SIM: An Automated Approach to Improve Web Service Interface Modularization

Ali Ouni*, Zouhour Salem†, Katsuro Inoue*, Makram Soui†
*Graduate School of Information Science and Technology, Osaka University, Osaka, Japan
{ali,inoue}@ist.osaka-u.ac.jp
†High Institute of Management (ISG), SOIE Group, University of Tunis, Tunisia
{salem,soui}@isg.rnu.tn

***Abstract*—Service interface structure is of primary importance in SOA to ensure best practice of third-party reuse. One of the key factors for deploying successful services is assuring an adequate interface structure. However, a common bad service design practice is to place semantically unrelated operations in a single interface. This poor design practice typically result in a system which is difficult to comprehend, maintain and evolve providing low performance and reusability. To address this problem, we present an automated approach, *SIM*, to support service developers improve the quality of their interface modularization. Our approach analyzes structural and semantic relationships among the operations exposed in a service interface to identify chains of strongly related operations. The identified operation chains are used to define new interfaces with higher cohesion and better usability. We empirically evaluate our approach on a benchmark of 22 real-world Web services, provided by Amazon and Yahoo. The obtained results show that the produced interfaces are (*i*) able to improve the service design quality, and (*ii*) recognized as 'useful' from developers point of view in improving their service design. Additionally, we found that *SIM* significantly outperforms a recent state-of-the-art approach.**

***Keywords*-Web service; interface; cohesion; modularization; design;**

I. Introduction

Service-oriented Architecture (SOA) has become the dominant architectural style and the leading edge of contemporary software development. The basic idea is to promote software reuse via ready-made, reusable and composable services that are available to end users who wish to compose them towards constructing a novel application. However, deploying successful services highly depends on how well-designed are the services [1], [2]. Indeed, one of the key factors for a successful service is assuring an appropriate design of its interface.

Recent studies found that developers seem to take little care of the structure of their WSDL documents [3]. A common bad design practice that often appear in real-world services is that their interfaces expose a large number of semantically unrelated operations with low interface cohesion [1], [2], [4]. Service interfaces tend to cover a lot of different abstractions and processes, leading to many operations associated with each abstraction. This will result in poorly designed systems that are hard to comprehend, reuse and maintain [5], leading to unsuccessful services.

As *first-class* design artifact, a service interface should be carefully and properly designed. Best practice for service design suggests that services should expose their operations in an appropriate modularization where each module, i.e., interface, defines operations that handle one abstraction at a time [5], [6]. Service interfaces will consequently exhibit low coupling and high cohesion [7]. Low coupling means that a service interface is dependent to a low number of other types, and interfaces, allowing an effective reuse. Cohesion refers to how strongly related the operations themselves are. High cohesion means that the service operations are related as they operate on the same, underlying core abstraction.

Correctly identifying service interfaces is a challenging and important service-oriented design activity. Service interfaces with unrelated operations often need to be restructured by distributing some of their functionalities to new interfaces, thus reducing their complexity and improving their cohesion, reusability, and maintainability. The research domain that addresses this problem is referred to as "refactoring" [8] to detect and then correct bad code/design practices. Although, there are many recently emerging approaches for detecting service design anomalies, i.e., antipatterns, [1], [9], [10], [11], [12], the correction step is still in its infancy.

One of the first attempts to service interface remodularization was by Athanasopoulos et al. [4], where they proposed an approach driven by a set of cohesion metrics to capture structural and conceptual relationships between operations. However, the concept of coupling between interfaces [13] is not considered which led to undesirable interface splits and highly coupled interfaces. Moreover, as a strongest cohesion metric [6], the conceptual cohesion can be improved to better capture semantic information embodied in operations names.

To address the above mentioned challenges, we propose in this paper a novel approach, namely *SIM* (Service Interface Modularization), to automatically suggest suitable partitioning of a service interface, while maximizing the interface cohesion and minimizing inter-interface coupling. *SIM* exploits different structural (communicational and sequential similarity) and semantic relationships between operations using a graph-based representation of an interface, where the nodes represent the operations and the weights on the edges represent the likelihood that two operations should belong to the same interface.

In an effort to demonstrate the effectiveness of our approach, we conduct an empirical evaluation on a benchmark of 22 real-world Web services, provided by Amazon and Yahoo. We compared our approach to a state-of-the-art technique [4] in terms of what design improvement a candidate remodularization will bring to the service. Moreover, we have qualitatively evaluated the usefulness of *SIM* from a developer point of view.

The results show that the interface remodularization solutions proposed by *SIM* are (*i*) able to significantly increase the cohesion of the refactored interfaces while maintaining an acceptable coupling; and (*ii*) are considered useful by developers in improving service interface design.

The rest of this paper is organized as follows. Section II provides the necessary background along with a motivating example. Our approach, SIM, is discussed in Section III. Our empirical study and its results are presented is Section IV. Threats to validity are analyzed in Section V, while the related work is discussed in Section VI. Finally, our conclusions and future work are stated in Section VII.

II. Background and motivation

In this section, we review some concepts that are prerequisites for our approach, and present a real-world motivating example.

A. Background

Web service. According to the W3C[1] (World Wide Web Consortium), a *Web service* provides a set of interfaces, each interface is defined as a WSDL port type and characterized by a set of operations. An operation corresponds to a particular functionality; its execution requires at most one input message and produces at most one output message. A message is characterized by a set of parameters; a parameter corresponds to either a primitive or complex element (XML type). A complex element have a set of constituent elements.

Modularity. Service interface *modularity* concerns, generally, the degree to which the operations of a service belong together and well partitioned into cohesive interfaces. Indeed, good *modularization* of software leads to system which is easier to design, develop, test, maintain, and evolve.

The importance of design modularity was best articulated by David et al. [14]: "*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 [15].

Cohesion. Service interface cohesion is the measure of the degree to which the operations exposed in a service interface conceptually belong together [5]. There are many types of cohesion including coincidental, logical, temporal, communicational, sequential, external, implementation, and conceptual cohesion [5].

[1]http://www.w3c.org/TR/ws-arch

Coupling. Coupling within a service measures the relationships between implementation elements belonging to the same service [13]. Service interface coupling is a measure of how strongly a service interface is connected to or relies on other service interfaces.

Web service antipatterns. Service 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 [1], [9], [10], [11]. Common Web service antipatterns include the god object Web service, fine-grained Web service, chatty service, ambiguous service, CRUDy interface and, the low cohesive operations in the same port type.

Refactoring. Software refactoring is defined by Fowler [8] 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 essential practice to improve software quality. Dudney et al. [7] 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*. This paper focus on automating the *Interface Partitioning* refactoring to improve service modularization.

B. Motivating example

To illustrate some of salient issues related to poor service interface modularity, let us consider a real-world service, the Amazon Elastic Compute Cloud service (EC2)[2] provided by Amazon. Figure 1 shows a fragment of the major interface of EC2 which exposes a quite large number of operations (87 operations) offering a variety of business abstractions. It allows its users to obtain, configure and control several computing resources including images, volumes, security, instances, and snapshots, grouped in a single interface, `AmazonEC2PortType`.

Consequently, for a client who wants to manage images using EC2 (e.g., client 1 in Figure 1), he should study the specifications of the existing `AmazonEC2PortType` interface which consists of 4,261 lines of WSDL and schema definitions, and a 812 pages API documentation guide[3]. However, only few operations might be useful for client 1 for managing images (*CreateImage()*, *RegisterImage()*, *DeregisterImage()*, *DescribeImages()*, *ModifyImageAttribute()*, *ResetImageAttribute()*, and *DescribeImageAttribute()*).

A more adequate modularization of the provided operations within distinct interfaces would simplify the comprehension and reuse of the functionalities that the client actually needs. For instance, a unique interface for managing images, another for volumes, another for security, another for snapshots, and so on. Indeed, inappropriate interface madularity might lead to a service which is difficult to comprehend and reuse in business processes, hard to maintain

[2]http://s3.amazonaws.com/ec2-downloads/2009-10-31.ec2.wsdl

[3]https://aws.amazon.com/documentation/ec2/

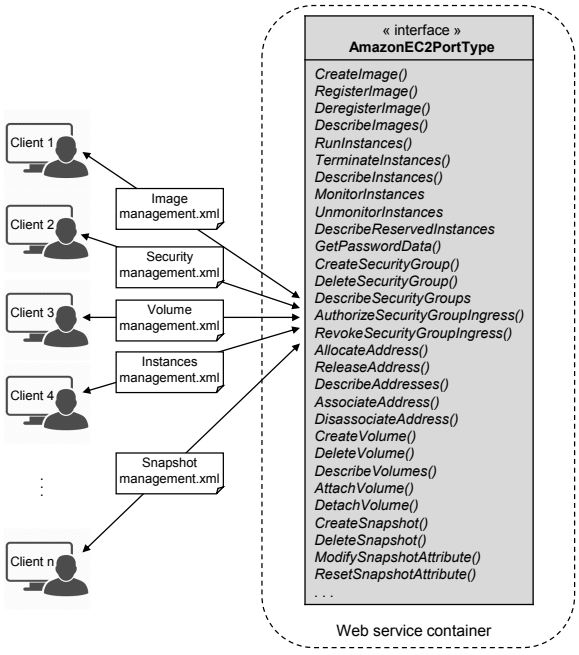


Figure 1: The Amazon Elastic Compute Cloud service (EC2) interface (a fragment of total 87 operations).

and extend because of its large number of non-cohesive operations. Above that, these services may suffer from a high response time or even unavailable to end users because it is overloaded [7].

Thus developers are encouraged to refactor and review their service interfaces to provide best practice of third-party reuse. Motivated by the above mentioned issues, this work aims at providing an automated approach to support developers in improving their service interface structure.

III. The proposed *SIM* approach

Our approach, *SIM*, aims at identifying refactoring opportunities in order to decompose large interfaces with semantically unrelated operations into two or more interfaces. The identified interfaces should have high cohesion, low coupling and attempt to encapsulate related abstractions. *SIM* is not trying to identify service interfaces suffering from semantically unrelated operations, but rather it assumes that such a design problem, i.e., antipattern, is detected [1], and focus on fixing it.

Figure 2 shows our approach process that consists of three main steps: (1) operations similarity extraction, (2) operation chains identification, and (3) light chains merging.

A. Step 1: Operations similarity extraction

SIM takes as input a Web service interface (WSDL file/url) to be refactored. Then, it parses the WSDL sources by tree walking up the XML hierarchy.

The parsed interface will be then analyzed to extract the different relationships between operations. To do so, we use cohesion metrics as an indicator of operations relatedness. *SIM* employs three commonly used interface cohesion metrics to extract operations similarity that will drive the remodularization 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. Similarly, we do not consider information concerning the usage of operations by clients, as this information is mostly influenced by the specific scenario where the service is used.

1) Sequential similarity (S_{seq}): SS quantifies the sequential properties of two service operations as defined by the sequential category of cohesion [5]. 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:

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

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 $O(op_2)$.

Message similarity (MS). Two messages are 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 average of:

- *The number of common subtrees:* it corresponds to the sum of the orders of common bottom-up subtrees of m_1 and m_2, divided by the order of the message that results from the union of m_1 and m_2, as defined in [2].
- *The number of common primitive types:* it 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.

By combining these two measures, MS aims at capturing message similarity. The more two messages share common primitive types, the more they are likely to be similar.

2) Communicational similarity (S_{com}): S_{com} quantifies the communicational properties of two service operations, as defined by the communicational category of cohesion [5]. 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:

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

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 $I(op_2)$.

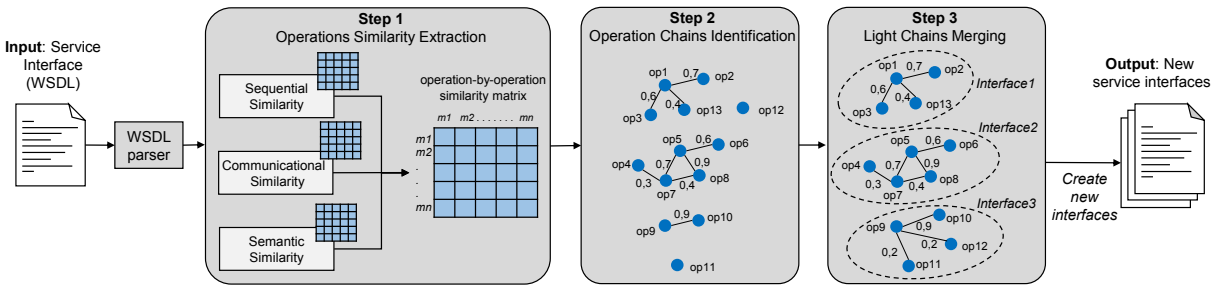


Figure 2: SIM process overview.

3) Semantic similarity (S_{sem}): S_{sem} quantifies the semantic relatedness of operations, as defined by the conceptual category of cohesion. We define a concrete refinement of the conceptual cohesion, as it is regarded as the strongest cohesion metric [6].

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 operation signature. Our lexical analysis consists of the four following steps:

1) Tokenization. The operation names are tokenized using a camel case splitter where each name is broken down into tokens/terms based on commonly used coding standards.
2) Filtering. We use a stop word list to cut-off and filter out all common English words[4] and reserved words from the extracted tokens. Typically, these words are irrelevant to the implemented concept. Such words carry a very low information value and can negatively affect the semantic similarity process as they have no direct relation to the business abstraction domain.
3) Lemmatization. This is a morphological process that transforms each word to its basic form (i.e., lemma). This process aims at reducing a word 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. Hence, different forms of words that may have similar meanings are grouped together and handled as identical word. For example, the verb 'to pay' may appear as 'pay', 'paid', 'paying', 'payment', 'payments'. The base form, 'pay' is then the lemma of all these words. To do so, we use Stanford's CoreNLP[5] to find the base forms of all extracted words.
4) Vocabulary expansion. To enhance the effectiveness of the semantic similarity, we utilize `WordNet`[6], 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 semantic 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 to the cosine similarity of their two weighted vectors $\vec{A}$ and $\vec{B}$ and defined as follows:

$$S_{sem}(op_1, op_2) = cosine(\vec{A}, \vec{B}) = \frac{\vec{A} \cdot \vec{B}}{\|\vec{A}\| \times \|\vec{B}\|} \quad (3)$$

Then, an operation-by-operation matrix is generated by combining all the used structural and semantic similarity measures. Each index in the matrix represents the overall similarity between two operations op_i and op_j, i.e., the likelihood they should be in the same interface. The index is computed follows:

$$Sim(op_i, op_j) = w_{S_{seq}} * S_{seq}(op_i, op_j) + w_{S_{com}} * S_{com}(op_i, op_j) + w_{S_{sem}} * S_{sem}(op_i, op_j) \quad (4)$$

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

B. Step 2 : Operation chains identification

After generating an operation-by-operation matrix for a given interface si, a dependency graph is constructed where the vertices represent the operations and the edges represent the similarity measure between them. Then, the service interface remodularization problem is formulated as a graph partitioning problem.

Due to the fine-grained message similarity and semantic similarity measures between operations, their similarity is often unlikely to be equal to zero. Consequently the generated graph tends to be a connected graph. To deal with this issue, we defined the threshold k as minimum coupling score between subgraphs. We filter the operation-by-operation matrix, based on the threshold k, where all similarity values less than k are converted to zero.

[4]http://www.textfixer.com/resources/common-english-words.txt
[5]nlp.stanford.edu/software/corenlp.shtml
[6]wordnet.princeton.edu

Although there are many ways to setup k according to the preferences of the service developers, it is difficult to choose a standard threshold value for all the interfaces being refactored. In fact, this depends on the context, the application domain, and the naming technique used by the original service developers. To deal with this problem, our approach employs a *dynamic* threshold taking into account the characteristics of the whole interface being refactored. We setup k as the first-quartile computed from all the values in the operation-by-operation matrix after filtering out all the zero values. Thus all similarity values that are less than k are considered as outliers, i.e., low coupling.

After filtering the operation-by-operation matrix and splitting the graph into disconnected subgraphs, we identify the chains of connected operations belonging to the different subgraphs. These chains represent the new interfaces of the service, and the threshold k is therefore used to control the coupling between the identified interfaces.

C. Step 3: Light chains merging

After identification of operation chains, some isolated and light chains (with a single operation or small number of operations) might be generated due to low similarity with the rest of operations in the interface being refactored (e.g., op_{11} and op_{12} in Figure 2). Such fine-grained interfaces are likely to be antipatterns [1], [9], as a core abstraction requires typically more than two operations. To avoid this situation, we define a threshold minimal interface size, $minSize$. Although we fixed $minSize = 2$, it can be easily configured by the developer according to his preferences.

Then, we compute the *Coupling* between light and appropriately-sized interfaces, and merge each light chain with the chain it is most coupled with. To this end, we define the coupling, *Cpl*, between two interfaces si_1 and si_2 as follows:

$$Cpl(si_1, si_2) = \frac{\sum_{op_i \in si_1, op_j \in si_2} Sim(op_i, op_j)}{\mid si_1 \mid \times \mid si_2 \mid} \quad (5)$$

where $Sim(op_i, op_j)$ is defined in equation 4, and $\mid si_1 \mid$ denotes the number of operations in the interface si_1.

Although SIM process is fully automated, the generated interfaces should be analyzed by the service developers who can accept the suggested remodularization as it is, or adjust it by moving operations from one interface to another, or merging/splitting some interfaces.

IV. Validation

This section presents our empirical evaluation to investigate how well *SIM* suggests effective and useful remodularization solutions and compare it with existing state-of-the-art alternative [4].

Our replication package is available online [16] to encourage future research in the field of Web service refactoring.

Table I: Experimental benchmark overview.

Service interface	Provider	ID	#operations	LoC_{seq}	LoC_{com}	LoC_{sem}
AutoScalingPortType	Amazon	I1	13	0,98	0,96	0,79
MechanicalTurkRequesterPortType	Amazon	I2	27	0,84	0,91	0,85
AmazonFPSPorttype	Amazon	I3	27	0,97	0,92	0,93
AmazonRDSv2PortType	Amazon	I4	23	0,96	0,91	0,58
AmazonVPCPortType	Amazon	I5	21	0,96	0,93	0,82
AmazonFWSInboundPortType	Amazon	I6	18	0,96	0,93	0,73
AmazonS3	Amazon	I7	16	0,97	0,89	0,75
AmazonSNSPortType	Amazon	I8	13	0,97	0,96	0,84
ElasticLoadBalancingPortType	Amazon	I9	13	0,97	0,93	0,72
MessageQueue	Amazon	I10	13	0,98	0,98	0,81
AmazonEC2PortType	Amazon	I11	87	0,98	0,97	0,93
KeywordService	Yahoo	I12	34	0,93	0,84	0,91
AdGroupService	Yahoo	I13	28	0,94	0,84	0,65
UserManagementService	Yahoo	I14	28	0,97	0,96	0,91
TargetingService	Yahoo	I15	23	0,96	0,74	0,74
AccountService	Yahoo	I16	20	0,98	0,92	0,88
AdService	Yahoo	I17	20	0,89	0,79	0,88
CompaignService	Yahoo	I18	19	0,91	0,83	0,91
BasicReportService	Yahoo	I19	12	0,99	0,91	0,92
TargetingConverterService	Yahoo	I20	12	0,8	0,84	0,53
ExcludedWordsService	Yahoo	I21	10	0,81	0,72	0,54
GeographicalDictionaryService	Yahoo	I22	10	0,99	0,79	0,65

A. Research questions

We designed our experiments to address the following research questions:

RQ1: *What is the impact of the suggested remodularizations by our approach on service interface design quality*?

RQ2: *Do the suggested remodularizations provide a better partitioning of abstractions from a developer's point of view*?

B. Analysis method

To evaluate our approach, we conducted our experiment on a benchmark of 22 real-world services provided by Amazon[7] and Yahoo[8]. We selected services that are identified as *god object Web service* antipatterns [1], [9] 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 [4], [17]. Table I presents our used benchmark.

To assess the efficiency of our approach, we compare it to a state-of-the-art approach [4]. In the rest of the paper we refer by *Greedy* to denote the approach proposed in [4]. *Greedy* is a cohesion-based approach that iteratively split a service interface using a greedy algorithm without considering the coupling between the generated interfaces.

To answer **RQ1**, we assess the design improvement that a candidate remodularization suggested by *SIM* will bring to the service comparing to *Greedy,Athanasopoulos2015tsc*. Our evaluation is based on Cohesion (*LoC*), Coupling, and Modularity metrics. For cohesion, we use the average of three widely used lack of cohesion metrics: lack of sequential cohesion (LoC_{seq}), lack of communicational cohesion (LoC_{com}), and lack of semantic cohesion (LoC_{sem}) [2]. For coupling, we define $Coupling$ as the average coupling values Cpl (cf. equation 5) between all pairs of produced interfaces. Finally, Modularity is the average of the overall

[7]http://aws.amazon.com/

[8]developer.searchmarketing.yahoo.com/docs/V6/reference/

cohesion and coupling. For each of these three metrics, we report the quality improvement value before and after remodularization, QI_{LoC}, $QI_{Coupling}$, and $QI_{Modularity}$.

To answer **RQ2**, we evaluated our approach from developer's point of view. To this end, we conducted an empirical study involving 12 independent volunteer subjects including 6 industrial developers and 6 graduate students in Software Engineering (2 MSc and 4 PhD candidates). All subjects are familiar with service-oriented development and SOAP Web services with an experience ranging from 4 to 9 years. The subjects were unaware of the techniques *SIM* and *Greedy* neither the particular research questions, in order to guarantee that there will be no bias in their judgment.

We asked the participants to evaluate the proposed remodularization by both *SIM* and *Greedy* via a survey hosted in *eSurveyPro*[9], an online Web application. Participants were asked to answer the following question: "*Does the new modularization improve the understandability of the service?*". Possible answers follow a five-point Likert scale to express their level of agreement: 1: *Strongly disagree*, 2: *Disagree*, 3: *Neutral*, 4: *Agree*, 5: *Fully agree*.

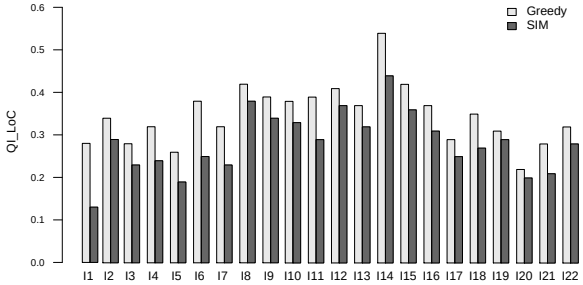
To draw statistically sound conclusions, we compared the participants evaluations of *SIM* and *Greedy* using the Wilcoxon rank sum test in a pairwise fashion [18] in order to detect significant efficiency differences between *SIM* and *Greedy*. Moreover, to assess the efficiency difference magnitude, we studied the effect size based on Cohen's d [18]. The effect size is considered: (1) small if $0.2 \leqslant d < 0.5$, (2) medium if $0.5 \leqslant d < 0.8$, or (3) large if $d \geqslant 0.8$.

C. Results

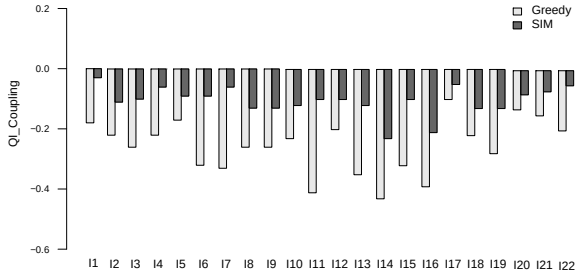
Results for RQ1. Figure 3 reports the results achieved by both *SIM* and *Greedy* in terms of cohesion, coupling and modularity. We expected an increase of cohesion (desired effect) due to the split of different operations exposed in the original interface. However, we also expected an increase of coupling (side effect), since splitting an interface into several interfaces typically results in an increment of the total dependencies between interfaces. For these reasons coupling and cohesion should be measured together to make a proper judgment on the complexity and quality of service interfaces (*Modularity* metric).

Looking at Figure 3a, we can see that for almost all the interfaces the cohesion is sensibly improved by both approaches. In particular, the improvement achieved by *Greedy* is better than *SIM*. However, Figure 3b shows the achieved coupling improvement with a clear deterioration. Indeed, this is natural as the original interface is single (thus its Coupling = 0). Consequently, any interface partitioning 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 messages.

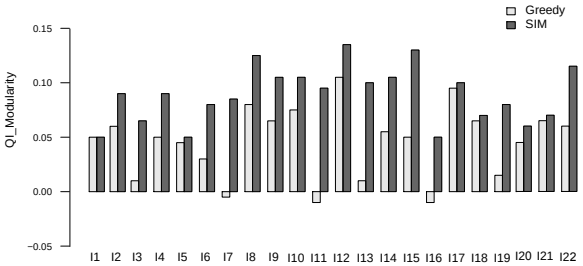
[9]http://www.esurveyspro.com



(a) Cohesion Improvement $QI_{Cohesion}$.



(b) Coupling Improvement $QI_{Coupling}$.



(c) Modularity Improvement $QI_{Modularity}$.

Figure 3: Quality improvements achieved by *SIM* and *Greedy* in terms of Cohesion, Coupling and Modularity.

As reported in figure 3b, *SIM* is able to remarkably reduce the coupling decrease for all the services. Improvement of cohesion usually comes at the expense of increase in coupling and vice versa.

A candidate remodularization is a good design solution if the improvement of cohesion is significantly greater than the deterioration of coupling. This balance is captured by the Modularity metric as reported in figure 3c. For the 22 services, interesting modularity improvements was achieved by *SIM* up to 0.13, while *Greedy* approach turns out to be less effective while recording three services (I7, I11 and I16) have a deteriorated modularity due to the high coupling resulted in the new interfaces.

Furthermore, Figure 4 shows a fragment of the *SIM* remodularization for the Amazon EC2 interface described in Section II-B (We provide full results in our replication package [16]). We noticed that its operations are better partitioned into several cohesive interfaces, where each interface exposes operations for a specific abstraction: instance, address, volume, security, snapshot and image managements. To get more qualitative sense, RQ2 assesses the results from

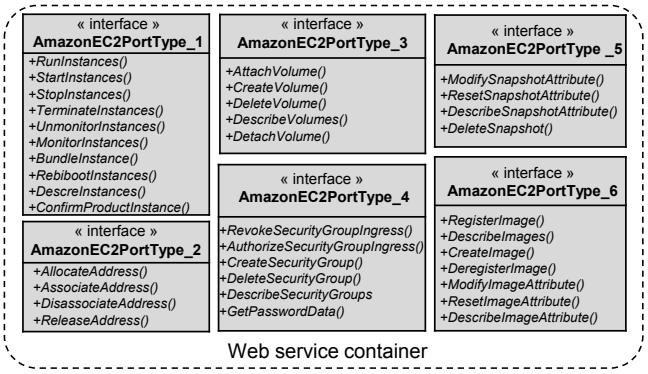


Figure 4: The Amazon EC2 service interface after remodularization using *SIM* (a fragment of total 87 operations).

a developer's perspective.

Results for RQ2. Figure 5 and Table II summarize the developers assessment for the new interface modularizations. For all the studied services, the participants rated the *SIM* remodulations with an average score of 3.71, while an average of 2.48 was recorded for the *Greedy* approach. In addition, as reported in table II, the rating results of *SIM* and *Greedy* was statistically different with a 'large' effect size (only for participants 4, 6 and 12, the effect size was medium). This provides evidence that the interfaces suggested by *SIM* are more adjusted to developers needs than those of *Greedy*. Moreover, on top of the 22 cases, participants identified two services where the original interface is relatively understandable even without remodularization, but they suggested that an early remodularization can be interesting to avoid potential difficulties in future service releases with additional operations.

An interesting point here was that the participants confirmed that the interfaces suggested by *SIM* tend to be more appropriately sized and describe distinct abstractions with less overlap. A participant commented on the generated Amazon EC2 interfaces (Figure 4) : "*This design indicates that service interfaces are not trying to do too much, and allows the service to be reused more effectively*". Moreover, we noticed that *Greedy* approach split some core abstractions into many interfaces. For instance, in the Amazon EC2 interface, 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 [4], [16]. On the other hand, most of the identified interfaces expose operations related to different core abstractions. For instance, for the same Amazon EC2 service, a suggested interface by *Greedy* contains *DetachVolume()*, *AttachVolume()* and *DescribeInstanceAttribute()*. Results show that this design is unlikely to be desirable for developers. Moreover, the obtained results suggest that coupling is as important metric as cohesion to drive Web service interface remodularization.

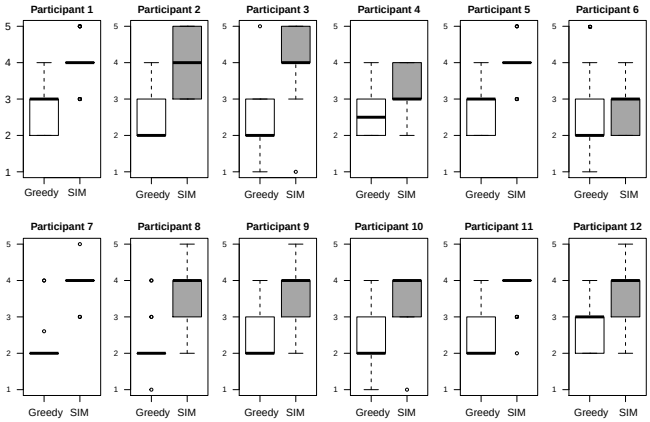


Figure 5: Developer's assessment for the remodularization solutions proposed by *SIM* and *Greedy*.

Table II: Statiscal results of the developers evaluation.

Participant	Average Rating		Statistical tests	
	SIM	Greedy	P-value	Effect size
Participant 1	4	2.68	<0,05	Large (1.874)
Participant 2	4	2.40	<0,05	Large (2.345)
Participant 3	4.18	2.31	<0,05	Large (2.203)
Participant 4	3.04	2.59	<0,05	Medium(0.654)
Participant 5	3.95	2.72	<0,05	Large (1.809)
Participant 6	2.86	2.40	0.085	Medium (0.578)
Participant 7	3.95	2.22	<0,05	Large (3.403)
Participant 8	3.77	2.22	<0,05	Large (2.255)
Participant 9	3.90	2.59	<0,05	Large (1.703)
Participant 10	3.63	2.31	<0,05	Large (1.827)
Participant 11	3.77	2.45	<0,05	Large (2.182)
Participant 12	3.5	2.90	<0,05	Medium (0.707)
Average	**3.71**	**2.48**	**<0,05**	**Large (1.014)**

V. Threats to validity

This section discusses threats to the validity of our study. Threats to external validity 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. Internal threats to validity can be related to the choice of the best configuration parameters, k, $minSize$, $w_{S_{seq}}$, $w_{S_{com}}$ and $w_{S_{sem}}$. Although we used several combinations in order to analyse the influence of each parameter on the obtained results, we are planing to empirically investigate all possible values. Another threat to the internal validity can be related to the knowledge and expertise of the human evaluators. Although we took care to select participants having from 4 to 9 years experience with service-oriented development, we plan to ask more experienced professionals on software quality assessment and software refactoring to provide their expert opinion.

VI. Related work

Much work has been done on automatic approaches for software refactoring to fix bad design and code practices.

In the recent few years, different approaches have proposed to discover design problems and antipatterns in Web services [1], [9], [10], [11], [12]. However fixing these antipatterns is still an unexplored and challenging task. One of the first attempts to address service interface partitioning was by Athanasopoulos et al. [4] (*Greedy*). Although their approach was able to improve cohesion, it is not perfectly adjusted to the developers' needs [4]. Limitations of the approach can be related to the coupling between interfaces which is not considered, and to the conceptual similarity which does not take full advantage of the semantic information embodied in operation names. *SIM* addresses explicitly these two drawbacks to improve the modularization quality.

Most of the related work focus on 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 [8]. Bavota et al. [19], [20] have proposed a similar approach to split a large class into smaller cohesive classes using structural and semantic similarity measures. Fokaefs et al. [21] 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.

VII. Conclusion

In this paper, we proposed an approach, *SIM*, to improve the design quality of Web service interfaces. Our approach aims at automatically partitioning large interfaces with semantically unrelated operations into smaller cohesive interfaces, each representing a distinct abstraction. An empirical study on a benchmark of 22 real-world Web services showed that our approach provides improved service interface modularity over the state-of-the-art approach. Our results show the added value of considering coupling and dedicated semantic similarity measure for automatic remodularization. As future work, we plan to involve clients usage in the remodularazation process, test our approach on additional Web services and refactor other common web service interface antipattern types, e.g., fine-grained Web service, ambiguous Web service, and chatty Web service [1].

Acknowledgment. This work was supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (S) (No.25220003), by Osaka University Program for Promoting International Joint Research.

References

[1] A. Ouni, M. Kessentini, K. Inoue, and M. O Cinneide, "Search-based web service antipatterns detection," *IEEE Transactions on Services Computing*, vol. PP, no. 99, 2015.

[2] 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.

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

[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] 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.

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

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

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

[9] 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.

[10] 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.

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

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

[13] M. Perepletchikov, C. Ryan, K. Frampton, and Z. Tari, "Coupling metrics for predicting maintainability in service-oriented designs," in *18th Australian Software Engineering Conference*, April 2007, pp. 329–340.

[14] D. N. Card and R. L. Glass, *Measuring Software Design Quality*. Prentice-Hall, Inc., 1990.

[15] D. Budgen, *Software Design*. Addision Wesley, 1999.

[16] [Online]. Available: http://sel.ist.osaka-u.ac.jp/people/ali/sim/

[17] 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.

[18] J. Cohen, *Statistical power analysis for the behavioral sciences*. Academic press, 1988.

[19] 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.

[20] 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.

[21] 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.