Cohesion-Driven Decomposition of Service Interfaces Without Access to Source Code

Dionysis Athanasopoulos, Apostolos V. Zarras, *Member, IEEE,* George Miskos, Valerie Issarny, and Panos Vassiliadis *Member, IEEE*

Abstract—Software cohesion concerns the degree to which the elements of a module belong together. Cohesive software is easier to understand, test and maintain. In the context of service-oriented development, cohesion refers to the degree to which the operations of a service interface belong together. In the state of the art, software cohesion is improved based on refactoring methods that rely on information, extracted from the software implementation. This is a main limitation towards using these methods in the case of Web services: *Web services do not expose their implementation; instead all that they export is the Web service interface specification*. To deal with this problem, *we propose an approach that enables the cohesion-driven decomposition of service interfaces, without information on how the services are implemented*. Our approach progressive decomposes a given service interface into more cohesive interfaces; the backbone of the approach is a suite of cohesion metrics that rely on information, extracted solely from the specification of the service interface. We validate the approach in 22 real-world services, provided by Amazon and Yahoo. We assess the effectiveness of the proposed approach, concerning the cohesion improvement, and the number of interfaces that result from the decomposition of the examined interfaces. Moreover, we show the usefulness of the approach in a user study, where developers assessed the quality of the produced interfaces.

Index Terms—Cohesion, Decomposition, Service interface

1 Introduction

Alice in the Web services world: Alice is an ordinary Java developer. Some time ago, she discovered the benefits of using Web services for developing software. Alice finds them very handy. As it is typically done, the applications that she develops access services via JAX-WS[1] proxies, generated from the WSDL specifications of the services. A JAX-WS proxy looks much like an ordinary Java class, but its methods delegate calls to service operations and bring the results back to the application.

However, using services also has its drawbacks. Often, new versions of the Web service interfaces are released and Alice spends quite some time to test and maintain her software, when this happens. For instance, one of the projects that Alice is involved in relies on the Amazon Simple Queue Service (SQS) service[2]. SQS facilitates message-based communication for applications running on the Amazon Cloud *via queues*; it provides operations for (a) the creation and management of message queues, (b) message storage and retrieval to/from message queues, (c) the management of queue access grants, and, (d) the management of message visibility timeouts. Developers blend calls to SQS operations in their code to allow their applications communicate via SQS message queues. Since 2007, the main interface of the service has been changed more than 4 times [3]. Whenever the `MessageQueue` interface changes, the continuous integration development platform (CIDP) that is used in Alice's project traces that the `MessageQueue` proxy has changed. Following, the CIDP rebuilds the whole application and retests all the classes since they depend on the changed proxy. This overall process takes too long. Worst, Alice spends much time on checking the built logs and the test results to find out which tasks went right, or wrong.

On the back of her head, Alice has an idea that could save her from this burden. The idea is to split the `MessageQueue` interface into a set of new interfaces and develop a corresponding set of surrogate classes that implement these interfaces (Figure 1). The methods of the surrogate classes would then call the actual `MessageQueue` operations, via the `MessageQueue` proxy. Making the application use the surrogate classes, instead of directly using the `MessageQueue` proxy, will decouple the constituent parts of the application from service operations that are not actually used in these parts. In this setting, changes to the `MessageQueue` interface shall affect certain surrogate classes. Then, only the parts of the application that use the affected surrogate classes will have to be re-built and re-tested. In fact, this idea would be useful for many others that use Amazon

- *D. Athanasopoulos, A. Zarras, G. Miskos and P. Vassiliadis are with the Department of Computer Science, University of Ioannina, Greece. E-mail: {dathanas, zarras, pvassil}@cs.uoi.gr, gmiskos@gmail.com*
- *V. Issarny is with the Inria research center of Paris-Rocquencourt, France E-mail: Valerie.Issarny@inria.fr*

1. download.oracle.com/otndocs/jcp/jaxws-2_0-fr-eval-oth-JSpec/
2. aws.amazon.com/sqs/
3. aws.amazon.com/articles/Amazon-SQS/1148

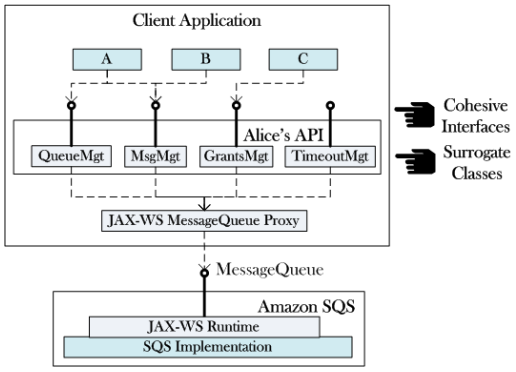


Fig. 1. A client application that relies on a cohesive decomposition of the `MessageQueue` interface.

SQS. So, Alice plans to make her new interfaces and the surrogate classes that implement them available as an open source Java API. Alice thinks that the same idea could also be useful in the case of services that provide a large number of operations. Amazon EC2[4], for instance, provides 87 operations grouped in a single interface. Similarly, Yahoo KeywordService[5] provides 34 operations, grouped in a single interface. The decomposition of such large interfaces could be used to develop APIs that provide the developers with higher-level views of what the services do.

Having in mind a larger community of developers that could benefit from her idea, puts Alice into deeper thoughts about the proper splitting of service interfaces. The decomposition of service interfaces should be done in a principled way. Alice recalls the fundamental notion of *cohesion*. In general, software cohesion refers to *the degree to which the elements of a module belong together* [1]. Cohesive software is easier to understand, test and maintain. In the case of Web service interfaces, the splitting should rely on a certain notion of cohesion that reflects the relatedness of the operations which are grouped in the same interface [2], [3], [4], [5], [6].

Technical challenge: Unfortunately, Alice cannot obtain her desideratum of splitting a service interface into a set of cohesive interfaces via the state of the art cohesion-driven refactoring methods [7], [8], [9], [10], [11], [12], [13], [14], [15]. On the one hand, like all Web services, the ones that Alice uses *do not expose their internals, i.e., their source code*; on the contrary, the very philosophy of Web services dictates that all that is exported by a Web service is *the Web service interface specification*. On the other hand, the cohesion-driven refactoring methods are tailored to operate by taking the source code into consideration. To overcome this problem, in this paper we propose an approach that facilitates *the cohesion-driven decomposition of service interfaces, without information on how the services are implemented*.

Contribution: The backbone of our approach is *a suite of cohesion metrics* for service interfaces. Specifically, to keep our approach independent from the way that cohesion is measured, we introduce *a generic cohesion metric* that quantifies the degree to which the operations of a service interface are related, based on *interface-level relations*, extracted from the service interface specification. Following, we reformulate the metrics of [6], as refinements of the generic cohesion metric; the Lack of Message-Level Cohesion (LoC_{msg}) and the Lack of Conversation-Level Cohesion (LoC_{conv}), account for interface-level relations, between operations that have similar types of inputs/outputs. We further extent [6], with a new metric, which is also introduced as a refinement of the generic cohesion metric; the Lack of Domain-Level Cohesion (LoC_{dom}), considers interface-level relations, between operations that are characterized by similar domain-level terms, which are extracted from the names of the operations. Our *cohesion-driven decomposition method* accepts as input a cohesion metric and a service interface. The given interface is progressively split into more cohesive interfaces. If it is no longer possible to produce more cohesive interfaces, the decomposition stops.

We have validated the proposed approach in 22 case studies that concern services provided by Amazon and Yahoo. We have evaluated the *effectiveness* of our approach concerning the cohesion improvement, and the number of interfaces that result from the decomposition of the examined interfaces. Moreover, we have assessed the usefulness of the approach in a user study, where developers evaluated the quality of the produced interfaces, as well as the success of each of the proposed cohesion metrics.

The rest of this paper is structured as follows. In Section 2, we discuss our contribution with respect to the state of the art. In Section 3, we present our metrics suite. In Section 4, we detail the *modus operandi* of the decomposition method. In Section 5, we discuss the results that we obtained. Finally, in Section 6 we summarize our contribution and discuss the future perspectives of this work.

2 Related Work

In this section, we discuss in further detail the contribution of our approach with respect to the state of the art. More specifically, in Section 2.1 we discuss the relation of our approach with previous efforts on software refactoring. Then, in Section 2.2 we focus on cohesion metrics that have been proposed in the object-oriented and the service-oriented paradigms.

2.1 Refactoring

Refactoring is a behavior preserving changing process that improves the quality of a software system [16].

4. aws.amazon.com/ec2/
5. developer.searchmarketing.yahoo.com/docs/V6/reference/services/

For an excellent survey on refactoring the interested reader may refer to [17].

Our approach, is more closely related to metrics-driven refactoring methods, which employ metrics to discover and repair design problems. To achieve this goal, the state of the art methods rely on *implementation-level relations*, derived from source code (Appendix A, Table 5 [6]). Specifically, in [18], Harman & Tratt focus on the improvement of coupling. To this end, they rely on dependencies between classes, quantified based on the well-known CBO metric [19]. In [8], [9] and [7], the goal is to improve the cohesion of classes, by taking into account relations between class methods and attributes (or other methods), used by the methods. Certain approaches consider the improvement of multiple quality properties. In particular, the methods proposed in [10] and [11] focus on both coupling and cohesion. The methods proposed in [12], [13] account for coupling, cohesion and code complexity. In [14] the proposed method considers the improvement of coupling, cohesion and code size. Finally, the method proposed in [15] accounts for coupling, cohesion, code complexity and size.

Concerning their *modus operandi*, the metrics-driven refactoring methods can be divided in two categories. In the first category, the methods require more involvement from the developer [8], [13], [7]. In particular, based on the values of the metrics that are considered, the methods identify possible refactorings that can improve the values of the metrics. Following, the developer is supposed to select and apply the refactoring that suits his/her preferences. In the second category, the methods do more work on behalf of the developer. These methods consider the refactoring as an iterative process. As long as the design of the classes can be improved the refactoring process keeps going. The algorithms that are used to realize the refactoring process vary. We have methods that rely on meta-heuristic optimization algorithms (e.g., hill-climbing [18], [14], simulated annealing [14], genetic algorithms [10], [12], [15]). Moreover, we have methods that are based on clustering [9].

Concerning the state of the art, *our approach is the first one that deals with the cohesion-driven decomposition of service interfaces*. We address this problem *without assuming knowledge on how the services are implemented*. Instead, we rely on *interface-level relations*, extracted from the specification of the service interfaces.

2.2 Cohesion Metrics

In the early 90's Chidamber and Kemerer proposed the well-known LCOM metric (Lack of Cohesion of Methods) for measuring the cohesion of object-oriented software [19]. The interested reader may refer to [20] and [21] for two detailed surveys of the cohesion metrics that have been proposed since the seminal work of Chidamber and Kemerer. In the service-oriented paradigm, cohesion was recognized as an important principle of service design in several approaches that concern the overall service-oriented development methodology [2], [3], [4]. The first efforts for measuring cohesion have been made in the work of Perepletchikov et al. [5]. The first study that investigated the issue of cohesion in the case of real-world services is reported in Athanasopoulos & Zarras [6]. Finally, another interesting work that concerns the cohesion of services is presented in [22].

In the object-oriented paradigm, the majority of the cohesion metrics measure the degree to which the methods of a class are related based on implementation-level relations (Appendix A, Table 6). Two class methods are considered as being related if they use common class attributes (or methods). In the object-oriented paradigm, we also have cohesion metrics that assess the relatedness of methods based on interface-level relations. In these metrics, two methods are considered as being related if they have parameters of the same type.

In the service-oriented paradigm, the SIIC metric [5] measures the relatedness of service operations, with respect to implementation-level relations. The value of SIIC for a service is the fraction of the number of the common service implementation elements used by the service operations, over the total number of service implementation elements used by the service operations. Similarly, the SCV metric [22] also relies on implementation-level relations. The value of SVC for a service is the normalized sum of the relatedness values that characterize the business entities, used by the service operations. A business entity is an information entity used by the service operation; the relatedness between business entities is calculated using a mathematical method called Singular Value Decomposition (SVD). SIDC and SISC [5] rely on interface-level relations. In particular, the value of SIDC for a service is the normalized sum of the pairs of service operations that have at least one input parameter type in common, and the pairs of service operations that have the same return type. The value of SISC is the fraction of the pairs of service operations that have sequential dependencies, over the total number of pairs; a sequential dependency signifies that the output of one operation satisfies the input of another operation. The SIUC metric [5], also operates at the interface-level. The value of SIUC for a service is the normalized sum of the number of service operations that are used by the clients of the service.

Our cohesion metrics focus on interface-level relations, because implementation-level information is typically not exposed by the services. For the same reason, we do not consider information concerning the usage of operations by the service clients. LoC_{msg} and LoC_{conv} are more closely related with SIDC and SISC, in the sense that these metrics also focus on the

6. Appendices available as supplemental material at the journal's site.

input/output parameters of service operations. SIDC and SISC consider equality between parameter types. On the other hand, LoC_{msg} and LoC_{conv} consider *similarity between parameter types*. The LoC_{dom} metric follows *a completely different direction* for measuring the cohesion of service interfaces, as it relies on relations between operations that are characterized by similar domain-level terms, which are extracted from the names of the operations.

The specification of service interfaces may further include ontology-based annotations (e.g., SA-WSDL[7]). At this stage, our metrics take into account the parts of the specification of service interfaces, concerning their names and input/output parameters. Nevertheless, the extension of the metrics to account for ontology-based annotations is an interesting issue for future research that can be achieved based on the recent advances in the field of ontology-based similarity and cohesion metrics (e.g., [23]).

From a broader perspective, our metrics are related with similarity-based cohesion metrics that have been employed in document clustering techniques (e.g., [24]). Nevertheless, the specification of a service interface is a document that has a specific structure and semantics and our metrics are tailored to these aspects.

3 Interface-Level Cohesion Metrics

In this section, we focus on the cohesion metrics that we employ for the decomposition of service interfaces. In Section 3.1, we introduce *a generic cohesion metric*, LoC_*, that quantifies the lack of cohesion of a service interface, based on *a generic similarity function*, OpS_*, between operations. In Section 3.2, we define three concrete refinements of the generic cohesion metric that rely on corresponding concrete similarity functions, which account for different kinds of interface-level relations between operations. In Appendix B, we validate the metrics with respect to the theoretical framework of Briand et al. [25].

3.1 Basic Concepts

Our overall approach for measuring cohesion is based on a generic view of the notion of service interface, which is given in the following definition.

***Definition** 1:* **(Service interface)** A service interface, si, is characterized by a name and a set of operations, $si.O$ (Table 1(1)). An *operation* is characterized by a *name*, an *input message* and an *output message* (Table 1(2)). A *message* is a set of parameters (Table 1(3)). Each *parameter* has a *name* and a *type*, which may be either an XML build-in type, or an XML complex type (Table 1(4)).

To measure cohesion, we further employ the concept of *interface-level graph*, which represents the interface-level relations between the operations of a given service interface. In general, two operations are related if their properties (e.g., names, parameters) are similar to some extent, according to a particular similarity function. More formally, the definition of the interface-level graph, with respect to a generic similarity function, OpS_*, is given below.

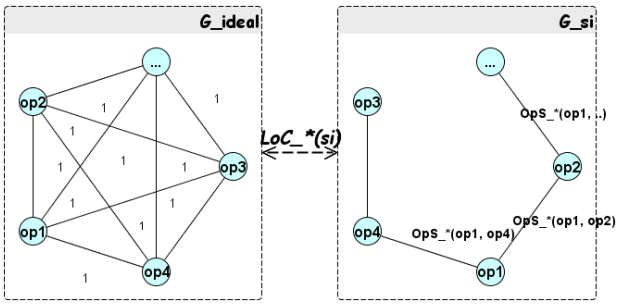


Fig. 2. The meaning of LoC_*.

***Definition** 2:* **(Interface-level graph)** An interface-level graph, $G^*_{si} = (V_{si}, E_{si}, OpS_*)$, for a service interface, si, and a similarity function, $OpS_* : si.O \times si.O \rightarrow [0, 1]$ that reflects the degree to which the operations of si are related, is a weighted graph with the following properties (Table 1(5)): (a) the nodes, V_{si}, of the graph represent the operations of si; (b) the edges, E_{si}, of the graph represent interface-level relations between pairs of operations; (c) an edge, (op_i, op_j), belongs to E_{si}, iff $OpS_*(op_i, op_j) > 0$; the weight that characterizes the edge is $OpS_*(op_i, op_j)$.

TABLE 1
Basic concepts.

$$si = (name : string, O) \quad (1)$$

$$O = \{op : operation\}$$

$$operation = (name : string, in : message, out : message) \quad (2)$$

$$message = \{p : parameter\} \quad (3)$$

$$parameter = (name : string, type : anyType) \quad (4)$$

$$G^*_{si} = (V_{si}, E_{si}, OpS_*) \quad (5)$$

$$LoC_*(si, OpS_*) = 1 - \frac{\sum_{(op_i, op_j) \in E_{si}} OpS_*(op_i, op_j)}{\frac{|V_{si}| * (|V_{si}| - 1)}{2}} \quad (6)$$

Ideally, a service interface, si, would be fully cohesive if every operation of si is related with all the others and the similarity between every pair of operations is maximum (Figure 2, left). To this end, we define an ideal interface-level graph as follows.

***Definition** 3:* **(Ideal interface-level graph)** The ideal interface-level graph $G^*_{ideal} = (V_{si}, E_{ideal}, OpS_*)$ for a service interface, si, has two properties: (1) G^*_{ideal} is complete; (2) for all, $(op_i, op_j) \in E_{ideal}$, $OpS_*(op_i, op_j) = 1$.

7. www.w3.org/2002/ws/sawsdl/

Intuitively, the lack of cohesion for a service interface, si, measures the amount of transformation that the actual interface-level graph $G^*_{si} = (V_{si}, E_{si}, OpS_*)$ of si (Figure 2, right) must withstand to become identical to the ideal graph, G^*_{ideal} (Figure 2, left). This practically amounts to adding the missing edges and complementing the weights of the existing edges to become equal to 1.

***Definition** 4:* **(Lack of interface-level cohesion)** The lack of cohesion of a service interface si, $LoC_*(si, OpS_*)$, is defined as the relative difference between the ideal interface-level graph, G^*_{ideal}, and the interface-level graph, G^*_{si}, as follows: $LoC_*(si, OpS_*) = \frac{|E_{ideal}| - \sum_{(op_i, op_j) \in E_{si}} (OpS_*(op_i, op_j))}{|E_{ideal}|}$.

Given that $|E_{ideal}| = \frac{|V_{si}| * (|V_{si}| - 1)}{2}$, with simple algebraic calculations we get the formula that is given in Table 1(6).

3.2 Metrics Definitions

The proposed cohesion metrics refine the generic definition of LoC_* that was given in Section 3.1. In particular, the definitions of the metrics that we provide in the following paragraphs employ the notion of interface-level graph; the interface-level graph that is used for each metric relies on a different similarity function between operations.

Message-Level Cohesion

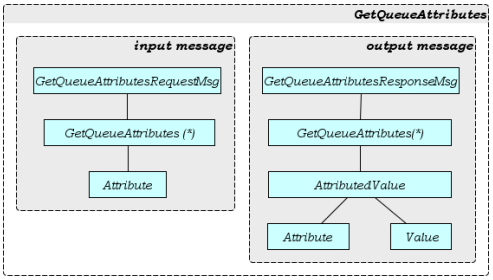
The notion of *message-level cohesion* assumes that two operations are related if their input (respectively, output) messages are similar. To measure the similarity between two messages we employ the notion of *message-level graph* that is defined below.

***Definition** 5:* **(Message-level graph)** A message-level graph, $G_m = (V_m, E_m)$, for a message, m, is a graph representation of the structure of m. The nodes of V_m are partitioned in three disjoint subsets $V_m = V^\mu_m \cup V^p_m \cup V^t_m$, defined as follows: (a) $V^\mu_m = \{v_\mu\}$, a single node representing the message itself, (b) V^p_m, a set of nodes, one per parameter of the message, and, (c) V^t_m, a set of nodes representing the elements of the structure of the parameter types. All edges of the graph, E_m denote whole-part relationships.

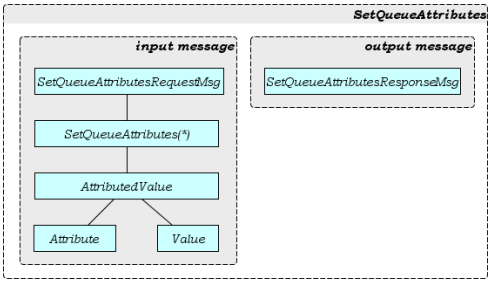
An explanation of Def. 5 is also worth here, concerning the types of the parameters: V^t_m includes nodes that represent primitive XML elements, or complex XML elements that consist of further (primitive or complex) XML elements. Bear in mind, that due to the XML nature of these types, they can contain cycles (thus, in general, they are graphs and not trees). Moreover, note that as in [6] nodes that correspond to generic meta-data elements are not included in a message-level graph, because they are not related to a particular service functionality.

Intuitively, two messages are similar if they have common parameters, or similar types of parameters. As parameter types are complex XML elements, whose specification comprises references to common subordinate (primitive or complex) XML elements, the message-level graphs of two similar messages contain a common subgraph that reflects the degree to which they are similar.

***Definition** 6:* **(Message similarity)** The similarity between the two messages, m_i, m_j, (Table 2(1)) is measured with respect to the message-level graphs, G_{m_i}, G_{m_j}, of m_i, m_j. Specifically, let $G_{m_i \cap m_j} = (V_{m_i \cap m_j}, E_{m_i \cap m_j})$ denote the maximum common subgraph of G_{m_i}, G_{m_j} that represents a syntactically correct XML schema. Moreover, let $G_{m_i \cup m_j} = (V_{m_i \cup m_j}, E_{m_i \cup m_j})$ be the union of G_{m_i}, G_{m_j} (i.e., $V_{m_i \cup m_j} = V_{m_i} \cup V_{m_j}$ and $E_{m_i \cup m_j} = E_{m_i} \cup E_{m_j}$). Then, the similarity, $MsgS(m_i, m_j)$, between m_i and m_j is the number of nodes of $G_{m_i \cap m_j}$, divided by the number of nodes of $G_{m_i \cup m_j}$.



(a) Messages of the `GetQueueAttributes` operation.



(b) Messages of the `SetQueueAttributes` operation.

Fig. 3. Examples of message-level graphs for `MessageQueue`.

The maximum common subgraph problem involves finding the largest subgraph of a graph G_{m_i} that is isomorphic to a subgraph of a graph G_{m_j} [26]. Solving the problem for two message-level graphs is simple; we match each subset of the nodes of G_{m_i} to its respective subset of G_{m_j}; as the nodes are uniquely labeled, the isomorphism is directly deduced by the nodes' labels.

Taking a step further, we define the message-level similarity between two operations as follows.

***Definition** 7:* **(Message-level operation similarity)** The message-level similarity, OpS_{msg}, between two operations, $op_i, op_j \in si.O$ of a service interface, si, is the average of (Table 2(2)):

1) the similarity between the input messages of op_i and op_j and

2) the similarity between the output messages of op_i and op_j.

Taking the example of Amazon SQS, Figure 3(a) shows the message-level graph for the input message of the `GetQueueAttributes` operation. The `GetQueueAttributesRequestMsg` node represents the message. The `GetQueueAttributes` node is a parameter that comprises of sequence of attributes. The `Attribute` node represents a primitive XML string element. Figure 3(a), further gives the message-level graph for the output message of the `GetQueueAttributes` operation. The `GetQueueAttributesResponseMsg` node represents the message. The `GetQueueAttributes` node represents a parameter that comprises a sequence of attribute value pairs. The `AttributedValue` node represents a complex XML element, which consists of two primitive XML string elements, represented by the `Attribute` and the `Value` nodes. Similarly, Figure 3(b) gives the message-level graphs for the input and the output messages of the `SetQueueAttributes` operation.

The maximum common subgraph for the message-level graphs of the two input messages comprises only the `Attribute` node. The union of the two graphs consists of 7 nodes. Hence, the similarity between the two input messages is $\frac{1}{7}$. On the other hand, the message-level graphs of the two output messages have nothing in common. Thus, the similarity between the two output messages is 0. Overall, the message-level similarity between the two operations is $\frac{\frac{1}{7}+0}{2}$.

Based on the message-level similarity between operations, we refine the LoC_* metric.

Definition 8: **(Lack of message-level cohesion)** For a service interface, si, the lack of message-level cohesion, $LoC_{msg}(si)$, is an alias for $LoC_*(si, OpS_{msg})$. Specifically, $LoC_{msg}(si)$ measures the relative difference between the interface-level graph, $G^{msg}_{si} = (V_{si}, E_{si}, OpS_{msg})$, defined based on the message-level similarity function, OpS_{msg}, and the ideal interface-level graph, G^{msg}_{ideal}.

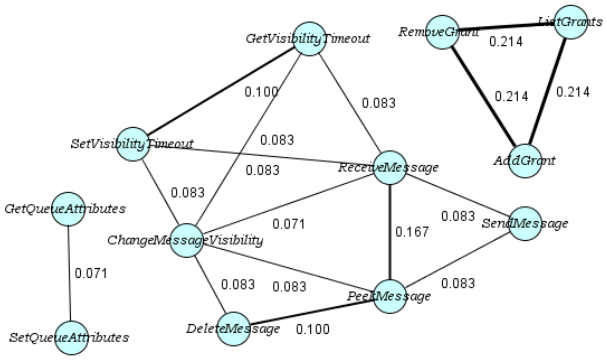


Fig. 4. $G^{msg}_{MessageQueue}$ for `MessageQueue`.

Figure 4, gives the interface-level graph for the `MessageQueue` interface that is derived with respect to OpS_{msg}. For presentation purposes the edges' width is proportional to the similarity between the operations. We see that the graph is not complete. Moreover, the message-level relations between the operations are weak; the similarities between the operations range from 0.07 to 0.21. Overall, the lack of message-level cohesion is $LoC_{msg}(MessageQueue) = 0.98$.

TABLE 2
Similarity functions.

$$MsgS(m_i, m_j) = \frac{|V_{m_i \cap m_j}|}{|V_{m_i \cup m_j}|} \quad (1)$$

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

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

$$OpS_{dom}(op_i, op_j) = \frac{|T_{op_i} \cap T_{op_j}|}{|T_{op_i} \cup T_{op_j}|} \quad (4)$$

Conversation-Level Cohesion

The notion of *conversation-level cohesion* assumes that an operation is related with another if the former's input (respectively output) message is similar with the latter's output (respectively input) message. More formally, we define the conversation-level similarity between two operations as follows.

Definition 9: **(Conversation-level operation similarity)** The conversation-level similarity between two operations, $op_i, op_j \in si.O$, of a service interface, si, is the average of (Table 2(3)):

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

Returning to our example, the input message of `GetQueueAttributes` (Figure 3(a)) and the output message of `SetQueueAttributes` (Figure 3(b)), have nothing in common. On the other hand, the maximum common subgraph for the output message of `GetQueueAttributes` and the input message of `SetQueueAttributes` includes three nodes (`AttributedValue`, `Attribute` and `Value`). Hence, the conversation-level similarity between the two operations is $\frac{\frac{3}{7}+0}{2}$.

Given the conversation-level similarity between operations, we introduce the following refinement of the LoC_* metric.

Definition 10: **(Lack of conversation-level cohesion)** For a service interface, si, the lack of

conversation-level cohesion, $LoC_{conv}(si)$, is an alias for $LoC_*(si, OpS_{conv})$. In particular, $LoC_{conv}(si)$ measures the relative difference between the interface-level graph, $G^{conv}_{si} = (V_{si}, E_{si}, OpS_{conv})$, defined with respect to the conversation-level similarity function, OpS_{conv}, and the ideal interface-level graph, G^{conv}_{ideal}.

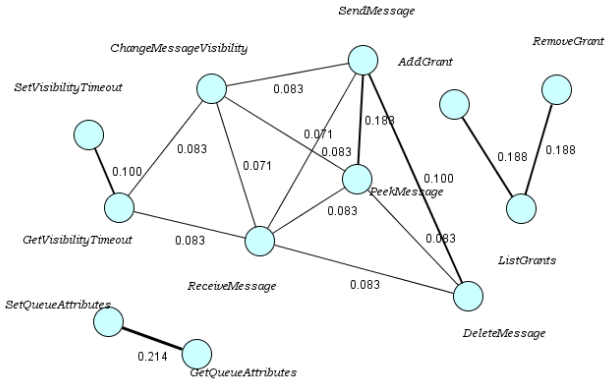


Fig. 5. $G^{conv}_{MessageQueue}$ for `MessageQueue`.

Regarding our example, the interface-level graph that shows the conversation-level relations for the `MessageQueue` interface is given in Figure 5. As in the case of message-level cohesion, the graph is not ideal. The overall lack of conversation-level cohesion is $LoC_{conv}(MessageQueue) = 0.98$.

Domain-Level Cohesion

The basic intuition behind the notion of *domain-level cohesion* is that the names of the operations that are provided by a service reflect what these operations do. More specifically, the names of the operations comprise *terms that correspond to certain actions* (e.g., set, get) and *terms that correspond to concepts of the domain that is targeted by the service* (e.g., queue, attribute, message). Based on this intuition, two operations are considered as being related if their names share domain-level terms.

In our approach, we assume that the names of the operations follow standard naming conventions of widely adopted coding styles; we extract the domain-level terms from the names of the operations based on this assumption. Following standard naming conventions is quite typical in practice in the case of major service providers. For instance, the Amazon services follow the PascalCase coding style[8] (the names of operations are sequences of terms with the first letter of each term being capitalized). On the other hand, the Yahoo services follow the Java coding style[9]. Then, we measure the domain-level similarity between two operations with the following similarity function.

8. msdn.microsoft.com/en-us/library/x2dbyw72(v=vs.71).aspx
9. www.oracle.com/technetwork/java/codeconventions-135099.html

***Definition** 11:* **(Domain-level operation similarity)** Let T_{op_i} and T_{op_j} denote the sets of the domain-level terms that are extracted from the names of two operations, $op_i, op_j \in si.O$, of a service interface, si. The domain-level similarity between the two operations (Table 2(4)) is the Jaccard similarity for T_{op_i} and T_{op_j} (i.e., the size of the intersection divided by the size of the union of T_{op_i} and T_{op_j}).

Getting back to our example, the name of `GetQueueAttributes` consists of the action term `Get`, which is related with two domain-level terms, `Queue` and `Attributes`. The name of `SetQueueAttributes` comprises the action term `Set`, which is also related with `Queue` and `Attributes`. Therefore, the domain-level similarity between the two operations is $\frac{2}{2}$.

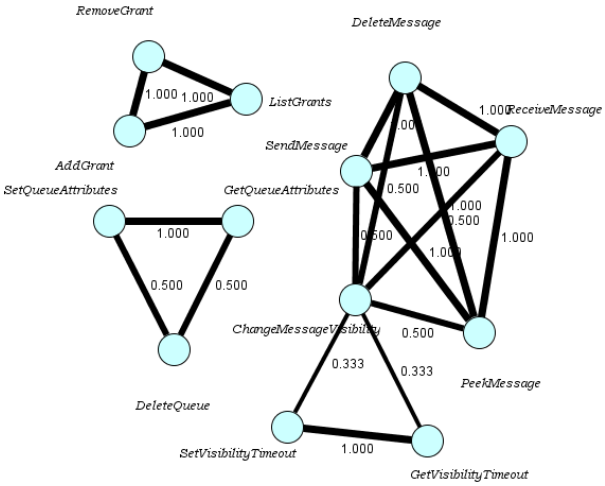


Fig. 6. $G^{dom}_{MessageQueue}$ for `MessageQueue`.

The refinement of the LoC_* metric, with respect to the domain-level similarity between two operations, is given below.

***Definition** 12:* **(Lack of domain-level cohesion)** The lack of domain-level cohesion, $LoC_{dom}(si)$, for a service interface, si, is an alias for $LoC_*(si, OpS_{dom})$. $LoC_{dom}(si)$ measures the relative difference between the interface-level graph, $G^{dom}_{si} = (V_{si}, E_{si}, OpS_{dom})$, defined with respect to the domain-level similarity function, OpS_{dom}, and the ideal interface-level graph, G^{dom}_{ideal}.

Concerning our example, in Figure 6, we have the interface-level graph that shows the domain-level relations for the `MessageQueue` interface. As in Figures 4 and 5 the graph is not complete. However, the domain-level relations are generally strong; the similarities between operations range from 0.3 to 1. Overall, the lack of domain-level cohesion is $LoC_{dom}(MessageQueue) = 0.81$.

Algorithm 1: $decomposeInterface()$

Input: $si : Interface$ /* An interface that is decomposed in more cohesive interfaces */
Output: $R_I = \{r : Interface\}$ /* The produced set of interfaces */
1. **var** $Q : Queue$ /* A queue that stores intermediate interfaces */
2. $R_I \leftarrow \emptyset$; $Q.enqueue(si)$
3. **repeat**
4. **var** $r_i : Interface$ /* holds an intermediate interface picked from the queue Q */
5. **var** $r_s : Interface$ /* the splinter interface that comprises operations removed from r_i */
6. **var** $r_r : Interface$ /* the interface that comprises the remaining operations of r_i */
7. $r_i \leftarrow Q.dequeue()$; $r_s \leftarrow null$; $r_r \leftarrow null$
8. $(r_s, r_r) \leftarrow createSplinter(r_i)$ /* *Phase 1: Returns the splinter r_s and r_r.* */
9. **if** $r_s = null$ **then**
10. $R_I \leftarrow R_I \cup \{r_i\}$ /* $LoC_*(r_i)$ can not be further improved. */
11. **else**
12. $(r_s, r_r) \leftarrow populateSplinter(r_s, r_r, r_i)$ /* *Phase 2: Populate r_s.* */
13. $Q.enqueue(r_r)$; $Q.enqueue(r_s)$
14. **end if**
15. **until** $Q.size() = 0$
16. **return** R_I

Algorithm 2: $createSplinter()$

Input: $r_i : Interface$ /* An intermediate interface picked from Q */
Output: $r_s : Interface$ /* The splinter interface that is created */
$r_r : Interface$ /* The interface that contains the remaining operations of r_i */
1. **var** $op_s : Operation$ /* The operation, whose removal maximizes the cohesion improvement of r_i */
2. **var** $\delta_{max} : Float$ /* The max cohesion improvement that can be achieved by removing an operation from r_i */
3. **var** r_{tmp} /* A temporary interface used to simulate the interface that results from the removal of an operation from r_i */
4. $r_s \leftarrow null$; $r_r \leftarrow null$; $\delta_{max} \leftarrow 0$; $op_s \leftarrow null$; $r_{tmp} \leftarrow null$
5. **for all** $op_i \in r_i.O$ **do**
6. $r_{tmp} \leftarrow$ **new** $Interface$
7. $r_{tmp}.O \leftarrow r_i.O - \{op_i\}$
8. **if** $LoC_*(r_i) - LoC_*(r_{tmp}) > \delta_{max}$ **then**
9. $\delta_{max} \leftarrow LoC_*(r_i) - LoC_*(r_{tmp})$
10. $op_s \leftarrow op_i$
11. **end if**
12. **end for**
13. **if** $\delta_{max} > 0$ **then**
14. $r_s \leftarrow$ **new** $Interface$; $r_s.O \leftarrow \{op_s\}$
15. $r_r \leftarrow$ **new** $Interface$; $r_r.O \leftarrow r_i.O - \{op_s\}$
16. **end if**
17. **return** (r_s, r_r)

4 Cohesion-Driven Decomposition

In this section, we detail the method that exploits the metrics defined in Section 3 for the cohesion-driven decomposition of service interfaces. Moreover, in Appendix C we provide a complementary analysis that concerns the stopping criteria and the complexity of the method.

From a broader perspective, the decomposition of a given service interface, si, into a set of more cohesive interfaces is a combinatorial optimization problem. The complexity of finding the optimal solution to this problem is exponential, since the powerset, $2^{si.O}$, of the set of operations of si should be examined. To deal with this issue, we employ a greedy approach, which progressively splits si in more cohesive interfaces (Algorithm 1). Note that although we assume as input a single service interface, the method can be easily applied in the case of a service that provides multiple interfaces. In such a case, the interfaces of the service can be merged into a single interface. Then, this interface can be given as input to the cohesion-driven decomposition method. As the decomposition proceeds, si is split in several interfaces, all of which are candidates to be further divided. To this end, we employ a queue, Q, which contains the interfaces that are candidates for decomposition (Algorithm 1, line 1). Initially, Q contains only the given interface, si (Algorithm 1, line 2). During each step (Algorithm 1, lines 4-16), the method dequeues from Q an intermediate interface r_i and checks whether it is possible to improve the cohesion of r_i, by removing a set of operations, which form a new interface r_s. Hereafter, we use the term *splinter interface* to refer to r_s, while r_r denotes the interface that comprises the rest of the operations of r_i.

The construction of the splinter interface takes place in two phases. The first phase, called $createSplinter$, checks if it is possible to improve the cohesion of r_i, by removing an operation (Algorithm 1, line 8). If this phase fails to find such an operation, ri is inserted in the results set, R_I (Algorithm 1, lines 9-10). Otherwise, the splinter interface, r_s, that contains the operation is returned as a result of $createSplinter$, along with the interface r_r that contains the remaining operations of r_i. The second phase, called $populateSplinter$, further improves the cohesion of r_s and r_r, by moving operations from r_r to r_s (Algorithm 1, line 12). Finally, the two interfaces, r_s, r_r are inserted in Q (Algorithm 1, line 13).

In further detail, the two phases of the decomposition are discussed below.

The $createSplinter$ phase accepts as input the intermediate interface, r_i, that is picked from Q (Algorithm 2). Following, it iterates over the operations of r_i (lines 5-12). Each iteration checks whether the removal of a single operation, op_i, from r_i improves the cohesion of the interface (line 8). To this end, the removal of op_i is simulated with the help of a temporary interface, r_{tmp}. Moreover, each iteration keeps track of the maximum cohesion improvement, δ_{max}, that can be achieved, and of the operation, op_s, that should be removed to achieve this improvement (lines 8-11). After this iterative process, if $\delta_{max} > 0$, the splinter interface, r_s, that contains op_s is created, along with the interface, r_r that contains the remaining operations of r_i (lines 13-16). The two new interfaces are returned as the results of $createSplinter$ (line 17). On the other hand, if is not possible to improve the cohesion of r_i, by removing an operation (i.e., $\delta_{max} = 0$), the results of $createSplinter$ are null.

The $populateSplinter$ phase accepts as input the intermediate interface, r_i, and the newly created interfaces, r_s, r_r (Algorithm 3). Then, it repeatedly moves operations from r_r to r_s (lines 8-27) as follows:

- The *populateSplinter* iterates over the operations of r_r (lines 11-23). Each iteration checks if an operation, op_i, can be moved from r_r to r_s. To perform this check, the movement of the operation is simulated with the help of two temporary interfaces, $r_{r_{tmp}}$, $r_{s_{tmp}}$. In particular, $r_{r_{tmp}}$ is employed to calculate the cohesion improvement, δ_{r_r}, that can be achieved for r_r, if the operation is moved (lines 12-14). Similarly, $r_{s_{tmp}}$ is employed to calculate the cohesion improvement, δ_{r_s}, that can be achieved for r_s (lines 15-17). The operation, op_i, is considered as a candidate to be moved if the following conditions hold (line 18): (a) the cohesion of r_r, after the move, is improved, i.e., $\delta_{r_r} > 0$, (b) the cohesion of r_s, after the move, is also improved, i.e., $\delta_{r_s} > 0$, and (c) the lack of cohesion of r_s, after the move, is smaller than the lack of cohesion of the intermediate interface r_i that was picked from Q. Each iteration further keeps track of the total cohesion improvement that can be achieved, by moving op_i, from r_r to r_s. Moreover, it keeps track of the operation op_s that maximizes the total cohesion improvement that can be achieved (lines 19-21).
- The operation, op_s, that maximizes the total cohesion improvement, δ_{total}, is moved from r_r to r_s (lines 24-26).
- The whole process stops when $op_s = null$ (line 27) and the updated r_s, r_r are returned (line 28).

Back to our example, Figures 7, 8 and 9, give the three different decompositions of `MessageQueue` that result based on LoC_{msg}, LoC_{conv} and LoC_{dom}, respectively [10]. In particular, the message-level decomposition of `MessageQueue` consists of 6 interfaces. The average lack of message-level cohesion of the interfaces is 0,92. Hence, an improvement has been made compared to the initial interface (Figure 4), but the improvement is small. This result is anticipated because the message-level relations between the operations of `MessageQueue` are not strong (Figure 4). The conversation-level decomposition consists of 7 interfaces. The average lack of conversation-level cohesion in this case is 0.88. Again, the improvement compared to the initial interface is small, because the conversation-level relations between the operations of `MessageQueue` are not strong (Figure 5). The domain-level decomposition of `MessageQueue` consists of 4 interfaces and the average lack of domain-level cohesion is 0.13. The improvement in this case is high, since the domain-level relations between the operations of `MessageQueue` are quite strong (Figure 6).

In the case of LoC_{dom}, the detailed execution of Algorithm 1 consists of 3 main steps. In

10. The input to the method was the 2007 version of the interface, aws.amazon.com/articles/Amazon-SQS/1148

Algorithm 3: *populateSplinter*()

Input: $ri : Interface$ /* An intermediate interface picked from Q */
$r_s : Interface$ /* The splinter interface that was created in Phase 1 */
$r_r : Interface$ /* The interface that contains the remaining operations of r_i */
Output: $r_s : Interface$ /* The populated splinter interface that comprises operations removed from r_r */
$r_r : Interface$ /* The interface that contains the remaining operations of r_i */

1. **var** $\delta_{r_r} : Float$ /* The cohesion improvement that can be achieved by removing an operation from r_r */
2. **var** $\delta_{r_s} : Float$ /* The cohesion improvement that can be achieved by adding an operation to r_s */
3. **var** $\delta_{total} : Float$ /* The total cohesion improvement $(\delta_{r_r} + \delta_{r_s})$ that can be achieved by moving an operation from r_r to r_s */
4. **var** $op_s : Operation$ /* The operation that is moved from r_r to r_s */
5. **var** $r_{r_{tmp}}, r_{s_{tmp}} : Interface$ /* Temporary interfaces used to simulate the interfaces that result after moving an operation from r_r to r_s */
6. $r_{r_{tmp}} \leftarrow null$; $r_{s_{tmp}} \leftarrow null$
7. /* Move operations from r_r to r_s */
8. **repeat**
9. $op_s \leftarrow null$; $\delta_{r_r} \leftarrow 0$; $\delta_{r_s} \leftarrow 0$; $\delta_{total} \leftarrow 0$
10. /* Find the operation op_s that improves the cohesion of r_r and r_s, and maximizes δ_{total} */
11. **for all** $op_i \in r_r.O$ **do**
12. $r_{r_{tmp}} \leftarrow$ **new** $Interface$
13. $r_{r_{tmp}}.O \leftarrow r_r.O - \{op_i\}$
14. $\delta_{r_r} \leftarrow LoC_*(r_r) - LoC_*(r_{r_{tmp}})$
15. $r_{s_{tmp}} \leftarrow$ **new** $Interface$
16. $r_{s_{tmp}}.O \leftarrow r_s.O \cup \{op_i\}$
17. $\delta_{r_s} \leftarrow LoC_*(r_s) - LoC_*(r_{s_{tmp}})$
18. **if** $((\delta_{r_r} > 0) \wedge (\delta_{r_s} > 0) \wedge (LoC_*(r_{s_{tmp}}) < LoC_*(r_i)))$ **then**
19. **if** $\delta_{r_s} + \delta_{r_r} > \delta_{total}$ **then**
20. $\delta_{total} \leftarrow \delta_{r_s} + \delta_{r_r}$; $op_s \leftarrow op_i$
21. **end if**
22. **end if**
23. **end for**
/* Move the operation op_s */
24. **if** $op_s \neq null$ **then**
25. $r_r.O \leftarrow r_r.O - \{op_s\}$; $r_s.O \leftarrow r_s.O \cup \{op_s\}$
26. **end if**
27. **until** $op_s = null$
28. **return** (r_s, r_r)

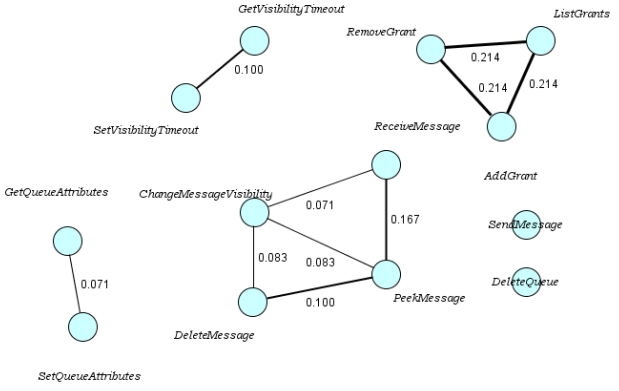


Fig. 7. Decomposition of `MessageQueue`, based on LoC_{msg}.

the first step, the general queue management operations (`DeleteQueue`, `SetQueueAttributes` and `GetQueueAttributes`) are removed from `MessageQueue`. These operations constitute the splinter interface, r_{s_1} (in Figure 1, this interface appears with the name `QueueMgt`). The remaining operations form r_{r_1}. Overall, the lack of cohesion of r_{s_1} is 0.33, while the lack

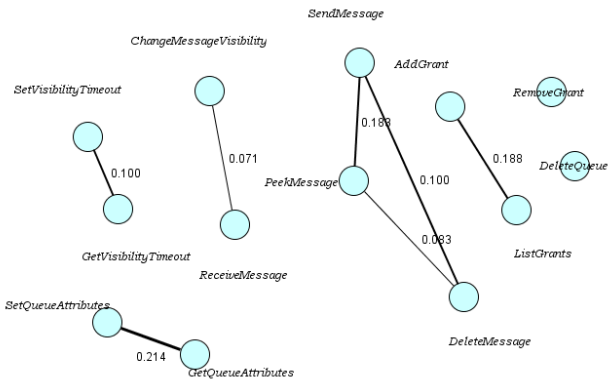


Fig. 8. Decomposition of `MessageQueue`, based on LoC_{conv}.

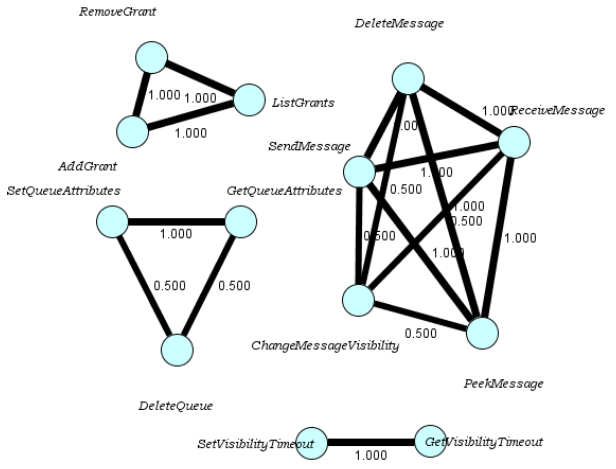


Fig. 9. Decomposition of `MessageQueue`, based on LoC_{dom}.

of cohesion of r_{r_1} is 0.72. In the second step, r_{r_1} is decomposed. In particular, the timeout management operations (`GetVisibilityTimeout` and `SetVisibilityTimeout`) are removed from r_{r_1}, and the splinter interface, r_{s_2}, is formed (in Figure 1, this interface is called `TimeoutMgt`). The rest of the operations of r_{r_1}, form r_{r_2}. The lack of cohesion of r_{s_2} is 0, while the lack of cohesion of r_{r_2} is 0.61. In the last step, r_{r_2} is decomposed, by removing the access rights management operations (`AddGrant`, `RemoveGrant` and `ListGrants`), which constitute the splinter interface, r_{s_3} (in Figure 1, this interface is named `GrantsMgt`). The rest of the operations (`SendMessage`, `ReceiveMessage`, `PeekMessage`, `DeleteMessage` and `ChangeMessageVisibility`) are related to messaging and form r_{r_3} (in Figure 1, this interface appears as `MsgMgt`). The lack of cohesion of r_{s_3} is 0, while the lack of cohesion of r_{r_3} is 0.2. To sum up, the results of the decomposition of `MessageQueue` are $R_I = \{r_{s_1}, r_{s_2}, r_{s_3}, r_{r_3}\}$.

5 Validation

To validate the proposed approach we developed a prototype tool in Java which is available, upon request, under a GPL license [11]. Our validation is based on real-world services, provided by Amazon and Yahoo. Specifically, we selected services that provide interfaces with at least 10 operations. Overall, we used 11 Amazon services and 11 Yahoo services[12]. Hereafter, we use identifiers A1-A11 and Y1-Y11 to refer, respectively, to the interfaces of the Amazon and the Yahoo services that we used. Table 3 provides the mapping between the identifiers and the service interfaces, along with the sizes of the interfaces (i.e., the number of provided operations) and the values of LoC_{msg}, LoC_{conv} and LoC_{dom} for the interfaces.

In the rest of this section we detail our findings. In Section 5.1 we concentrate on the effectiveness of the proposed approach from a quantitative perspective. In Section 5.2, we discuss the usefulness of the approach from the developers' perspective. Finally, in Section 5.3 we discuss threats to validity.

TABLE 3
Amazon & Yahoo case studies.

(a) *Amazon: aws.amazon.com/*

Service Interface					
Name	*Size*	*ID*	LoC_{msg}	LoC_{conv}	LoC_{dom}
AmazonEC2PortType	87	A1	0.98	0.99	0.94
MechanicalTurkRequesterPortType	27	A2	0.92	0.84	0.83
AmazonFPSPortType	27	A3	0.92	0.97	0.96
AmazonRDSv2PortType	23	A4	0.91	0.96	0.56
AmazonVPCPortType	21	A5	0.95	0.98	0.82
AmazonFWSInboundPortType	18	A6	0.93	0.96	0.73
AmazonS3	16	A7	0.89	0.97	0.75
AmazonSNSPortType	13	A8	0.96	0.97	0.84
ElasticLoadBalancingPortType	13	A9	0.93	0.97	0.72
MessageQueue	13	A10	0.98	0.98	0.81
AutoScalingPortType	13	A11	0.96	0.98	0.79

(b) *Yahoo:developer.searchmarketing.yahoo.com/docs/V6/reference/*

Service Interface					
Name	*Size*	*ID*	LoC_{msg}	LoC_{conv}	LoC_{dom}
KeywordService	34	Y1	0.84	0.93	0.91
AdGroupService	28	Y2	0.84	0.94	0.65
UserManagementService	28	Y3	0.96	0.97	0.91
TargetingService	23	Y4	0.74	0.96	0.74
AccountService	20	Y5	0.92	0.98	0.88
AdService	20	Y6	0.79	0.89	0.88
CampaignService	19	Y7	0.83	0.91	0.91
BasicReportService	12	Y8	0.91	0.99	0.92
TargetingConverterService	12	Y9	0.84	0.80	0.53
ExcludedWordsService	10	Y10	0.72	0.81	0.54
GeographicalDictionaryService	10	Y11	0.79	0.99	0.65

5.1 Effectiveness

To assess the effectiveness of the approach from a quantitative perspective we focus on the following research questions:

RQ1: To what extent is cohesion improved by adopting the proposed method ?

RQ2: Is the number of produced interfaces reasonable with respect to the size of the decomposed interface ?

11. For information on requesting a copy of the tool see www.cs.uoi.gr/~dathanas/software/software.htm

12. The WSDL specifications of the Amazon and the Yahoo services can be found at: www.cs.uoi.gr/~zarras/WS-Decomp-Material/

To respond to these questions we decomposed the examined service interfaces, based on the metrics that we defined in Section 3, and the method that we detailed in Section 4. To address RQ1, we measured the *cohesion improvement*, $CI(si)$, that is achieved for a service interface si. Formally, for a set of interfaces, R_I, produced by the proposed method for si, the cohesion improvement is: $CI(si) = \frac{LoC_*(si,OpS_*) - \frac{\sum_{r \in R_I} (LoC_*(r,OpS_*))}{|R_I|}}{LoC_*(si,OpS_*)} * 100\%$. To address RQ2, we measured the number of interfaces, $DS(si) = |R_I|$, produced by the proposed method for si. We further examined the relation between the number of operations offered by si (the independent variable) and $DS(si)$ (the dependent variable), using ordinary least squares regression (OLS). Hereafter, we use the term, *decomposition of* si, to refer to the set of interfaces, R_I, that is produced by the proposed method for si. Moreover, we use to term, *size of decomposition*, to refer to $DS(si)$.

RQ1: Figure 10(left col.), gives the values of CI that we obtained for the examined service interfaces. Concerning our first question, the combination of the proposed method with the domain-level cohesion metric (i.e., LoC_{dom}) was effective in all cases. The cohesion improvement for the domain-level decompositions is medium-high (CI ranges from 38% to 100%). The combination of the proposed method with the message-level cohesion metric (i.e., LoC_{msg}) was also effective in all cases. The cohesion improvement for the message-level decompositions is medium (CI is up to 41.9%). Finally, the combination of the proposed method with the conversation-level cohesion metric (i.e., LoC_{conv}) was effective in 77% of the cases. The cohesion improvement for the conversation-level decompositions is low. In 5 cases (A9, Y2, Y4, Y7, Y10), the similarities between the operations of the examined interfaces were such that the conversation-level cohesion of the initial interfaces could not be further improved.

RQ2: Figure 10(middle col.), gives the values of DS that resulted for the examined interfaces. Moreover, Figure 10(right col.) gives the results of the OLS analysis; in the x-axis of the scatter plots we have the number of the operations that are offered by the examined interfaces, in the y-axis we have the values of DS, and at the lower left corner of the scatter plots we have the regression equations and the values of the R^2 statistic. In general, the values of the R^2 statistic range from 0 to 1; high R^2 values indicate that a regression equation explains well the relationship between the variables involved in the equation. In our analysis, the values of the R^2 statistic are quite high (ranging from 0.71 to 0.89). Thus, the size of the decompositions, produced by the proposed method, linearly increases with the number of operations that are offered by the decomposed interfaces. The regression equations that we obtained for the different cohesion metrics are similar. The maximum value of the regression coefficients that could result from the OLS analysis is 1. A regression coefficient that equals to 1, would mean that the number of interfaces that are produced by the decomposition method equals to the number of operations of the decomposed interface. In our analysis, the regression coefficients are quite small, ranging from 0.33 to 0.35. Hence, the size of the produced decompositions is reasonable, with respect to the number of operations of the decomposed interface. Nevertheless, there are certain cases where the size of the produced decompositions is relatively high – see Fig. 10(middle col.). For instance, for the combination of the decomposition method with the domain-level cohesion metric we have the cases of A3 and Y3. Similarly, for the combination of the decomposition method with the message-level cohesion metric we have the cases of A3 and Y7. Finally, for the combination of the decomposition method with the conversation-level cohesion metric we have the cases of A2 and Y5.

5.2 The Developers' Opinions

To evaluate the usefulness of the approach from the developers' perspective we investigate the following research questions:

RQ1: Does the proposed approach produce useful results for the developers ?

RQ2: What are the developers' preferences (if any) concerning the metrics that are employed ?

RQ3: To what extent should the results be refined to fully satisfy the developers' needs ?

To address the aforementioned questions we looked for volunteers with the following skills: software development experience; knowledge of the service-oriented computing paradigm, related technologies and standards. Overall, 10 volunteers participated in our study. The participants had 3 to 15 years experience in software development. They were all familiar with the service-oriented computing paradigm. We organized the participants in two groups. The first group assessed the decompositions of the Amazon service interfaces, while the second group assessed the decompositions of the Yahoo service interfaces.

In a first meeting with the participants, we explained the overall purpose of the study, without giving any details, concerning the metrics and the method used for the decomposition of the examined service interfaces. Following, we gave to each participant a document [13] that contained the following information for each one of the examined interfaces: (a) a high-level description (represented as a UML class) of the interface; (b) the domain-level, the message-level and the conversation-level decompositions of the

13. The documents can be found at www.cs.uoi.gr/~zarras/WS-Decomp-Material/

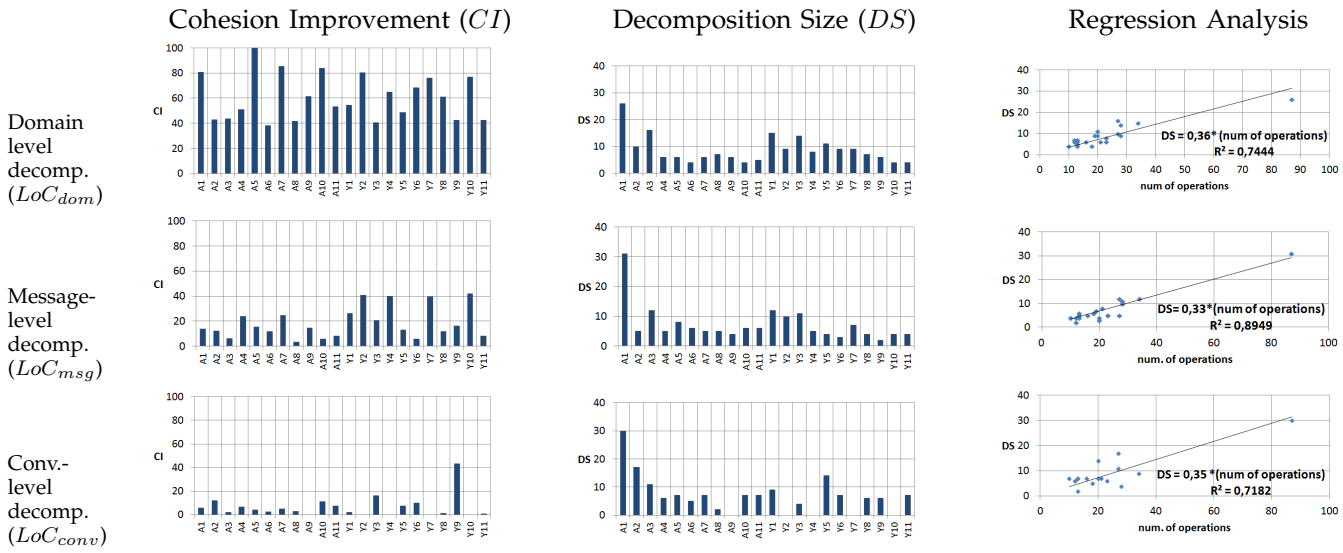
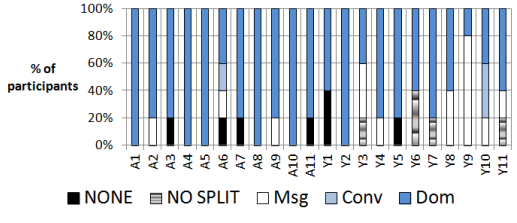
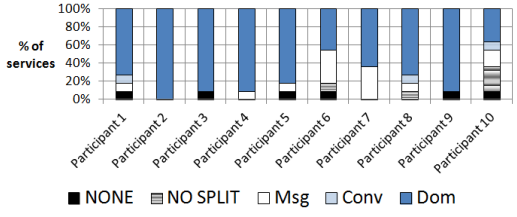


Fig. 10. Effectiveness Assessment.

interface. The decompositions were given in random order. The document further contained detailed instructions concerning the assessment tasks that should be performed for each one of the examined service interfaces. In the *first task*, the participants had to choose whether a service interface should be decomposed, or remain as is. In the *second task*, the participants had to report which of the provided decompositions is closest to their preferences; in this task the participants could also report that none of the provided decompositions is satisfactory. The *third task* was to suggest, if necessary, further changes on a selected decomposition.



(a) The choices made for each service.



(b) The choices made by each participant.

Fig. 11. Usefulness from the developers' perspective.

In a second meeting with each of the participants, we collected the documents and we analyzed the participants' feedback. The participants' feedback is summarized in Figure 11. In this figure we use the following notations that correspond to the possible choices that could be made by a participant for a particular service interface: `NO-SPLIT` - the participant suggested that the interface should not be decomposed; `NONE` - none of the provided decompositions was selected by the participant; `Msg` - the participant selected the message-level decomposition; `Conv` - the participant selected the conversation-level decomposition; `Dom` - the participant selected the domain-level decomposition. Figure 11(a), gives for each service the percentage of the participants that made a particular choice. Finally, Figure 11(b) gives for each participant the percentage of the services for which he/she made a particular choice.

RQ1: Concerning the first question, the participants suggested to decompose most of the examined interfaces. The only exceptions are Y3, Y6, Y7 and Y11 (Figure 11(a)). For Y3, Y7 and Y11, one of the participants suggested to leave the interface as is, while the others were in favor of decomposing the interface. For Y6, two of the participants suggested to leave the interface as is. For most of the service interfaces, the participants selected decompositions that were among the ones that we provided. In the Amazon services, we have 4 cases (A3, A6, A7, A11 in Figure 11(a)), for which one of the participants was not satisfied by any of the provided decompositions. For A6 and A7, the participants proposed their own decompositions. Specifically, the proposed decomposition for A6 was: "3 interfaces to manage fulfilment, items and shipments". For A7 the suggestion was: "3 interfaces for objects, buckets, access policies". In the Yahoo services, we have the case of Y1 (Figure 11(a)), for which two of the participants were not satisfied by any of the provided decompositions. The participants pointed out that the proposed decompositions do not separate clearly the underlying concepts (keywords,

bids, adGroups, optimization guidelines). Moreover, in the Yahoo services we have the case of Y5 (Figure 11(a)), for which one of the participants proposed his own decomposition: "three sets of operations one for the standard accounts, one for the mobile accounts and one for the credit cards".

RQ2: Regarding the second question, the domain-level cohesion metric worked very well for the participants. Specifically, in 63% of the services, more than 80% of the participants selected the domain-level decomposition (Figure 11(a)). Concerning each one of the participants, the percentage of the services for which the domain-level decomposition was selected ranges from 36% to 100% (Figure 11(b)). On the other hand, the percentage of the services for which the message-level decomposition was selected ranges from 0% to 36% (Figure 11(b)). Finally, the percentage of the services for which the conversation-level decomposition was selected ranges from 0% to 9% (Figure 11(b)).

RQ3: Concerning the third question, in several cases the participants did not suggest any changes in the selected decompositions. In the 55 decompositions that have been chosen for the Amazon services (5 participants $\times$ 11 services) there were 21 such occurrences; in the 55 decompositions of the Yahoo services this amounted to 18 occurrences. However, we also have several cases for which the participants moved certain operations between interfaces. In the 55 decompositions that have been chosen for the Amazon services there were 20 such occurrences; in the 55 Yahoo decompositions, this amounted to 14 occurrences. Moreover in several cases the participants decreased the size of the decompositions by merging certain interfaces. Specifically, we had 21 occurrences for the Amazon services and 24 occurrences for the Yahoo services. The details of the individual participants' suggestions are found in Appendix D.

TABLE 4
X^2 test for the overall results.

	NO-SPLIT	NONE	Msg	Conv	Dom
Observed	4.50%	5.40%	13.51%	2.70%	73.87%
Expected	20.00%	20.00%	20.00%	20.00%	20.00%
Squared diffs.	12.00	10.65	2.10	14.95	145.11
X^2	184.83				
ρ **value**	6.81E-39				

To conclude this study, we performed a X^2 test, so as to check the statistical significance of the results. The goal of the test was to examine the following null hypothesis:
H_0: The choices that have been made by the participants are not significantly different from the ones that we would have by chance alone.

Table 4 provides the details for the X^2 test that we performed. In particular, the first row of Table 4 gives the percentages of NO-SPLIT, NONE, Msg, Conv and Dom that we observed in the study in the overall 110 choices that have been made by the participants (22 services $\times$ 5 participants per service). The second row of Table 4 gives the percentages of NO-SPLIT, NONE, Msg, Conv and Dom, that we would have by chance alone. Based on the squared differences between the expected and the observed percentages, the overall X^2 value that we got is 184.83. Then, according to the X^2 distribution, the probability of having a X^2 as large as 184.83, by chance alone, is too small ($\rho \ll 0.001$). Therefore, we rejected H_0.

5.3 Threats to Validity

A possible threat to the internal validity of the results that we obtained from the developers' involved in the validation is the developers' fatigue or boredom. To reduce this threat we arranged our study according to the developers' availability, instead of imposing a strict schedule. To avoid effects caused by interactions between the developers, we made clear that the required tasks should not be performed in a collaborative manner. Finally, to avoid learning effects, the different decompositions of each interface were provided to the developers in a random order. Regarding external validity, our validation is among the very few ones [27], [6] that involve real services. Specifically, we used a representative set of services, provided by two major service providers; the services offer diverse functionalities and their interfaces vary in size and complexity. Moreover, we employed a representative set of developers that have knowledge of the service-oriented computing paradigm, related technologies and standards. On the other hand, a possible limitation is that the validation was not based on a large number of developers. Nevertheless, the number of developers that we considered is comparable with other similar studies [5], [9], [7].

6 Conclusion

Take away: In this paper, we have proposed an approach that enables the cohesion-driven decomposition of service interfaces, without information on how the services are implemented. Our experimental findings showed that the proposed approach is able to improve cohesion. The number of interfaces produced by the approach linearly increases with the size of the decomposed interface. In general, the developers found the proposed approach useful.

Limitations & future perspectives: As anticipated, the decompositions produced by the method are not perfectly adjusted to the developers' needs. In certain cases, the developers would prefer smaller and more cohesive decompositions, therefore there is further room to improve the proposed method. Future work can be pursued towards avoiding unnecessary splits and accounting for the user's positive/negative feedback. At the same time, although our approach is based on the practical assumption that only service

interface specifications are available, future research can address the problem of service interface decomposition, based on semantic annotations that could allow a better assessment of the functional relations between operations. Moreover, whereas we investigate the effect of different kinds of relations to service cohesion, one can possibly improve the results, via a combination of naming and structure similarity; finding the right combination involves studying several potential alternatives like linear/non-linear aggregate functions, single/multi-objective aggregate functions, etc. Finally, the decomposition of service interfaces could take into consideration other practical criteria like management costs and reusability.

REFERENCES

[1] W. Stevens, G. Myers, and L. Constantine, "Structured Design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[2] M. Papazoglou and W.-J. van den Heuvel, "Service-Oriented Design and Development Methodology," *International Journal on Web Engineering and Technology*, vol. 2, no. 4, pp. 412–442, 2006.

[3] C. Legner and T. Vogel, "Design Principles for B2B Services - An Evaluation of two Alternative Service Designs," in *Proceedings of the IEEE International Conference on Service Computing (SCC)*, 2007, pp. 372–379.

[4] T. Kohlborn, A. KortHaus, T. Chan, and M. Rosemann, "Identification and Analysis of Business and Software Services - A Consolidated Approach," *IEEE Transactions on Services Computing*, vol. 2, no. 1, pp. 1–15, 2009.

[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] D. Athanasopoulos and A. Zarras, "Fine-Grained Metrics of Cohesion Lack for Service Interfaces," in *Proceedings of the 9th IEEE International Conference on Web Services (ICWS)*, 2011, pp. 588–595.

[7] N. Tsantalis and A. Chatzigeorgiou, "Identification of Move Method Refactoring Opportunities," *IEEE Transactions on Software Engineering*, vol. 99, no. 3, pp. 347–367, 2009.

[8] F. Simon, F. Steinbrückner, and C. Lewerentz, "Metrics Based Refactoring," in *Proceedings of the 5th IEEE European Conference on Software Maintenance and Reengineering (CSMR)*, 2001, pp. 30–39.

[9] M. Fokaefs, N. Tsantalis, A. Chatzigeorgiou, and J. Sander, "Decomposing Object-Oriented Class Modules Using an Agglomerative Clustering Technique," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, 2009, pp. 93–101.

[10] D. Doval, S. Mancoridis, and B. S. Mitchell, "Automatic Clustering of Software Systems Using a Genetic Algorithm," in *Proceedings of the 9th IEEE Software Technology and Engineering Practice (STEP)*, 1999, pp. 73–81.

[11] B. D. Bois, S. Demeyer, and J. Verelst, "Refactoring: Improving Coupling and Cohesion of Existing Code," in *Proceedings of the 11th IEEE Working Conference on Reverse Engineering (WCRE)*, 2004, pp. 144–151.

[12] O. Seng, J. Stammel, and D. Burkhart, "Search-Based Determination of Refactorings for Improving the Class Structure of Object-Oriented Systems," in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, 2006, pp. 1909–1916.

[13] L. Tahvildari and K. Kontogiannis, "Improving Design Quality Using Meta-Pattern Transformations: A Metric-Based Approach," *Journal of Software Maintenance*, vol. 16, no. 4-5, pp. 331–361, 2004.

[14] M. O'Keeffe and M. í Cinnéide, "Search-Based Refactoring for Software Maintenance," *Journal of Systems and Software*, vol. 81, no. 4, pp. 502–516, 2008.

[15] M. Bowman, L. C. Briand, and Y. Labiche, "Solving the Class Responsibility Assignment Problem in Object-Oriented Analysis with Multi-Objective Genetic Algorithms," *IEEE Transactions on Software Engineering*, vol. 36, no. 6, pp. 817–837, 2010.

[16] W. F. Opdyke, "Refactoring Object-Oriented Frameworks," Ph.D. dissertation, Univ. of Illinois - Urbana Champaign, 1992.

[17] T. Mens and T. Tourwé, "A Survey of Software Refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[18] M. Harman and L. Tratt, "Pareto Optimal Search Based Refactoring at the Design Level," in *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, 2007, pp. 1106–1113.

[19] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[20] L. C. Briand, J. W. Daly, and J. Wüst, "A Unified Framework for Cohesion Measurement in Object-Oriented Systems," *Empirical Software Engineering*, vol. 3, no. 1, pp. 65–117, 1998.

[21] J. A. Dallal and L. Briand, "A Precise Method-Method Interaction-Based Cohesion Metric for Object-Oriented Classes," *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 2, 2012.

[22] A. Kazemi, A. Rostampour, A. Zamiri, P. Jamshidi, H. Haghighi, and F. Shams, "An Information Retrieval Based Approach for Measuring Service Conceptual Cohesion," in *Proceedings of the 11th IEEE International Conference on Quality Software (QSIC)*, pp. 102–111.

[23] Y. Ma, k. Lu, Y. Zhang, and B. Jin, "Measuring Ontology Information by Rules Based Transformation," *Knowledge-Based Systems*, vol. 50, no. 0, pp. 234–245, 2013.

[24] G. Bordogna and G. Pasi, "A quality driven hierarchical data divisive soft clustering for information retrieval," *Knowledge-Based Systems*, vol. 26, no. 0, pp. 9–19, 2012.

[25] L. Briand, S. Morasca, and V. R. Basili, "Property-Based Software Engineering Measurement," *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68–86, 1996.

[26] G. Valiente, *Algorithms on Trees and Graphs*. Springer, 2000.

[27] M. Fokaefs, R. Mikhaiel, N. Tsantalis, E. Stroulia, and A. Lau, "An Empirical Study on Web Service Evolution," in *Proceedings of the 9th IEEE International Conference on Web Services (ICWS)*, 2011, pp. 49–56.

Dionysis Athanasopoulos pursuits his Ph.D. at the Univ. of Ioannina. His research interests include software design principles, software maintenance and service-oriented computing.

Apostolos V. Zarras is an assistant professor at the Univ. of Ioannina. His research interests include software architecture & design, software maintenance and middleware.

George Miskos received his M.Sc. degree from the Univ. of Ioannina. His research focuses on service-oriented computing.

Valerie Issarny is a research director at Inria Paris-Rocquencourt. Her research focuses on the architecture-based development of software-intensive distributed systems.

Panos Vassiliadis is an associate professor at the Univ. of Ioannina. His research focuses on the rigorous modeling of data, software and their interdependence.