Fine-grained Metrics of Cohesion Lack for Service Interfaces

Dionysis Athanasopoulos, Apostolos Zarras

Fine-grained Metrics of Cohesion Lack for Service Interfaces

Dionysis Athanasopoulos
Department of Computer Science
University of Ioannina
Ioannina, Greece
dathanas@cs.uoi.gr

Apostolos V. Zarras
Department of Computer Science
University of Ioannina
Ioannina, Greece
zarras@cs.uoi.gr

***Abstract*—A design issue that often appears in real-world services is that their interfaces are not cohesive, i.e., they consist of many and possibly unrelated operations. This issue may complicate the comprehension of the services functionalities and the maintenance of the applications that use them. Currently, the state of the art on cohesion metrics for service interfaces is limited. In particular, there exist coarse-grained metrics of cohesion lack, which consider that the operations of a service interface are related if the types of certain of their input/output data exactly match. The problem in this approach is that operations which operate on data characterized by similar, but not exactly matching, types are treated as being totaly unrelated. Consequently, the aforementioned metrics may overestimate the cohesion lack of service interfaces. In this paper, we undertake a more elaborate approach. Specifically, we propose two fine-grained metrics of cohesion lack, which are defined with respect to the structural similarity of the input/output data types of interface operations. The proposed metrics are formally defined and analytically assessed with respect to fundamental properties of software metrics. Moreover, the usefulness of the metrics in identifying cohesion problems is evaluated in real-world services.**

I. Introduction

The service-oriented architecture (SOA) paradigm aims at facilitating the work of distributed application developers [14], [5]; software reuse is promoted via services that are available to anyone who wishes to compose them towards constructing a novel customized application. Nevertheless, the rapid and low-cost aspects of the overall development process should not be taken for granted. These aspects depend on how well-designed are the services, used for building service-oriented applications. In this paper, we investigate the issue of cohesion. Cohesion was introduced in the early 70s [13] and refers to *the degree to which the elements of a module belong together*. The lack of cohesion leads to poorly designed systems that are hard to comprehend and maintain.

In general, cohesion can be regarded from different perspectives [13], [17]. According to [13], [17], the best possible types of cohesion are functional, sequential and communicational. In a functionally cohesive module, all the elements of the module contribute in a single well defined task. In a sequentially cohesive module, the outputs of one element are used as inputs for other elements. In a communicationally cohesive module, the elements operate on the same data.

Generally, in service interfaces, *cohesion concerns the degree to which the operations of a service interface belong together*. However, since functional cohesion is a purely conceptual notion that can not be quantified, *we specifically focus on the notions of sequential and communicational cohesion*.

Taking a real-world case, Amazon is a major service provider that offers a variety of Web services. Among these services, the Amazon Simple Queue Service (SQS) enables communication via message queues, allocated on the Amazon infrastructure. Figure 1 shows one of the main interfaces of the SQS service, called `MessageQueue`[1]. This interface provides a quite 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.

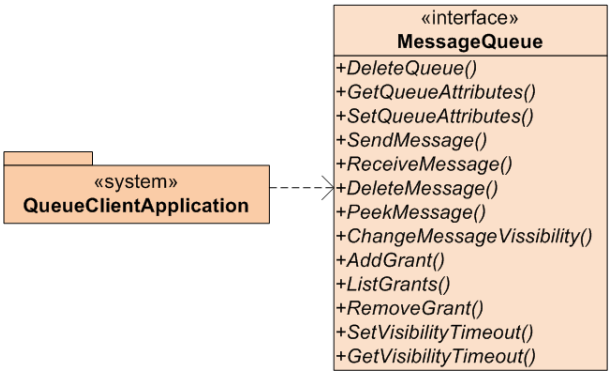

Figure 1. The Amazon `MessageQueue` interface.

The `MessageQueue` interface is not cohesive, in the sense that it includes various functionalities that do not belong together (queue attributes management, access rights management, message exchange operations). Consequently, a developer who aims at building a queue client

[1] http://docs.amazonwebservices.com/AWSSimpleQueueService/2007-05-01/SQSDeveloperGuide/

(`QueueClientApplication` in Figure 1) that communicates with other queue clients through an existing queue is supposed to study the specification of the `MessageQueue` interface (which consists of 838 lines of WSDL and XML schema definitions) and a 49 pages API reference guide. Nevertheless, amongst the various operations offered by the `MessageQueue` interface, only 4 are actually related to the exchange of messages. A more cohesive decomposition of the provided operations into separate interfaces that relate to the management of queue attributes, the management of access rights and the exchange of messages would simplify the comprehension of the functionalities that the developer actually needs. Regarding the queue client application, we may further consider maintenance scenarios that involve new releases of the `MessageQueue` interface. Such scenarios may be quite frequent. Specifically, since 2006 there have been various different releases of `MessageQueue`, which were not always backwards compatible. In the 2008 release, the access rights management operations (`ListGrants()`, `AddGrant()`, `RemoveGrant()`) were removed from the interface. Dealing with such situations, amounts to reasoning about whether the changes from one release to the other actually affect the queue client application. Again, a more cohesive decomposition of the provided operations that groups the message exchange operations into a separate interface would simplify the aforementioned reasoning.

Despite the widely recognized importance of cohesion as a general design property of software, the state of the art on cohesion metrics in the context of the SOA paradigm is quite limited. Specifically, for sequential and communicational cohesion, there exist coarse-grained metrics of cohesion lack, which consider that the operations of a service interface are related if the types of certain of their input/output data exactly match [10], [11], [12]. A limitation of these metrics is that operations which operate on data characterized by similar, but not exactly matching, types are treated as being totaly unrelated. Such cases of operations are frequent in real world services. For instance, in our example, several operations of the `MessageQueue` interface use similar but not exactly matching data types (e.g., see Figure 2 for the input/output data of the `GetQueueAttributes()` and the `SetQueueAttributes()` operations). Consequently, the state of the art metrics for sequential and communicational cohesion may overestimate the cohesion lack of service interfaces.

Based on the previous discussion, in this paper we propose two fine-grained metrics of cohesion lack, which are defined with respect to the structural similarity of the input/output data types of interface operations. The proposed metrics are formally defined and analytically assessed with respect to fundamental properties of software metrics. Moreover, the usefulness of the metrics in identifying cohesion problems is evaluated in real-world services.

The rest of the paper is structured as follows: Section 2 defines formally the proposed metrics and validates them with respect to fundamental software metrics properties. Section 3 details the methodology of the evaluation and analyzes the experimental results. Section 4 focuses on related work. Finally, Section 5, summarizes the contribution of the paper and discusses the future work.

II. Cohesion of Service Interfaces

Our overall approach for the definition of the proposed metrics is based on a generic conceptual model for services, derived from the *W3C* standard services architecture[2]. According to this model, a service is characterized by a name and provides a set of *interfaces* (Table I(1, 2)). An interface is characterized by a name and a set of *operations* (Table I(3, 4)). An operation corresponds to a particular functionality; its execution requires at most one *input message* and produces at most one *output message* (Table I(5)). A message is modeled as an unordered rooted tree (Table I(6-9)). The tree root represents the message. The non-leaf vertices correspond to *complex elements*, i.e., elements characterized by a name and a complex XML type, which consists of further constituent elements. The leaves of the tree represent *primitive elements*, i.e., elements characterized by a name and a XML build-in type.

Table I
Service model.

$$Service = (name : string, I) \quad (1)$$
$$I = \{si : Interface\} \quad (2)$$
$$Interface = (name : string, O) \quad (3)$$
$$O = \{op : Operation\} \quad (4)$$
$$Operation = (name : string, in : Message, out : Message) \quad (5)$$
$$Message = (V, E) \quad (6)$$
$$V = \{v : Element\} \quad (7)$$
$$Element = (name : string, type : anyType) \quad (8)$$
$$E = \{(v_i, v_j) \in V \times V | i \neq j\} \quad (9)$$

A. Communicational & Sequential Cohesion Metrics

The fundamental notions of communicational and sequential cohesion can be adapted in the case of services as follows.

Definition 1: Sequential cohesion: An interface $si \in s.I$ of a service s is sequentially cohesive to some extent, if it includes pairs of operations, op_i, op_j, such that the input message of op_j (resp. op_i) and the output message of op_i (resp. op_j) comprise common (complex or primitive) elements. More specifically, for complex elements the term common refers to elements characterized by the same complex XML type, while for primitive elements the term

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

common refers to elements, characterized by the same name and build-in type. In this case, the operations op_i, op_j are *sequentially related*, in the sense that certain output data produced by op_i (resp. op_j) may be used as input for op_j (resp. op_i).

Definition 2: Communicational cohesion: An interface $si \in s.I$ of a service s is communicationally cohesive to some extent, if it includes pairs of operations op_i, op_j, such that their input messages and/or their output messages comprise common (complex or primitive) elements. In this case, the operations are *communicationally related*, in the sense that they may use similar input data and/or produce similar output data.

The common elements of messages play an important role in both of the previously defined types of cohesion. The following definition reflects the *similarity of messages* more formally.

Definition 3: Message similarity: Let S_{m_i,m_j} be the set of the common elements of two messages m_i, m_j. According to our model, each $t_i \in S_{m_i,m_j}$ is a common bottom-up subtree of m_i and m_j[3]. In the trivial case, where $|t_i.V| = 1$, t_i corresponds to a common primitive element, otherwise t_i corresponds to a common complex element. Then, the similarity $MS(m_i, m_j)$ of m_i, m_j is defined as the sum of the orders of the common bottom-up subtrees of S_{m_i,m_j}, divided by the order of the message that results from the union of m_i and m_j (Table II(1)).

Table II
DEFINITIONS OF METRICS.

$$MS(m_i, m_j) = \frac{\sum_{\forall t_i \in S_{m_i,m_j}} |t_i.V|}{|m_i.V \cup m_j.V|} \quad (1)$$

$$OpS_{seq}(op_i, op_j) = \frac{MS(op_i.in, op_j.out)}{2} + \frac{MS(op_i.out, op_j.in)}{2} \quad (2)$$

$$OpS_{com}(op_i, op_j) = \frac{MS(op_i.in, op_j.in)}{2} + \frac{MS(op_i.out, op_j.out)}{2} \quad (3)$$

$$C_{si} = \{(op_i, op_j) \in si.O \times si.O | (op_i \neq op_j) \wedge (op_j, op_i) \notin C_{si}\} \quad (4)$$

$$LoC_S(si) = 1 - \frac{\sum_{\forall (op_i,op_j) \in C_{si}} OpS_{seq}(op_i, op_j)}{\frac{|si.O|*(|si.O|-1)}{2}} \quad (5)$$

$$LoC_C(si) = 1 - \frac{\sum_{\forall (op_i,op_j) \in C_{si}} OpS_{com}(op_i, op_j)}{\frac{|si.O|*(|si.O|-1)}{2}} \quad (6)$$

By definition, the values $MS(m_i, m_j)$ range from 0 to 1; $MS(m_i, m_j) = 1$ if the messages *exactly match*, while $MS(m_i, m_j) = 0$ if the messages are completely unrelated. The similarity between two messages increases with the number of the bottom-up subtrees that they have in common and the orders of these subtrees. Technically, the specification of every service includes references to XML schemas that contain the definitions of the complex XML data types used in the definitions of the interfaces provided by the service. These complex XML data types constitute the set of candidate bottom-up subtrees that may be common in two messages. In this way, constructing the set S_{m_i,m_j} amounts to checking whether a complex XML data type is used in m_i and m_j.

Nevertheless, when constructing S_{m_i,m_j} special attention must be paid to certain common elements. In practice, the operations of many service interfaces use input messages and/or produce output messages that have common bottom-up subtrees, which correspond to generic meta-data elements. These elements are not related to the particular functionality of the operations and may result in misleading values of cohesion lack for the service interfaces. Dealing with this issue, when measuring the value of $MS(m_i, m_j)$ amounts to filtering out from S_{m_i,m_j} the common bottom-up subtrees that correspond to generic meta-data elements. Apparently, excluding the aforementioned subtrees from $MS(m_i, m_j)$ requires user intervention so as to identify the generic meta-data elements. However, this task is quite simple; the generic meta-data elements can be easily identified, since they are included in all input, or in all output messages.

Based on message similarity, we can define the similarity of operations. In particular, OpS_{seq} reflects the similarity of operations from the perspective of sequential cohesion, while OpS_{com} reflects the similarity of operations from the perspective of communicational cohesion.

Definition 4: Sequential similarity: The sequential similarity between two operations $op_i, op_j \in si.O$ of an interface si is defined as (Table II(2)) the average of:

1) the similarity of the input message of op_i and the output message of op_j and
2) the similarity of the output message of op_i and the input message of op_j.

Definition 5: Communicational similarity: The communicational similarity between two operations $op_i, op_j \in si.O$ of an interface si is defined as (Table II(3)) the average of:

1) the similarity of the input messages of op_i and op_j and
2) the similarity of the output messages of op_i and op_j.

Based on the previous notions of similarity between operations, we define the proposed metrics of cohesion lack for service interfaces. In particular, LoC_S evaluates sequential cohesion (i.e., whether interfaces consist of sequentially similar operations), while LoC_C evaluates communicational cohesion (i.e., whether interfaces consist of communicationally similar operations).

[3]In general, a tree t is a bottom-up subtree of t' if the root v of t is a vertex of t' and the rest of the vertices of t are the descendants of v in t' [15]

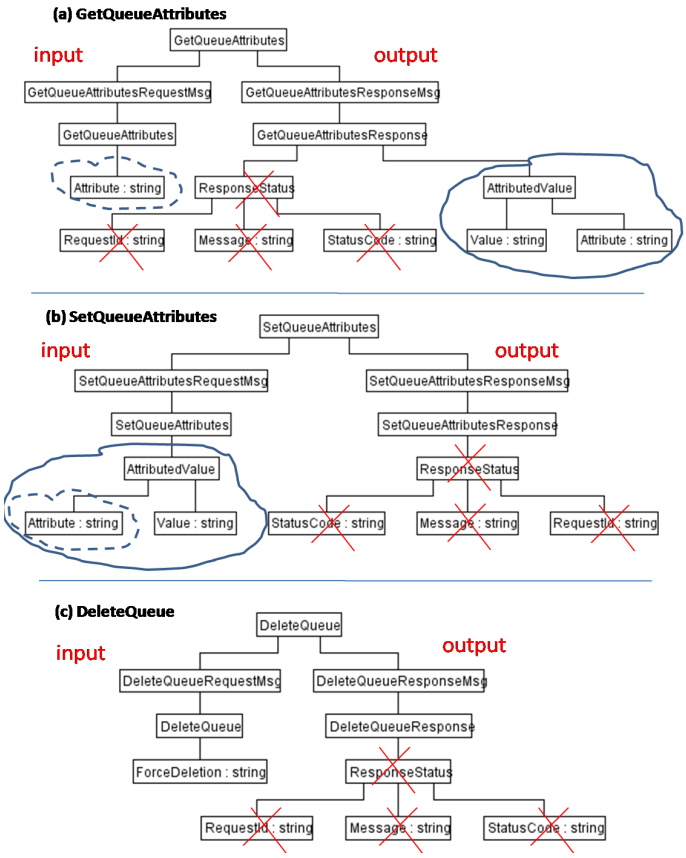

Figure 2. Examples of `MessageQueue` operations.

Definition 6: Lack of cohesion metrics: Let C_{si} be the set of all pairs of operations of an interface $si \in s.I$ of a service s, formed such that it contains either (op_i, op_j), or (op_j, op_i) (Table II(4)). The lack of sequential cohesion, $LoC_S(si)$ for si is defined as the complement of the average sequential similarity of the pairs of operations that belong to C_{si} (Table II(5)). Similarly, the lack of communicational cohesion, $LoC_C(si)$ for si is defined as the complement of the average communicational similarity of the pairs of operations that belong to C_{si} (Table II(6)).

B. Analytic Validation

To provide a first analytical assessment of the proposed metrics we follow the approach of Chidamber & Kemerer [6]. Specifically, the goal is to examine the behavior of LoC_S and LoC_C with respect to the properties of software metrics that have been proposed by Weyunker [16] (i.e. *non-coarseness*, *non-uniqueness*, *design-details-importance*, *monotonicity*, *non-equivalence* and *increased-complexity*). As discussed in detail in this subsection, the proposed metrics exhibit a similar behavior with the Lack of Cohesion of Methods ($LCOM$) metric proposed by Chidamber & Kemerer [6] in their object-oriented metrics suite. Moreover, the behavior of LoC_S and LoC_C with respect to the examined properties is the same. Therefore, hereafter we use the term LoC_* to refer to either LoC_S, or LoC_C.

Briefly, *non-coarseness* states that the values of LoC_* are not the same for all interfaces, whilst *non-uniqueness* states that the values of LoC_* are not different for all interfaces. The *design-details-importance* property states that different interfaces that offer the same functionality, may have different LoC_* values. As discussed below, proving that these properties are satisfied by LoC_* can be done, by following similar steps as in [6] for the case of $LCOM$. The remaining three properties considered in [6], concern the behavior of LoC_* when merging two interfaces si, sj; the merging of two interfaces $si \bigcup sj$ results in a service interface that provides the union of the operations of si and sj, i.e., $(si \bigcup sj).O = si.O \bigcup sj.O$. Following, we show that the *monotonicity* property is not satisfied, i.e., LoC_* may not increase monotonically when merging si with sj. Moreover, we show that the *non-equivalence* property is satisfied, i.e., if si, sj are characterized by the same LoC_* value and each one of them is merged with the same interface sk, the resulting two interfaces may be characterized by different LoC_* values. Finally, we show that the *increased-complexity* property is not satisfied, i.e., the value of LoC_* for the interface that results from merging si, sj may be less than the sum of the LoC_* values of the merged interfaces.

Property 1: (Non-coarseness) Given an interface si, another interface sj can always be found such that $LoC_*(si) \neq LoC_*(sj)$.

Proof: To prove this proposition we rely on the approach followed by Chidamber and Kemerer [6]. Specifically, in [6] the authors assume that the numbers of methods and attributes that characterize different classes are independent identically distributed random variables. Similarly, we can assume that the orders of the common bottom-up subtrees of the messages that characterize operations which belong to different service interfaces are independent identically distributed random variables. Based on this assumption, for any interface si there is a nonzero probability that there exists an interface sj, such that $LoC_*(si) \neq LoC_*(sj)$. ■

Property 2: (Nonuniqueness) There can exist interfaces si, si, such that $LoC_*(si) = LoC_*(sj)$.

Proof: As in the proof of Property 1 we can rely on the assumption that the orders of the common bottom-up subtrees of the messages that characterize operations which belong to different service interfaces are independent identically distributed random variables. Thus, for any interface si, there is a nonzero probability that there exists an interface sj, such that $LoC_*(si) = LoC_*(sj)$. ■

Property 3: (Design-details-importance) Given two interfaces si, sj, the fact that si and sj provide the same functionalities does not imply that $LoC_*(si) = LoC_*(sj)$.

Proof: In general, the definition of an interface that

provides certain functionalities is a design choice that is not restricted in any sense. For the same functionality it is possible, for instance, to define alternative operations whose input/output messages are structured differently. ■

Property 4: (Monotonicity) There can exist service interfaces si, sj, such that LoC_* is not monotonically increasing with respect to their merging, i.e. the following inequality may not hold: $LoC_*(si) \leq LoC_*(si \bigcup sj)$.

Proof: Let si, sj be the two interfaces that are going to be merged. If due to design flaws the operations of si are not related (communicationally or sequentially) with each other, while certain of them are related with the operations of sj, then obviously after merging the two interfaces we shall have that $LoC_*(si) > LoC_*(si \bigcup sj)$. ■

Property 5: (Nonequivalence) There exist service interfaces si, sj, sk such that, $LoC_*(si) = LoC_*(sj)$ does not imply that $LoC_*(si \bigcup sk) = LoC_*(sj \bigcup sk)$.

Proof: Let si, sj, sk be three interfaces, such that $LoC_*(si) = LoC_*(sj)$. However, sk may be such that its operations are related (communicationally or sequentially) with the operations of si, but unrelated with the operations of sj. Then, obviously $LoC_*(si \bigcup sk) \neq LoC_*(sj \bigcup sk)$. ■

Property 6: (Increased-complexity) There can exist service interfaces si, sj, such that the following inequality does not hold: $LoC_*(si) + LoC_*(sj) < LoC_*(si \bigcup sj)$.

Proof: Let si, sj be the two interfaces that are going to be merged. If due to design flaws the operations of si are not related (communicationally or sequentially) with each other, while certain of them are related with the operations of sj, and the same holds for sj, then obviously after merging the two interfaces we shall have that $LoC_*(si) + LoC_*(sj) > LoC_*(si \bigcup sj)$. ■

As in the case of the $LCOM$ metric [6], the fact that the *monotonicity* and the *increased-complexity* properties are not satisfied is not a negative result. In general, merging and splitting interfaces are candidate solutions for improving the cohesion of interfaces and the non-satisfaction of the aforementioned properties shows that the benefits of applying any of these solutions are reflected by the values of the metric.

C. Example: Lack of cohesion of `MessageQueue`.

Returning to the case of the SQS service, Figure 2 gives examples of operations provided by the `MessageQueue` interface. An example that highlights the negative effect of generic meta-data elements involves the `ResponseStatus` element. `ResponseStatus` is a complex element that contains general purpose response meta-data; the element is included in all of the output messages that are used in SQS. Then, for the output messages of `SetQueueAttributes` and `DeleteQueue`, for instance, there exists a common bottom-up `ResponseStatus` subtree. In other words, the output messages of the two operations appear to be similar, despite the fact that their functionalities are completely unrelated. According to the SQS specification, the `SetQueueAttributes` operation manipulates certain queue attributes, while the `DeleteQueue` operation deletes a particular queue. Applying the filtering discussed earlier solves the problem. In particular, if we filter out the `ResponseStatus` elements, the output messages of `SetQueueAttributes` and `DeleteQueue` are no longer similar to each other.

In our example, we can further observe a common bottom-up subtree (solid-line) in the input message of the `SetQueueAttributes` operation and the output message of the `GetQueueAttributes` operation. This tree consists of 3 vertices, while the union of the two messages consists of 7 vertices. Hence, the similarity of the two messages is $\frac{3}{7}$. On the other hand, the similarity between the input message of `GetQueueAttributes` and the output message of `SetQueueAttributes` is 0 because these messages do not contain common bottom-up subtrees. Therefore, the value of the sequential similarity OpS_{seq} for the operations is $\frac{\frac{3}{7}}{2}$.

Finally, we can observe a trivial common bottom-up subtree (dashed-line) in the input message of the `SetQueueAttributes` operation and the input message of the `GetQueueAttributes` operation. The subtree consists of a single vertex named `Attribute`. The union of these two messages comprises 7 vertices. Therefore, the similarity of the two messages is $\frac{1}{7}$. Since the output messages of these two operations are not similar, the overall communicational similarity OpS_{com} for the operations is $\frac{\frac{1}{7}}{2}$.

Overall, for the `MessageQueue` interface there are 78 pairs of operations, that contribute to the values of the proposed lack of cohesion metrics. Specifically, we have: $LoC_S(MessageQueue) = 0.98$ and $LoC_C(MessageQueue) = 0.98$. As expected, in both cases, the values of the metrics are very close to 1, reflecting the cohesion lack of the `MessageQueue` interface.

III. Evaluation

The purpose of the evaluation is to assess the usefulness of the proposed metrics in identifying potential cohesion problems for real-world services. To automatically calculate the value of the metrics, we implemented a prototype that consists of two core components as depicted in Figure 3. The *ServiceParser* component of the prototype takes as input the WSDL-based specifications of services, and populates a data structure that corresponds to the conceptual model of Table I. The *CohesionLackCalculator* component of the prototype can be customized so as to calculate the value of any of the metrics given in Table II(5, 6).

In the rest of this section we describe the methodology of the evaluation, the case studies that were used, and the results that we obtained.

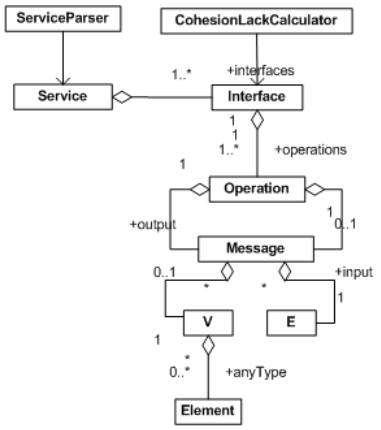

Figure 3. The prototype implementation.

A. Methodology of the Evaluation

To assess the capability of the proposed metrics to determine cohesion problems, we experimented with services provided by Amazon[4]. The issue is whether the values of the metrics are indicative of the existence of potential cohesion problems. To address the aforementioned issue, the strategy of our evaluation process comprises two steps. Firstly, for every interface the values of the communicational and the sequential cohesion metrics are calculated. Secondly, if the value of any of the two metrics is close to 1, we turn to a manual inspection of the interface. The purpose of the manual inspection is to check the relatedness of the operations within the interface based on its documentation.

B. Case Studies

Amazon provides 21 services that are available through the Web. Amongst these services, 16 were used for the purpose of our evaluation. The remaining 5 services were excluded due to a technical constraint of our current prototype. Our prototype accepts as input WSDL-based specifications of service interfaces, while for the 5 services that were excluded from the evaluation, Amazon does not provide specifications in the aforementioned format. The 16 Amazon services provide an overall number of 19 interfaces; 14 of the examined services provide a single interface, while the remaining 2 services (the Fulfilment Web Service (FWS) and the SQS service that served as an example in previous sections) provide 3 and 2 interfaces, respectively.

Hereafter, for reasons of simplicity we use identifiers A1-A19 to refer to the interfaces of the Amazon services. The mapping of identifiers to service interfaces and the sizes of the interfaces (i.e., the number of provided operations) are given in detail in Table III. In general, we can observe that Amazon provide interfaces that consist of large numbers of operations; more than 50% of the interfaces provide more than 10 operations, while the largest interface is A18 that comprises 87 operations.

[4]Amazon services - http://aws.amazon.com/

Table III
AMAZON SERVICE INTERFACES.

Service Interface		
Name	*Size*	*ID*
CloudWatchPortType	2	A1
ElasticMapReducePortType	4	A2
AmazonFBAOutboundPortType	7	A3
AmazonSNSPortType	13	A4
MechanicalTurkRequesterPortType	27	A5
ElasticLoadBalancingPortType	13	A6
AmazonFPSPortType	27	A7
ImportExportPortType	5	A8
QueueService	2	A9
AmazonFWSInventoryPortType	4	A10
AmazonLSPortType	6	A11
AmazonSDBPortType	9	A12
MessageQueue	13	A13
AutoScalingPortType	13	A14
AmazonFWSInboundPortType	18	A15
AmazonVPCPortType	21	A16
AmazonRDSv2PortType	23	A17
AmazonEC2PortType	87	A18
AmazonS3	16	A19

C. Results

The results that we obtained are provided in detail in Table IV. Starting from the largest interface (A18) of our case studies, we observe that the values of LoC_S and LoC_C are 0.99 and 0.98 respectively, indicating the existence of a cohesion problem. Obviously, the problem was expectable because of the very large number of operations (A18 contains 87 operations). In this case, the interface contains lots of unrelated operations leading to metric values very close to 1.

Turning into the manual inspection of A18, its overall purpose is to give access to a virtual computing environment. The operations of A18 can be divided into 22 groups. The operations of each group are more similar to each other than to the operations of the other groups. Indicatively, we enumerate the operations of two groups. The first group consists of operations for the manipulation of IP addresses: `AllocateAddress()`, `AssociateAddress()`, `DescribeAddresses()`, `DisassociateAddress()`, and `ReleaseAddress()`. The second group concerns the manipulation of account keys, and comprises the operations: `CreateKeyPair()`, `DeleteKeyPair()`, and `DescribeKeyPairs()`.

Continuing with the medium sized interfaces (A5, A7, A15-A17, and A19), whose number of operations is between 15 and 30, we observe that the values of the metrics range from 0.84 to 0.95. Even if the size of the interfaces is almost the 25% of the size of A18, the lack of cohesion values

Table IV
RESULTS FOR THE CASE STUDIES.

	Cohesion Type	
ID	LoC_S	LoC_C
A1	0.86	1.00
A2	0.94	0.97
A3	0.94	0.94
A4	0.97	0.96
A5	0.84	0.91
A6	0.97	0.93
A7	0.97	0.91
A8	0.96	0.93
A9	1.00	0.90
A10	0.99	0.83
A11	0.97	0.95
A12	0.97	0.94
A13	0.98	0.98
A14	0.98	0.96
A15	0.96	0.93
A16	0.98	0.95
A17	0.96	0.91
A18	0.99	0.98
A19	0.97	0.88

are still very high. Again, the manual inspection showed cohesion problems that justify the values of the metrics. For instance, the A16 interface provides a secure and seamless bridge between a company's existing information technology (IT) infrastructure and the cloud of Amazon. In this case, we identified 6 groups of closely related operations. Indicatively, we enumerate the operations of two groups. The first group consists of operations for the manipulation of a connection between a company's infrastructure and the Amazon cloud: `CreateVpc()`, `DeleteVpc()`, and `DescribeVpcs()`. The second group concerns the manipulation of subnets within the Amazon cloud, and comprises the operations: `CreateSubnet()`, `DeleteSubnet()`, and `DescribeSubnets()`.

Turning to the small sized interfaces (A1-A4, A6, A8-A14), we were intuitively expecting lower values of the metrics than the cases of the medium sized interfaces. However, we observe that this assumption does not hold because the values of the metrics range in nearly the same level as before (from 0.83 to 0.98). The inspection of the interfaces showed that although the size of the interfaces is small, their operations are not similar to each other.

As an example, we discuss the A13 interface, which also served as an example in previous sections. We observe that the value of both metrics is equal to 0.98 indicating a cohesion problem. Based on the documentation, we identified 4 groups of tightly related operations. The first group comprises operations for adding/removing/listing access permissions for a particular queue: `AddGrant()`, `ListGrants()`, and `RemoveGrant()`. The second group concerns getting/setting certain queue timeouts, and consists of the operations: `GetVisibilityTimeout()`, and `SetVisibilityTimeout()`). The third group consists of operations for sending messages to a queue, receiving/deleting messages from a queue and changing the visibility of messages: `SendMessage()`, `PeekMessage()`, `ReceiveMessage()`, `DeleteMessage()`, and `ChangeMessageVisibility()`. The last group enables the deletion of a queue and consists of only one operation: `DeleteQueue()`.

Overall, based on our results we can conclude in the following point: the proposed metrics succeed in indicating cohesion problems for real-world service interfaces. Moreover, in our services set, we observed that the definition of cohesive interfaces was not a primary priority of the designers. This does not necessarily mean that the design of the services is flawed. However, cohesion should not be neglected from the design of widely used services, given that the effects of cohesion lack may affect numerous applications all over the Web that depend on these services.

IV. RELATED WORK

The metrics proposed in this paper are inspired from various cohesion metrics that have been proposed in the context of OO software development. In particular, in [6] the authors propose the well known LCOM metric (Lack of Cohesion of Methods), which is defined in terms of the number of pairs of class methods that use common class attributes and the number of pairs of class methods that do not use common class attributes. Moreover, in [8] the authors propose metrics of cohesion lack defined in terms of the number of methods that use each one of the attributes of a given class. In [9], [3] the authors propose metrics that further take into account class methods which use other class methods. In [2] the authors propose a cohesion metric that is based on the parameter types of the methods that belong to a given class. Given the variety of OO metrics that have been proposed in the literature, certain approaches proposed unified frameworks for the quality analysis of software [1] and frameworks for the comparison of existing metrics [4], [7].

In the context of service-oriented development, the aforementioned OO metrics can be used to assess the internal implementation of services. However, they can not be used to reflect the relevance of the operations of service interfaces. Generally, in SOA the state of the art on cohesion metrics for service interfaces is limited to the work of Perepletchikov et al. [10], [11], [12]. In this line of research the authors consider various notions of cohesion and propose corresponding metrics. Specifically, certain metrics focus on measuring the cohesion between a service interface and the internal service implementation (SIIC). The values of these metrics are calculated as a function of the number of operations that use the same internal implementation elements. Certain other metrics concentrate on the cohesion between the users

of a service and the service (SIUC). In this case, the values of the proposed metrics are calculated as a function of the number of service consumers that use all of the operations.

Regarding sequential and communication cohesion, which are amongst the most desirable fundamental notions of cohesion [13], [17], the authors proposed two metrics, namely SISC and SIDC. According to SIDC, a service interface is cohesive if its operations are characterized by common types of input parameters and common types of output parameters. Similarly, regarding SISC, a service interface is cohesive if its operations are sequentially dependent, in the sense that the types of certain output parameters of one operation match the types of certain input parameters of another operation. SISC and SIDC are quite coarse grained, compared to the metrics that we proposed in this paper. In particular, in the case of our metrics the relatedness between operations is measured with respect to the similarity of their input/output data types, while in SISC and SIDC the relatedness between operations is measured with respect to exactly matching input/output data types. As already discussed (Section 1), the latter requirement (i.e., exactly matching input/output data types) may lead to the overestimation of cohesion lack. Another problem that may also negatively affect the precision of SISC and SIDC is that the approach proposed in [10], [11], [12] for the calculation of the metrics values does not make any distinction between normal input/output data and meta-data that are common in all operations.

V. Conclusion and Future Work

This paper investigated the notion of cohesion in service interfaces. We proposed two fine-grained metrics that measure the lack of communicational and sequential cohesion. The metrics were formally defined with respect to the structural similarity of the input/output data types of interface operations, and analytically validated with respect to fundamental properties of software metrics. Moreover, we evaluated the capability of the metrics to identify cohesion problems in real-world services.

An interesting issue that we currently work on is to provide automated support for improving the cohesion of a service interface. A possible way to do it is to develop a restructuring mechanism that makes use of the proposed metrics in order to divide the interface into groups of closely related operations. Each group corresponds to a separate interface which is more cohesive than the original one. To this end, we plan to investigate the use of hierarchical clustering algorithms along with the metrics that have been proposed in this paper.

References

[1] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28:4–17, January 2002.

[2] J. Bansiya, L. H. Etzkorn, C. G. Davis, and W. Li. A class cohesion metric for object-oriented designs. *JOOP*, 11(8):47–52, 1999.

[3] J. M. Bieman and B.-K. Kang. Cohesion and Reuse in an Object-Oriented System. *ACM SIGSOFT Software Engineering Notes*, 20:259–262, 1995.

[4] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[5] J. Cardoso and A. Sheth. *Semantic Web Services, Processes and Applications*. Springer, 2006.

[6] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[7] S. Counsell, S. Swift, and J. Crampton. The interpretation and utility of three cohesion metrics for object-oriented design. *ACM Transactions on Software Engineering and Methodology*, 15(2):123–149, 2006.

[8] B. Henderson-Sellers, L. L. Constantine, and I. M. Graham. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems*, 3:143–158, 1996.

[9] M. Hitz and B. Montazeri. Measuring Coupling and Cohesion in Object-Oriented Systems. In *Proceedings of the International Symposium on Applied Corporate Computing*, 1995.

[10] M. Perepletchikov, C. Ryan, and K. Frampton. Cohesion Metrics for Predicting Maintainability of Service-Oriented Software (QSIC). In *Proceedings of the 7th IEEE International Conference on Quality Software*, pages 328–335, 2007.

[11] M. Perepletchikov, C. Ryan, K. Frampton, and H. W. Schmidt. Formalising service-oriented design. *JSW*, 3(2):1–14, 2008.

[12] M. Perepletchikov, C. Ryan, and Z. Tari. The impact of service cohesion on the analyzability of service-oriented software. *IEEE Transactions on Services Computing*, 3(2):89–103, 2010.

[13] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[14] E. Thomas. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, 2005.

[15] G. Valiente. Simple and efficient subtree isomorphism. Technical Report LSI-00-72-R, Technical University of Catalonia, Department of Software, 2000.

[16] E. J. Weyuker. Evaluating Software Complexity Measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988.

[17] E. Yourdon and L. Constantine. *Structured Design*. Prentice-Hall, 1979.