Palma, M. Nayrolles, B. J. Conseil, Y.-G. Guéhéneuc, B. Baudry, and J.-M. Jézéquel, "Specification and detection of soa antipatterns," in *Service-Oriented Computing.* Springer, 2012, pp. 1–16.
51. M. A. Torkamani and H. Bagheri, "A Systematic Method for Identification of Antipatterns in Service Oriented System Development," *International Journal of Electrical and Computer Engineering*, vol. 4, no. 1, pp. 16–23, 2014.
52. 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.
53. D. Romano, S. Raemaekers, and M. Pinzger, "Refactoring fat interfaces using a genetic algorithm," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 351–360.
54. D. Romano and M. Pinzger, "A genetic algorithm to find the adequate granularity for service interfaces," in *2014 IEEE World Congress on Services*, 2014, pp. 478–485.
55. J. M. Rodriguez, M. Crasso, C. Mateos, and A. Zunino, "Best practices for describing, consuming, and discovering web services: a comprehensive toolset," *Software: Practice and Experience*, vol. 43, no. 6, pp. 613–639, 2013.
56. J. M. Rodriguez, M. Crasso, A. Zunino, and M. Campo, "Automatically detecting opportunities for web service descriptions improvement," in *Software Services for e-World.* Springer, 2010, pp. 139–150.
57. 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.
58. G. Bavota, A. De Lucia, and R. Oliveto, "Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures," *Journal of Systems and Software*, vol. 84, no. 3, pp. 397–414, 2011.
59. G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Automating extract class refactoring: an improved method and its evaluation," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1617–1664, 2014.
60. M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou, "Identification and application of extract class refactorings in object-oriented systems," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2241–2260, oct 2012.
61. 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.
62. 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.
63. 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.

64. 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.