Web Service Interface Decomposition Using Formal Concept Analysis

Marwa Daagi*, Ali Ouni†, Marouane Kessentini‡, Mohamed Mohsen Gammoudi*, Salah Bouktif*
*ISAMM / Riadi Laboratory-ENSI, University of Mannouba, Mannouba, Tunisia
†Department of Computer Science and Software Engineering, CIT, UAE University, UAE
‡ Department of Information and Computer Science, University of Michigan, MI, USA
{daagi, gammoudi}@fst.rnu.tn, {ouniali, salahb}@uaeu.ac.ae, marouane@umich.edu

***Abstract*—In the service-oriented paradigm, Web service interfaces are considered contracts between Web service subscribers and providers. The structure of service interfaces has an extremely important role to discover, understand, and reuse Web services. However, it has been shown that service developers tend to pay little care to the design of their interfaces. A common design issue that often appears in real-world Web services is that their interfaces lack cohesion, i.e., they expose several operations that are often semantically unrelated. Such a bad design practice may significantly complicate the comprehension and reuse of the services functionalities and lead to several maintenance and evolution problems. In this paper, we propose a new approach for Web service interface decomposition using a Formal Concept Analysis (FCA) framework. The proposed FCA-based approach aims at identifying the hidden relationships among service operations in order to improve the interface modularity and usability. The relationships between operations are based on cohesion measures including semantic, sequential and communicational cohesion. The identified groups of semantically related operations having common properties are used to define new cohesive and loosely coupled service interfaces. We conducted a quantitative and qualitative empirical study to evaluate our approach on a benchmark of 26 real world Web services provided by Amazon and Yahoo. The obtained results show that our approach can significantly improve Web service interface design quality compared to state-of-the-art approaches.**

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

I. Introduction

Service Oriented Architecture (SOA) is commonly materialized through Web Services, i.e., programs with well-defined interfaces that can be published, located and consumed by client applications [1]. SOA promotes software reuse via ready-made, reusable and composable services that are available to end users through their public interfaces. Indeed, deploying successful services highly depends on how well designed are the service interfaces [2], [3], [4].

SOA provides a collection of principles and methodologies for designing and developing service-based systems (SBSs). Developers may see Web services simply as a better mechanism to enable interoperation between clients and server-based functions, and do not consider in practice the principles of software design also supported by SOA. In particular, such developer's behaviour tends to ignore the general structuring principles regarding well-defined services, abstraction, modularity, and many more Quality of Service (QoS) requirements. This practice results in a "hodge-podge" of unrelated operations in a single Web Service interface [3]. Indeed, recent studies found that developers seem to take little care of the structure of their WSDL documents [5].

A common bad design practice that often appears in real-world services is that their interfaces expose a large number of semantically unrelated operations with low interface cohesion [6], [7], [4], [8]. As a consequence, service interfaces tend to cover a lot of different abstractions and processes, leading to many operations associated with each abstraction. At the extreme, the result will be a single Web service that vends all external functions, with which all clients interact. From a superficial perspective, this could work and it could be a possible design. However, as the interface details of the Web service inevitably change over time, when a single Web service implements a lot of operations, many clients will be affected by the change. Furthermore, reusing badly designed service interfaces will result in poorly designed client systems that are hard to comprehend, change, and maintain, leading to unsuccessful services. [3].

Most of existing works on Web service quality assurance focus on several aspects of QoS, but only few are focusing on web service design's quality. One of these works, carried by Athanasopoulos et al. [8] addressed the problem of service interface structure, where a greedy approach driven by a set of cohesion metrics was proposed to split service interfaces based on structural and conceptual relationships between operations. Recently, Ouni et al. [9], proposed an approach to split large service interfaces into smaller and cohesive interfaces using a graph partitioning technique, where nodes represent the operations of a Web Service, while the arcs capture the similarity among them. The main limitation of these approaches is that some hidden relationships between the operations properties cannot be captured through simple greedy or graph based approach. Moreover, coupling between interfaces is not considered, which results into undesirable interface splits and highly coupled interfaces, severely limiting service reusability.

To address this problem, we propose in this paper a novel approach using Formal Concept Analysis (FCA) [10] to find

978-1-5386-0752-7/17 $31.00 © 2017 IEEE
DOI 10.1109/ICWS.2017.30

the optimal decomposition of Web service interfaces that improve its design quality and usability. FCA is a principled way for identifying objects sharing similar properties based on well developed mathematical foundations [11]. Furthermore, the relationships between concepts are semantically very important. These concepts could be structured based on a Galois Lattice to express these relationships as a concept hierarchy from a collection of objects and their properties [11]. Each concept represents the set of objects, e.g., service operations, sharing the same properties. Our FCA-based approach uses a set of measures for similarity among the operations of a Web service. This similarity is captured by metrics representing three cohesion categories, namey semantic, sequential and communicational cohesion.

To evaluate our approach, we conducted an empirical study on a benchmark of 26 Web services provided by Yahoo and Amazon. As a first part of our experiment, we evaluated how well our FCA-based approach can improve the structure of service interfaces comparing to recent state-of-the-art approaches [8], and [9] that use traditional cohesion-based clustering techniques to improve the modularity of service interfaces. The second part of our experiment aimed at assessing the usefulness of our approach in a user study, where 14 expert developers are involved to evaluate the quality of the produced interfaces for each Web service. Overall, the results show that (*i*) FCA achieved interesting decomposition results by a significant improvement of the service interface modularity, (*ii*) our approach provides useful decomposition solutions for developers to improve service understandability and reusability.

The rest of the paper is structured as follows. Section II provides the necessary background, while section III decribes a real-world example to illustrate our approach. Section IV describes our FCA-based approach for service interface decomposition. Section V presents the empirical study and discusses the results. The related work is discussed in Section VI. Finally, Section VII, concludes and discusses the future work.

II. Background

A. Web service interface design

A *Web service interface* corresponds to a port type, which is the most important WSDL (web service description language) element. A Web service has at least one interface which consists of a collection of operations that can be performed to accomplish a specific functionality. A service interface represents the main source of interaction between the service and their subscribers. It serves not only as a communication media between the client and the service, but also as a specification of the service functionality [12]. It acts also as a hinge between the business processes and the underlying software services.

The quality and design structure of the service interface is a good indicator of the quality and complexity of the service itself. The interface description is also a good indicator of how the service may be used and evolved. Thus, interfaces need to be constructed in a modular, flexible and testable fashion. However, different Web service antipatterns have emerged which result from bad design and/or implementation practices. The most common antipatterns are:

- *God object Web service Interface (GOWS)*: implements a high number of operations related to different business and technical abstractions in a single service [3].
- *Fine grained Web service Interface (FGWS)*: is a too fine-grained service whose overhead (communications, maintenance, and so on) outweighs its utility [3].
- *Chatty Web service Interface (CWS)*: represents an antipattern where a high number of operations is required to complete one abstraction [13], [4].
- *Data Web service Interface (DWS)*: contains mostly accessor operations. In a distributed environment, some Web services may only perform some simple information retrieval or data access operations [14].
- *CRUDy Interface Interface (CI)*: is an antipattern where the design encourages services to follow a RPC-like behavior by declaring create, read, update, and delete (CRUD) operations, e.g., createX(), readY(), etc. [3].

B. Formal Concept Analysis (FCA)

Formal Concept Analysis (FCA) is a theoretical framework which structures a set of objects based on the properties that they share [10]. FCA is a method mainly used for the analysis of data, i.e., for deriving implicit relationships between objects described through a set of attributes on the one hand and these attributes on the other hand. The data are structured into units which are formal abstractions of concepts of human thought, allowing meaningful comprehensible interpretation [11]. Thus, FCA can be seen as a conceptual clustering technique as it also provides intensional descriptions for the abstract concepts or data units it produces. Central to FCA is the notion of a formal context:

Definition 1 (Formal Context). A triple (G, M, I) is called a formal context if G and M are sets and $I \subseteq G \times M$ is a binary relation between G and M. The elements of G are called objects, those of M attributes and I is the incidence of the context. For $A \subseteq G$, we define: $A' := m \in M | \forall g \in A : (g, m) \in I$ and dually for $B \subseteq M : B' := g \in G | \forall m \in B : (g, m) \in I$. Intuitively speaking, A' is the set of all attributes common to the objects of A, while B' is the set of all objects that have all attributes in B. Furthermore, we define what a formal concept is:

Definition 2 (Formal Concept). A pair (A, B) is a formal concept of (G, M, I) if and only if $A \subseteq G, B \subseteq M, A' = B$ and $A = B'$. In other words, (A, B) is a formal concept if the set of all attributes shared by the objects of A is identical with B and on the other hand A is also the set of all objects that have all attributes in B. A is then called the extent

and B the intent of the formal concept (A, B). The formal concepts of a given context are naturally ordered by the subconcept-superconcept relation as defined by: $(A_1, B_1) \leq (A_2, B_2) \Leftrightarrow A_1 \subseteq A_2 (\Leftrightarrow B_2 \subseteq B_1)$. Thus, formal concepts are partially ordered with regard to inclusion of their extents or (which is equivalent) inverse inclusion of their intent.

III. Running Example

To better understand the problem of Web service interface decomposition and how our approach works to solve it, we consider a real world example as shown Figure 1. Our running example is the Web service Amazon Virtual Private Cloud (`AmazonVPC`)[1] provided by Amazon, a major service provider. The service interface exposes 21 operations, grouped in a single interface offering a variety of business abstractions including set up private cloud within the Amazon cloud computing service, control virtual networking environment, configuration of route tables and network gateways, launch instances into a publicly accessible subnet, control buckets, requests, users, or groups, and so on.

The current service design seems a version of putting too much into a single component and not creating separate, distinct abstractions. This creates a broad coupling between a wide range of server functions and all the clients that use the different parts, severely limiting reuse. Moreover, the interface description is a good indicator of how the service may evolve. Any change to the interface will affect both the business process and services using it. Indeed, web service interfaces need to fulfill several quality requirements in addition to their functional requirements.

To provide a best practice of third-party reuse, an appropriate design of the service interface should be provided. Thus, developers are encouraged to appropriately design their service interfaces. Motivated by the above mentioned issues, the goal of this paper is to provide an automated technique to find an appropriate decomposition of a large service interface into coarse-grained and cohesive interfaces implementing distinct abstractions.

IV. Approach: Web Service Interface Decomposition Using FCA

In this section, we describe our approach for Web service interface decomposition using Formal Concept Analysis.

Figure 2 depicts the general structure of our approach. To decompose a Web service, our approach consists of four main steps: (1) parsing the WSDL document, (2) computing the similarity/dependency between all operations of a Web service, (3) building the formal context, and (4) generating the formal concepts. The generated formal concepts consist of groups of operations sharing common properties and will be used to define new interfaces. In the following we describe each of the four steps.

[1]https://aws.amazon.com/vpc/

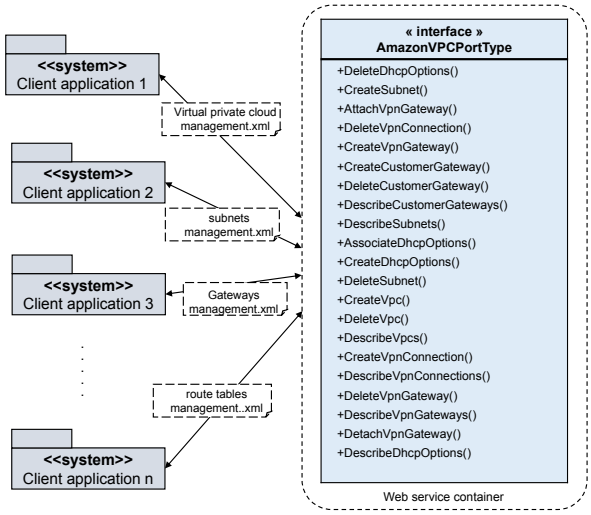

Figure 1: Running Example: Amazon Virtual Private Cloud (Amazon VPC).

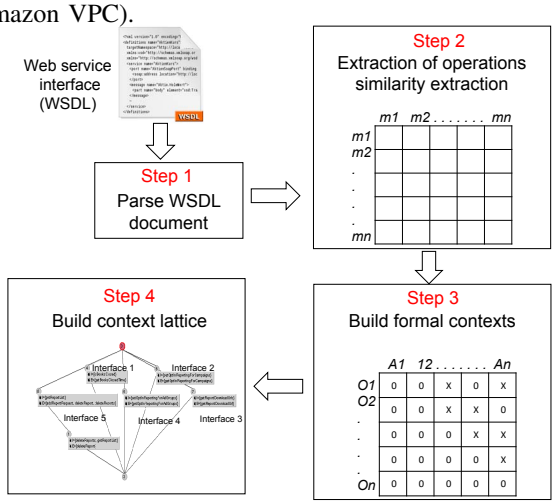

Figure 2: Approach overview.

A. Step 1: Parse the WSDL document

Our approach takes as input a Web service interface as a WSDL file/url to be decomposed. Then, it parses the WSDL source by tree walking up the XML hierarchy. The parser allows to extract all the WSDL elements including the port types, operations signatures, messages, types, and bindings, etc. All these information will be used to identify the relationships between all operations.

B. Step 2: Operations similarity extraction

Once all data are extracted from the WSDL file, our approach identifies the different relationships between all operations. To do so, we use three existing and commonly used cohesion metrics namely sequential (S_{seq}), communicational (S_{com}), and semantic (S_{sem}) cohesion [7], [9], [8]. These cohesion metrics reflect different similarity measures and different properties that a set of operations may share. Then, the overall similarity between two operations op_i and

	CreateSubnet()	AssociateDhcpOptions()	AttachVpnGateway()	CreateCustomerGateway()	DeleteCustomerGateway()	DeleteDhcpOptions()	DeleteSubnet()	DeleteVpc()	DeleteVpnConnection()	DeleteVpnGateway()	DescribeCustomerGateways	....	DetachVpnGateway()
CreateSubnet()	1	0.01	0.03	0.21	0.00	0.00	0.27	0.01	0.00	0.00	0.01		0.01
AssociateDhcpOptions()	0.01	1	0.02	0.00	0.02	0.43	0.02	0.03	0.02	0.02	0.00		0.02
AttachVpnGateway()	0.03	0.02	1	0.17	0.17	0.01	0.01	0.02	0.17	0.41	0.17		0.43
CreateCustomerGateway()	0.21	0.00	0.17	1	0.41	0.00	0.00	0.00	0.00	0.16	0.44		0.16
CreateDhcpOptions()	0.20	0.41	0.00	0.16	0.00	0.41	0.00	0.00	0.00	0.00	0.02		0.00
CreateVpc()	0.25	0.02	0.02	0.21	0.00	0.01	0.00	0.27	0.00	0.00	0.01		0.01
CreateVpnConnection()	0.21	0.00	0.18	0.19	0.01	0.00	0.00	0.00	0.41	0.17	0.03		0.17
CreateVpnGateway()	0.23	0.01	0.42	0.42	0.16	0.00	0.00	0.01	0.16	0.41	0.18		0.41
DeleteCustomerGateway()	0.00	0.02	0.17	0.41	1	0.18	0.22	0.22	0.18	0.40	0.41		0.18
DeleteDhcpOptions()	0.00	0.43	0.01	0.00	0.18	1	0.22	0.22	0.18	0.18	0.00		0.02
DeleteSubnet()	0.27	0.02	0.01	0.00	0.22	0.22	1	0.26	0.22	0.22	0.00		0.02
DeleteVpc()	0.01	0.03	0.02	0.00	0.22	0.22	0.26	1	0.22	0.22	0.00		0.03
DeleteVpnConnection()	0.00	0.02	0.17	0.00	0.18	0.18	0.22	0.22	1	0.40	0.00		0.18
DeleteVpnGateway()	0.00	0.02	0.41	0.16	0.40	0.18	0.22	0.22	0.40	1	0.16		0.43
DescribeCustomerGateways()	0.01	0.00	0.17	0.44	0.41	0.00	0.00	0.00	0.00	0.16	1		0.16
⋮													
DetachVpnGateway()	0.01	0.02	0.43	0.16	0.18	0.02	0.02	0.03	0.18	0.43	0.16		1

Figure 3: The operartion-by-operation matrix obtained for AmazonVPC.

op_j is a weighted sum of the three similarity measures as given by Equation 1.

$$Sim(op_i, op_i) = \alpha \times S_{seq}(op_i, op_i) + \beta \times S_{com}(op_i, op_i) + \gamma \times S_{sem}(op_i, op_i) \quad (1)$$

where $\alpha + \beta + \gamma = 1$.

The similarity measure will be calculated for each pair of operations in an interface, to generate an operation-by-operation matrix. A similarity measure $Sim : G \times G \rightarrow [0, 1]$ can be defined as follows:

$\forall op_i \in G \Rightarrow Sim(op_i, op_i) = 1$ (an operation with itself)

$\forall op_i, op_j \in G, i \neq j \Rightarrow Sim(op_i, op_j) \in [0, 1]$

The calculated similarity values can be presented by a symmetric square matrix, as shown in Figure 3 for our running example, the AmazonVPC interface. The matrix is of size $n = |G|$, where its diagonal elements are all equal to 1 (similarity of an operation with itself). For example, the operations *CreateCustomerGateway()* and *DeleteCustomerGateway()* have high similarity measure as they operate on the same data types captured through the sequential and communicational similarity, as well as a high semantic similarity as they share several tokens in their signatures. This similarity measure, will then drive the decomposition process to group together operations with high similarity.

C. Step 3: Formal context building

After generating an operation-by-operation matrix for a given interface si, we construct our formal context using FCA. In our FCA model, both the set of objects and attributes are the operations of the service interface. In this way, a formal context of Web service operations becomes $K = (G, M, I)$, where $G = M = \{op_i \mid 2 \leqslant n_K, n_K > 2\}$ is the set of interface operations. We suppose that a Web service must contain more than two operations to be decomposed.

We define the binary relation I as the maximum similarity value between an operation op_i from the objects set and the rest of operations from the attributes set $\forall op_j \in M$.

	CreateSubnet()	AssociateDhcpOptions()	AttachVpnGateway()	CreateCustomerGateway()	DeleteCustomerGateway()	DeleteDhcpOptions()	DeleteSubnet()	DeleteVpc()	DeleteVpnConnection()	DeleteVpnGateway()	DescribeCustomerGateways	....	DetachVpnGateway()
CreateSubnet()	0	0	0	0	0	0	0	0	0	0	0		0
AssociateDhcpOptions()	0	0	0	0	0	1	0	0	0	0	0		0
AttachVpnGateway()	0	0	0	0	0	0	0	0	0	0	0		1
CreateCustomerGateway()	0	0	0	0	1	0	0	0	0	0	1		0
CreateDhcpOptions()	0	0	0	0	0	0	0	0	0	0	0		0
CreateVpc()	0	0	0	0	0	0	0	0	0	0	0		0
CreateVpnConnection()	0	0	0	0	0	0	0	0	1	0	0		0
CreateVpnGateway()	0	0	0	0	0	0	0	0	0	0	0		0
DeleteCustomerGateway()	0	0	0	1	0	0	0	0	0	0	1		0
DeleteDhcpOptions()	0	1	0	0	0	0	0	0	0	0	0		0
DeleteSubnet()	0	0	0	0	0	0	0	0	0	0	0		0
DeleteVpc()	0	0	0	0	0	0	0	0	0	0	0		0
DeleteVpnConnection()	0	0	0	0	0	0	0	0	0	0	0		0
DeleteVpnGateway()	0	0	0	0	0	0	0	0	0	0	0		1
DescribeCustomerGateways()	0	0	0	1	1	0	0	0	0	0	0		0
⋮													
DetachVpnGateway()	0	0	1	0	0	0	0	0	0	1	0		0

Figure 4: The formal context obtained for AmazonVPC.

Formal concepts naturally endow "cohesiveness" because their extents comprise members sharing all the properties in the respective intents. This means that the identified concepts, i.e., groups of service operations, exhibit the highest cohesion value. Thus, FCA allows identifying highly cohesive sets that could jointly replace the AmazonVPC large interface and hence improve the overall modularity.

For instance, the table reported in Figure 4 illustrates a binary context derived from our running example AmazonVPC where both the objects and the attributes are the service operations. The binary relation I is given by the cross table and describes which operation(s) op_i from the attributes is(are) more cohesive to op_j from the objects. For example, the operation *CreateCustomerGateway()* has a binary relation with both operations *DeleteCustomerGateway()* and *DescribeCustomerGateways()* which means the three operations will form a formal concept.

D. Step 4: Context Lattice Generation

After building our formal context, we rake either the superior or inferior part around the diagonal of our symmetric matrix to generate a context lattice. Our approach identifies the set of all formal concepts relative to the given context. Then, the identified sets of concepts represent the new partitioning of the service interface, in such a way that each concept corresponds to a new interface. To reduce the redundancies and similarities in the generated concepts, our approach identifies all concepts C_i and C_j where there exists an operation $op \in C_i.intension$ and at the same time, $op \in C_j.extension$, and merge them into a single concept.

Figure 5 shows an example of the generated Galois lattice for AmazonVPC. As can be seen in the lattice, we have C_2 = {*DescribeVpnGateways, DeleteVpnGateway, AttachVpnGateway*}{*DetachVpnGateway*} and C_7 ={*CreateVpnGateway*}{*DescribeVpnGateways*}. We note that the operation *DescribeVpnGateways* belongs to two different concepts: $C_7.intension$ and $C_2.extension$. To avoid operations redundancy, we merge these two concepts

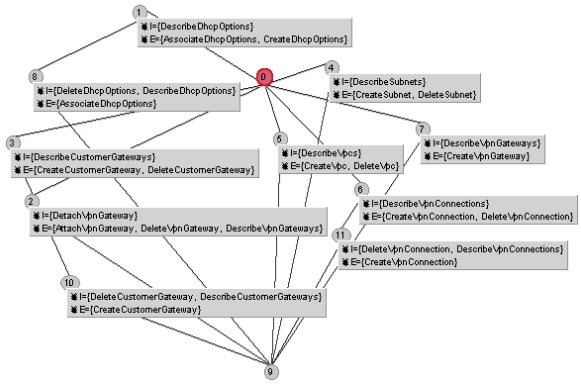

Figure 5: The formal context generated for our running example `AmazonVPC`.

using the formulas: $C_2.extension \cup C_7.intension$ and $C_2.intension \cup C_7.extension$ to obtain a new concept C_{27} ={*DescribeVpnGateways, DeleteVpnGateway, AttachVpnGateway*}{*DetachVpnGateway, CreateVpnGateway*}.

To address the problem of operations redundancy within the set of obtained concepts, provided with the specialization order, we proceed to merge all concepts where $C_i \leq_s C_j$. To this end, we used the following formulas: $C_i.extension \cup C_j.extension$ and $C_i.intension \cup C_j.intension$. As an example from the AmazonVPC lattice in Figure 5, we have $C_1 \leq_s C_8$. These two concepts should be merged according to our formula. We thus obtain a new lattice where C_1 and C_8 are merged into a new concept C_{18}={*AssociateDhcpOptions, CreateDhcpOptions*}{*DescribeDhcpOption, DeleteDhcpOption*}.

V. Empirical Evaluation

This section reports our empirical study to evaluate the efficiency of our Web services interface decomposition approach. We first present our research questions and then describe and discuss the obtained results.

A. Research questions

We designed our experimental study to address two research questions about the applicability and performance of our approach in comparison to existing Web services remodularization approaches [8] and [9], and the usefulness in practice from a developer point of view. The two research questions are as follows:

- **RQ1.** To what extent can our FCA-based approach identify relevant Web services interface decomposition solutions?
- **RQ2.** To what extent can our approach be useful in practice by Web services developers?

B. Evaluation Measures and Procedure

To evaluate our approach, we conducted our experiment on a benchmark of 26 real-world services provided by Amazon[2] and Yahoo[3]. We selected services with interfaces exposing from 7 to 87 operations. We chose these Web services because their WSDL interfaces are publicly available, and they were previously studied in the literature [15], [8], [9]. Our benchmark is summarized in Table I, and publicly available[4] for future replications.

TABLE I: Experimental benchmark overview.

Service interface	Provider	ID	#operations
AutoScalingPortType	Amazon	I1	13
MechanicalTurkRequesterPortType	Amazon	I2	27
AmazonFPSPorttype	Amazon	I3	27
AmazonRDSv2PortType	Amazon	I4	23
AmazonVPCPortType	Amazon	I5	21
AmazonFWSInboundPortType	Amazon	I6	18
AmazonS3	Amazon	I7	16
AmazonSNSPortType	Amazon	I8	13
ElasticLoadBalancingPortType	Amazon	I9	13
AmazonEC2PortType	Amazon	I10	87
MessageQueue	Amazon	I11	13
AmazonSimpleDB	Amazon	I12	11
CloudWatch	Amazon	I13	11
CommerceService	Amazon	I14	9
MapReduce	Amazon	I15	7
KeywordService	Yahoo	I16	34
AdGroupService	Yahoo	I17	28
UserManagementService	Yahoo	I18	28
TargetingService	Yahoo	I19	23
AccountService	Yahoo	I20	20
AdService	Yahoo	I21	20
CompaignService	Yahoo	I22	19
BasicReportService	Yahoo	I23	12
TargetingConverterService	Yahoo	I24	12
ExcludedWordsService	Yahoo	I25	10
GeographicalDictionaryService	Yahoo	I26	10

To answer **RQ1**, we assess the design improvement resulted by service decompositions identified by our approach in comparison to recent state-of-the-art Web services decomposition approaches *Greedy* [8] and *SIM* [9]. The approach *Greedy* [8] uses a greedy algorithm to split service interfaces based on cohesion metrics, while *SIM* [9] is based on a graph partitioning approach to find connected sub-groups of operations. We use mainly two evaluation metrics: (*i*) modularization quality, and (*ii*) number of antipatterns:

(i) Modularization Quality Improvement (MQI): The Modularization Quality (MQ) metric was originally proposed by Mancoridis et al. [16] to guide the allocation of classes to highly cohesive and loosely coupled packages. The MQ metric was re-formulated over the years, and its most recent incarnation is described in [17]. To evaluate our approach, we adapted MQ in the context of Web service interfaces, by building an analogy between a *package* in the original formulation and a *service interface* in our context.

MQ consists of assigning scores to each interface in the Web service, measuring the interfaces' individual trade-off between cohesion and coupling. The cohesion of an interface refers to the average similarity values between all its operations using Equation 1, while the coupling between two interfaces si_1 and si_2 refers to the similarity between each pair of operations $Sim(op_i, op_j)$ where $op_i \in si_1$ and

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

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

[4]https://github.com/ouniali/AmazonYahooBenchmark

$op_j \in si_2$. The aim of the *MQ* metric is to reward increased cohesion with a higher *MQ* score and to punish increased coupling with a lower *MQ* score. The higher the value of *MQ*, the better the quality of the modularization. The *MQ* value of the overall Web service is computed as follows:

$$MQ = \sum_{i=1}^{n} MF(si_i) \tag{2}$$

$$\text{where, } MF(si_i) = \begin{cases} 0, & \text{if coh}(si_i) = 0 \\ \frac{coh(si_i)}{coh(si_i) + \frac{cop(si_i)}{2}}, & \text{if coh}(si_i) > 0 \end{cases} \tag{3}$$

The MQ is thus given by the sum of the Modularization Factors (MF) of each interface si_i in the Web service. $MF(si_i)$ represents the trade-off between cohesion (coh) and coupling (cop) for interface si_i. Since the similarities involved in the measurement of the interfaces' coupling will be double counted during MQ computation, $cop(si_i)$ is divided by 2. Then, the modularization quality improvement (*MQI*) of a Web service corresponds to the percentage increase of MQ. *MQI* is calculated as follows:

$$MQI = \frac{MQ_{dec} - MQ_{ini}}{MQ_{ini}} \times 100 \tag{4}$$

where MQ_{dec} and MQ_{ini} are the MQ scores before and after decomposition, respectively.

(ii) Number of antipatterns (NAP): This metric refers to the number of design antipatterns in the new interfaces. Indeed, a modularity improvement should not compromise the overall design as many design antipatterns may appear in the new generated interfaces such as fine-grained, chatty or CRUDy interfaces (cf. Section II-A). The detection of design defects is performed using the detection rules of [4].

$$NAP(ws) = \sum_{\forall si_i \in ws} antipattern(si_i) \tag{5}$$

where the function $antipattern(si_i)$ returns the number of antipattern instances in the interface si_i using dedicated detection rules [4]. Note that the same interface could contain more than one antipatterns as some antipatterns tends to co-occur, e.g., chatty interface (*CWS*) and data interface (*DWS*).

To answer **RQ2**, we asked our group of 18 independent developers to manually decompose each of the studied Web service interfaces(cf. Table I) in order to improve their interface readability and understandability. The involved developers include 12 master students in Software Engineering and 8 Ph.D. students in Software Engineering from the University of Michigan. All the participants are volunteers and familiar with Web services and refactoring in general. The experience of these participants on programming ranged from 2 to 19 years. 11 out of the 18 participants are currently active programmers in software industry. The manually identified interfaces by the participants were considered as the ground truth, allowing the calculation of the precision and recall of

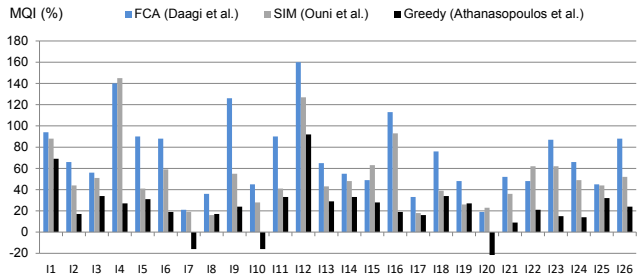

Figure 6: The Modularization improvement (MQI) results achieved by each of the *FCA*, *SIM* and *Greedy* approaches.

our approach. We compute the precision and recall scores as follows:

$$Precision = \frac{TP}{TP + FP} \tag{6}$$

$$Recall = \frac{TP}{TP + FN} \tag{7}$$

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 contain the same operations.

C. Results

1) Results for RQ1: Figure 6 reports the results for RQ1 in terms of modularization quality improvement (*MQI*). On average, for both Amazon and Yahoo Web services, our FCA-based approach achieved significant *MQI* scores ranging from 19% to 160% with an average of 71% for all the 26 services. This score significantly outperforms both state-of-the-art approaches *Greedy* and *SIM* having an average of 23% and 52%, respectively.

We observe that *Greedy* produced 3 out of 26 services with negative *MQI*, as reported in Figure 6. This is mainly due to the high coupling resulted in the new interfaces, specifically for `AmazonS3` (I7), `AmazonEC2` (I10) and `AccountService` (I20). Overall, we assume that a candidate decomposition is a good design solution if the improvement of cohesion is significantly greater than the deterioration of coupling. This balance is well captured by the *MQI* [17], to reflect the overall improvement.

Furthermore, we evaluate the resulted decompositions in terms of number of antipatterns (NAP). The existence of antipatterns is highly undesirable as it will make the new design hard to understand, to change and reuse [13], [4]. Figure 7, reports the achieved results by each of the FCA, *SIM* and *Greedy* approaches. On average, our FCA-based approach produced 3.2 antipatterns per decomposition for

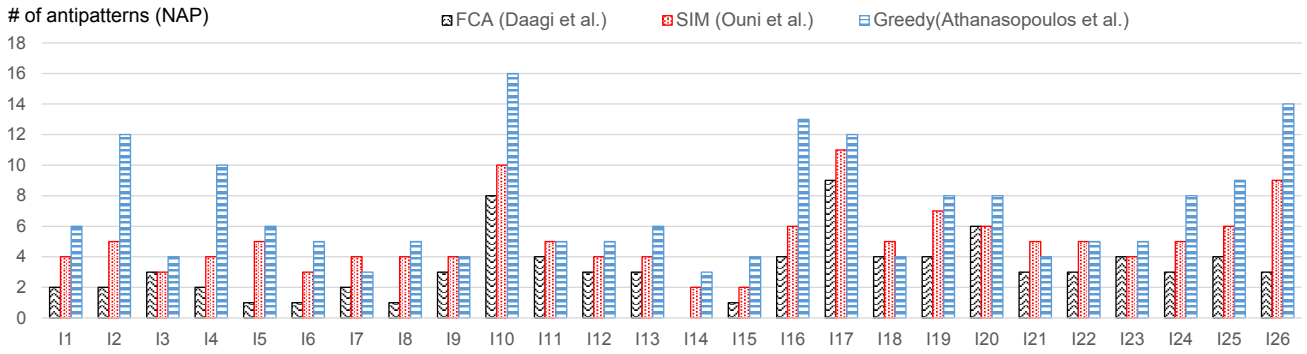

Figure 7: The number of resulted antipatterns (NAP) from each approach, FCA, SIM and Greedy for each Web service.

both Amazon and Yahoo services, while *SIM* and *Greedy* produced a quite higher average of 5.1 and 7.1, respectively. Indeed, this is normal that some antipatterns appear and they are often unavoidable during the design and/or implementation. However, if the number of antipatterns is high, it will result into bad performance and reduced reusability.

For example, some Web service interfaces, such as *AdGroupService* and *AmazonEC2*, have a higher number of antipatterns as they expose a large number of operations that operate on several data types with several CRUDYy and accessor operations.

2) Results for RQ2: Figures 8 and 9 report the results for RQ2 in terms of precision and recall based on the manual decompositions provided by developers. We assume that a useful automated approach should achieve results similar to what developers would like to have. Overall, we found that a considerable number of generated interfaces, with an average of more than 68% of precision on all the 26 Web services, were already suggested manually by the developers, while SIM and greedy achieved only 52% and 27%, respectively.

The achieved recall scores are slightly higher, in average, than the precision ones with a score of 73% achived by our FCA-based approach, while *SIM* and *Greedy* achieved a recall of 59% and 31%, respectively. Indeed, we found that some of the interfaces suggested manually by developers do not exactly match the solutions provided by our approach. In addition, it is worth notice that the slight deviation with the expected interfaces is not always related to incorrect ones but to the fact that slightly different possible decompositions solutions could be optimal.

VI. Related work

In the recent few years, different approaches have proposed to discover design problems and antipatterns in Web services [4], [13], [18], [19], [2], [20], [21]. However fixing these antipatterns is still an unexplored and challenging task. One of the first attempts to address service interface decomposition was by Athanasopoulos et al. [8] (*Greedy*) based on a greedy algorith that iteratively group together a set of operations based on their cohesion. More recently,

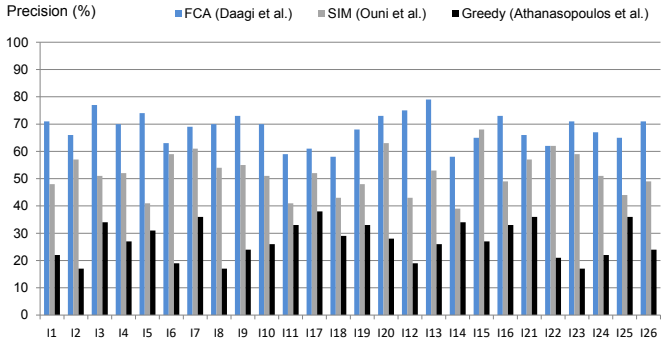

Figure 8: The precision results achieved by each approach, *FCA*, *SIM* and *Greedy* for all Web services.

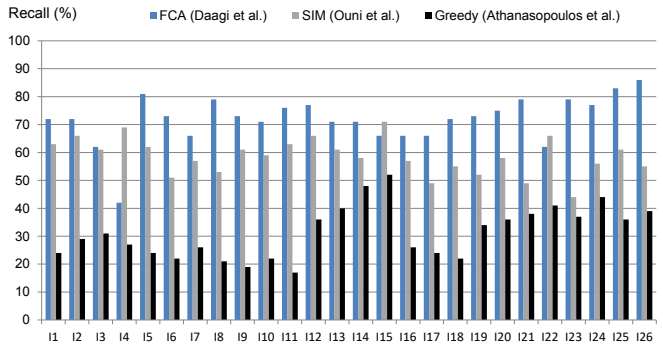

Figure 9: The recall results achieved by each approach, *FCA*, *SIM* and *Greedy* for all Web services.

Ouni et al. [9] have formulated the problem of service interface decomposition as a graph partitioning problem to find strongly related operations. Although both approaches were able to improve cohesion, they resulted into a high number of new interfaces affected by antipatterns. Limitations of both approaches could be related to the coupling between interfaces which is not considered. Our approach attempts to explicitly address these two drawbacks to improve the interface design quality, taking the advantages of FCA.

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 [22]. Moha et al. [23] proposed a refactoring approach using relational concept analysis to fix Blob code smells in OO designs taking the advantage of the strength of FCA in finding cohesive classes. Bavota et al. [24] have proposed a similar approach to split a large class into smaller cohesive classes using structural and semantic similarity measures. Fokaefs et al. [25] proposed an automated extract class refactoring approach using hierarchical clustering algorithm to identify cohesive subsets of methods and attributes. However, the *Extract Class* 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. Conclusions and Future Work

In this paper, we proposed a novel approach for Web service interfaces decomposition towards improving their design quality. Our approach uses FCA and cohesion metrics to decompose Web service interfaces into smaller, cohesive and loosely coupled ones. The proposed approach is validated on 26 real-world Web services, and the obtained results provide evidence that our approach performs significantly better than state-of-the-art techniques in terms of design quality improvement. Moreover, we evaluated our approach with 18 developers since we strongly believe that an automated approach must achieve solutions that fit developer expectations. The obtained results show that our generated refactoring solutions are similar to manually performed refactorings by developers at 68% of precision and 72% of recall. As future works, we plan to extend our work to consider additional Web-services in our validation to generalize our findings. In addition, we are planning to extend our approach to balance between refactorings in the code and interface levels. Furthermore, we will consider the programmer in-the-loop when identifying the refactoring solutions.

Acknowledgment. This Work is supported by the Research Start-up (2) 2016 Grant G00002211 funded by UAE University, and based in part upon support of the National Science Foundation under Grant Number 1661422.

References

[1] M. P. Papazoglou, "Service-oriented computing: Concepts, characteristics and directions," in *International Conference on Web Information Systems Engineering*, 2003, pp. 3–12.

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

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

[4] 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, 2016.

[5] M. Crasso, J. M. Rodriguez, A. Zunino, and M. Campo, "Revising WSDL Documents: Why and How," *IEEE Internet Computing*, vol. 14, no. 5, pp. 48–56, 2010.

[6] 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.

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

[8] 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.

[9] 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, pp. 91–98.

[10] B. Ganter and R. Wille, *Formal concept analysis: mathematical foundations*. Springer Science & Business Media, 2012.

[11] B. Ganter, G. Stumme, and R. Wille, *Formal concept analysis: foundations and applications*. springer, 2005, vol. 3626.

[12] H. M. Sneed, "Measuring web service interfaces," in *12th IEEE International Symposium on Web Systems Evolution (WSE)*, 2010, pp. 111–115.

[13] 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.

[14] 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.

[15] M. Fokaefs, R. Mikhaiel, N. Tsantalis, E. Stroulia, and A. Lau, "An empirical study on web service evolution," in *International Conference on Web Services*, 2011, pp. 49–56.

[16] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, "Using automatic clustering to produce high-level system organizations of source code," in *6th International Workshop on Program Comprehension*, 1998, pp. 45–52.

[17] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, 2006.

[18] 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.

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

[20] H. Wang, M. Kessentini, and A. Ouni, "Bi-level identification of web service defects," in *International Conference on Service-Oriented Computing*. Springer, 2016, pp. 352–368.

[21] H. Wang, A. Ouni, M. Kessentini, B. Maxim, and W. I. Grosky, "Identification of web service refactoring opportunities as a multi-objective problem," in *IEEE International Conference on Web Services (ICWS)*, 2016, pp. 586–593.

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

[23] N. Moha, J. Rezgui, Y.-G. Guéhéneuc, P. Valtchev, and G. El Boussaidi, *Using FCA to Suggest Refactorings to Correct Design Defects*, 2008, pp. 269–275.

[24] 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.

[25] 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.