AASII: a novel algorithm for assessing services interfaces integrability

Marco Crasso[1,2], Cristian Mateos[1,2], Alejandro Zunino[1,2], and Marcelo Campo[1,2]

[1] ISISTAN Research Institute. UNICEN University. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel.: +54 (2293) 439682.
[2] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

Abstract. Current main and preferred trends in software development are the integration of prefabricated software components, called services. From the Service-Oriented Computing paradigm and its satellite paradigms, namely Software As A Service and Process As A Service, to the Cloud Computing paradigm, the integration of third-party software services into applications is an essential but at the same time non-trivial task mostly because in practice an interface is the only available description of a service. This paper presents an algorithm for assessing the integrability of services that works by exploiting the information present in services interfaces. An evaluation of the proposed algorithm from a theoretical perspective is provided, and the computational complexity of the proposed algorithm is also analyzed along with an example of its usage in relevant integrability scenarios.

Keywords: COMPONENT-BASED SOFTWARE ENGINEERING; SERVICE-ORIENTED COMPUTING; INTEGRABILITY.

1 Introduction

It is quite clear that in the last decade many researchers have focused on computing paradigms to develop software systems by integrating already developed components. IEEE defines integration as "the process of combining software components, hardware components, or both into an overall system" [1]. Integrability is defined as the capability of linking together different software components or services to act as a coordinated whole. The concept of integrability is essential for both already established computing paradigms, like component-based software development, and emerging ones, like service-oriented computing.

Component-based software development is a branch of software engineering for building software in which functionality is split into a number of logical software components with well-defined interfaces [2]. An interface consists of an specification of the set of operations signatures that a component offers to the outer world. An operation signature describes the input and output data messages exchanged by a particular operation. Components are designed to hide their associated implementation, to not share state, and to communicate with other components via message exchanging. The spirit of the component-based paradigm is that application components only know each other's interfaces, thus high levels of flexibility and reuse can be achieved.

More recently, the Service-oriented Computing (SOC) paradigm has evolved from component-based notions to face the challenges of software development in heterogeneous distributed environments, suggesting unprecedented levels of reusability [3]. A service-oriented application can be viewed as a component-based application that is created by integrating two types of components: *internal*, which are those locally embedded into the application, and *external*, which are those statically or dynamically bound to services. A service can be defined as a piece of functionality done by an external provider who is specialized in the management of this operation. Besides, each service is wrapped with a network-addressable interface that exposes its capabilities to the outer world while hiding implementation details that may restrict interoperability. When building a new application, a software designer may decide to provide an implementation for some application components, or to integrate an existing implementation instead. From now on, we will refer to this latter as *outsourcing*. In this context, to outsource a component C means to integrate an existing service S that fills the hole left by the missing functionality, which is represented by C [4].

As there may be many publicly available services that serve to fill the missing pieces of a particular application, an early problem is how to allow developers to effectively select the most integrable service. Note that addressing this problem would minimize the impact of outsourcing on the software life cycle, in particular on development and maintenance. In this direction, the work presented in [5] allows assessing different quality attributes from services interfaces, and defines reusability and modularity as the most important features of a service for a system integrator. The author presents metrics for calculating these features from services interfaces descriptions. However, these metrics are modeled as ordinary functions of one variable, thus they can not be employed to answer how does a component S integrates into an application expecting another component C. Moreover, it is difficult to assess which is the most integrable component for C from a set of candidate services $S_1, ..., S_n$.

At the same time, other efforts have striven to measure interfaces similarity, which to some degree alleviates the task of integrating services, since intuitively the more similar the services interfaces, the more integrable they are. Related works focus on assessing how semantically and syntactically similar the operations names and exchanged data between two services are [6]. These metrics operate by summing similarity between pairs of matched operations. A matched operation is an operation from an interface C for which one operation of an interface S has been assigned as its correspondence. However, to the best of our knowledge the similarity metrics proposed so far fail at quantifying the integrability quality attribute, as they do not consider the implications of non-matched operations. A non-matched operation represents a responsibility of one of the components for which a link to a responsibility belonging to the other component could not be established. Non-matched operations hinder the capability of linking together different services interfaces to act as a coordinated whole, since they may represent different mismatch patterns [7] i.e. compatibility problems from a functional standpoint between two services.

This paper presents a novel algorithm, called AASII, for assessing how compatible, in integrability terms, two interfaces C and S are. The proposed algorithm bases on a function to calculate the similarity among a pair of operations from both C and S in-

terfaces, an assignation problem solver [8], and the *Integrability Metric* (IM), which is originally introduced by this paper too. The IM has been designed not only to consider operation similitude values, but also to penalize non-matched operations. Moreover, the IM considers the relevance of non-matched operations. The proposed algorithm is flexible in the sense that it can be employed in conjunction with any previously defined operation similarity function provided this latter adheres to the *Nonnegativity* property, in concordance with most of those functions surveyed in [6].

This paper is organized as follows. Section 2 describes the approached problem. The AASII algorithm and mathematical properties of the IM are presented in Section 3. Section 4 demonstrates AASII usage through relevant integrability scenarios. Section 5 surveys relevant works on assessing software integrability. Finally, Section 6 presents concluding remarks and future research opportunities.

2 Problem statement

Recently, different programming models for developing service-oriented applications have been proposed. In particular, in [4] the authors introduce a programming model that conceptually organizes service-oriented applications as a set of coordinated components working together. From this set, some components –i.e. the internal ones– are locally implemented, whereas others –i.e. the external ones– are linked to third-party services. The main requirement of this model is that developers must define the desired interface of each external component, before actually knowing the details of potential candidate services.

In an open world setting, where services are built by different organizations, it is not necessarily true that all the available implementations of an abstract functionality expose the same public interface [9]. Because of this, in [4] the authors propose to use a layered architecture that introduces an intermediate layer to abstract away potential mismatches between those internal components that depend on a desired external interface, and potential target services. Accordingly, the replacement of a service only requires to modify the intermediate layer, thus internal components remain untouched.

The programming model presented in [4] helps to improve service-oriented applications maintenance, but at the expense of a higher effort to develop the intermediate layer, which is commonly materialized using the Adapter design pattern. Here, a service adapter accommodates the actual interface of a service to the interface expected by internal components. In other words, service adapters carry the necessary logic to transform the operation signatures of the interfaces expected by internal components to the actual interfaces of selected services. For instance, if a service operation returns a list of integers, but the application expects an array of floats, a service adapter would perform the type conversion.

To exemplify this, Figure 1 depicts a desired interface C, which is called from the internal components C' and C''. The adapter A is the component responsible for adapting C to the interface provided by a concrete service S. Concretely, this means that for an operation of C, there is an operation within A that performs the necessary conversions for preparing the input expected by the operation of S that best matches the operation of C, and after calling S operation, A will prepare the obtained result. Afterwards, each

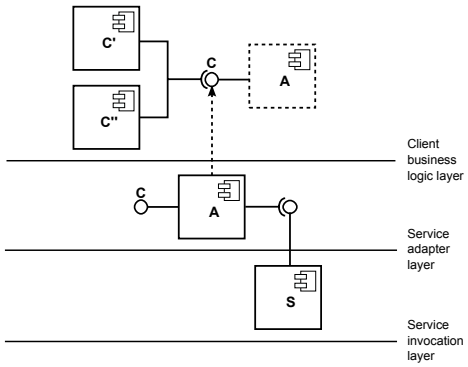

Fig. 1. Integrating a service into a service-oriented application using the model proposed in [4].

time the internal component of the service-oriented application C' and C'' call an operation offered by C, they will be interacting with A, irrespective of which adaptation process is being executed to reach a concrete provider such as S.

In the previous scenario, the cost of developing service adapters is driven by how integrable a candidate service interface and a desired (client-side) interface are. Some intuitive criteria for assessing this cost includes considering:

- the kind and amount of types conversions performed within the adapter,
- the difference on the number of operations provided by each interface,
- the number of matched operations and the number of non-matched ones,
- the relevance of non-matched operations in terms of how many internal components call them,
- the degree of functional similitude between the operations of C and S.

In this context, the integration cost can be defined as the total effort spent on defining inter-component links [10]. However, though the intuitiveness and essentialness of having a systematic procedure to assess the integrability between services interfaces, as far as we known this is still an open problem and developers should rely exclusively on their expertise.

3 AASII: Algorithm for Assessing Services Interfaces Integrability

This section describes a novel algorithm for assessing services interfaces integrability called AASII. The algorithm takes as input three elements. First, two interfaces, namely C and S, where C stands for the interface expected by the internal components of a service-oriented application, whereas S represents a candidate service interface. The third input consists of details about the relevance of each operation offered by interface C. Commonly, the relevance of an operation is computed as its individual fan-in. Then, given these inputs the AASII algorithm performs three steps and returns an integrability metric value. The three steps can be summarized as:

1. calculate an operations similitude matrix, multiply it by operations relevance, and normalize it,
2. calculate the best possible assignation between C operations and S operations,
3. calculate the integrability metric.

The first step receives C and S, and a vector of operations relevance $r =< r_1, .., r_{n_c} >$, in which r_i stands for the importance of the i^{th} operation op_i belonging to interface C. Commonly, r_i represents the number of calls that the operation receives from the system that owns C. The first step then returns μM, an $n_c \times n_s$ matrix, where n_i is the number of operations offered by either interfaces. The μM matrix results from the normalization of the *operations similitude matrix* M. The operations similitude matrix M comprises $n_c \times n_s$ cells, in which$cell_{i,j}$ contains the similitude value between the i^{th} operation of the first interface and the j^{th} operation of the second interface. For calculating the similitude between two operations, AASII can employ any similitude function available in the literature, provided it is a function $f : op_i \times op_j \rightarrow V_{sim}$, where V_{sim} is the domain for the similitude values among operations, and the following property is satisfied:

$$\forall i, i\varepsilon[1, n_c] \wedge \forall j, j\varepsilon[1, n_s] : f(op_i, op_j) \geq 0 \quad (1)$$

The format of the interfaces given as input depend on the function $f : op_i \times op_j \rightarrow V_{sim}$. In the case, for example, of employing an $f : op_i \times op_j \rightarrow V_{sim}$ that receives interfaces descriptions expressed in the Web Service Description Language (WSDL), then AASII requires C and S in WSDL as well. Alternatively, UML an any other specification language could be employed.

Prior to normalize the operations similitude matrix M, each $cell_{i,j}$ of the matrix is multiplied by the r_i element of r. By doing this, each similitude value is weighted in accordance with the relevance of C operations. Clearly, when the relevance of all the operations of C is the same, this step can be omitted.

Finally, in order to obtain the normalized operation matrix μM from M, each cell value is divided by the sum of all cells values, formally: $\mu M_{i,j} = \frac{M_{i,j}}{\sum_{c=1}^{n_c} \sum_{s=1}^{n_s} M_{c,s}}$.

The second step consists on calculating the best matching operation. The input of this step is the normalized operations similitude matrix μM. On the other hand, the output of this step are two vectors a and v, called the *assignations vector* and the *operation similitude values vector*, respectively. A component a_i of the assignations vector contains a number value in the range $[0, n_s]$, meaning that the i^{th} operation of the C interface should be assigned to the a_i operation of the S interface, or 0 when no target operation was found. Since calculating $a =< a_1, ..., a_{n_c} >$ is analogous to a classic assignation problem, AASII can employ, for instance, the well-known Hungarian algorithm [8] to find best possible assignations. At the same time, a component v_i of the operation similitude values vector points to the cell $\mu M_{i,a_i}$ for each i provided $a_i \neq 0$.

It is worth noting that in the context of interface integrability is rather rare to assign, and in turn bind, two or more source operations to the same target operation. Then, it is not mandatory that all the operations offered by the first interface (C) be assigned to an operation of the interface S. This situation occurs when $n_c > n_s$. Therefore, the length of the vector a, i.e. $|a|$, is always equal to n_c, whereas $|v| \leq |a|$, because the operation similitude values vector does not contain components for those operations that could

not be assigned, i.e. when $a_i = 0$. For the sake of readability, the constant *NA* represents the number of not assigned operations, i.e. $|v|$, the length of the operation similitude values vector.

Finally, the third step returns a number representing how well the interface S integrates to those components that are already integrated to interface C. We define this value as the Integrability Metric (IM) (Equation 2). For calculating the IM, the operations relevance vector r, the normalized operations similitude matrix μM, the operation similitude values vector v, and both interfaces sizes are required. Conceptually, the IM considers not only how well the assigned operations match with regard to the employed $f : op_i \times op_j \rightarrow V_{sim}$, but also considers how many operations of C could not be matched onto operations of S, vice versa. Moreover, the IM considers whether non matched operations are relevant. Formally:

$$IM(\mu M, v, n_c, n_s) = (\sum_{i=1}^{n_c} sim(op_i)) \times \frac{1}{\alpha} \times \frac{\sum_{i=1}^{n_c} rel(op_i)}{\sum_{i=1}^{n_c} r_i} \times \frac{1}{\beta} \times [1 - (\frac{n_s - A}{n_s})] \quad (2)$$

$$sim(op_i) = \begin{cases} 0 & a_i = 0 \\ \mu M_{i,a_i} & a_i \neq 0 \end{cases}$$

$$rel(op_i) = \begin{cases} 0 & a_i = 0 \\ r_i & a_i \neq 0 \end{cases}$$

, where $\mu M_{i,a_i}$ stands for the normalized value that results from calculating $f(op_{c,i}, op_{s,a_i})$, and A represents the number of operations for which the assignation problem solver algorithm could assign one target operation at step 2 (i.e. $|v|$). The constants α and β allows to ponder the significance of having non-matched operations within C and S, respectively. This is because in most scenarios having a dangling operation in C might be worst than calling an external service S with extra (non invoked at all) operations. A dangling operation in C means that the required functionality represented by the operation could not be fulfilled with the overall functionality offered by S.

3.1 Theoretical validation of the IM

The IM is a new proposed software engineering metric, as such its usefulness should be proved by a validation process. In the software engineering field, there are several works for theoretical validation of metrics, most of which consider different criteria, and are constituted by the definition of mathematical properties to which a software engineering metric should adhere [11,12].

In [12] the authors propose a validation framework comprising three properties: *Null Value*, *Monotonicity*, and *Additivity*. The work is considered by several researchers as the most practical and popular formal validation framework for metrics [13]. Additionally, we consider two more metric properties, namely *Nonnegativity* and *Commutativity* [11].

Property 1. Null Value, the IM of two interfaces is null if the resulting assignation vector is the null vector $a =< 0, .., 0 >$.

Proof. The IM multiplies operation similitude values by the percentage of operations that could be assigned. Then, if the rate of assigned operations is zero, the mentioned multiplication is zero. Formally, $\forall i, i \varepsilon [1, n_c] \wedge a_i = 0 \rightarrow sim(op_i) = 0$.

Property 2. Monotonicity, the IM provides a scale of values.

Proof. The IM multiplies operation similitude values by the percentage of operations that could be assigned. This multiplication grows as the operation similitude values or the percentage of operations that could be assigned grows. Then, the more the similar the operations of two interfaces or the higher the percentage of operations that could be assigned among them, the higher the IM. The lowest value for IM is 0, because of property 1. Instead, the higher value for IM is 1, because operation similitude values are normalized and in turn multiplied by factors ranging from 0 to 1.

Property 3. Not Additive, the IM of two interfaces C and S, is not necessarily equal to the sum of the IM values of the interfaces c_i and s_i, and c_j and s_j, where any operation of C or S is an operation of either c_x or s_x, and $n_c = \sum n_{c_x}$ and $n_s = \sum n_{s_x}$, respectively.

Proof. There are many different alternatives to split interfaces C and S into smaller ones, in terms of quantity of operations, while keeping $n_c = \sum n_{c_x}$ and $n_s = \sum n_{s_x}$. Each splitting alternative from C, i.e. c_x, will be differently matched onto s_x, i.e. the splitting alternatives that come from S. Then, each pair of splitting alternatives has its own operation similitude matrix, and assignation vector. However, in order to obtain smaller interfaces that the sum of their IM results equal to the IM of the original interfaces, the splitting alternatives should be arranged to reach the best operation assignation. Besides being impossible to do unless a global knowledge of the operations similitude is known, this means that there are at least one pair of splitting alternatives for which the best operation assignation is not obtained, therefore the sum of their IM would not be equal to the original.

Property 4. Nonnegativity, an IM value cannot be a negative number.

Proof. The IM is obtained by the sum of operation similitude values that are always non-negative numbers because of the first property of $f : op_i \times op_j \rightarrow V_{sim}$ (see Equation 1), and the product of positive factors. Therefore, this property is satisfied.

Property 5. Not commutable, the IM of two interfaces depends on the order used for evaluating them, regardless $f : op_i \times op_j \rightarrow V_{sim}$ is commutable.

Proof. The IM comprises the multiplication of three factors, namely the sum of operations similitude values, the percentage of assigned operations of the first interface, and the percentage of assigned operations of the second interface. The second of these factors depends on which order are given as input the two interfaces under evaluation.

3.2 Computational Complexity Analysis of the AASII

The computation complexity analysis of the AASII algorithm may be determined as the maximum complexity among calculating the operations similitude matrix (step 1), calculating the best assignations (step 2), and calculating the integrability metric (step 3).

The computational complexity required to compute the similitude matrix is the maximum between $O(n_c \times n_s)$ and $O(f)$, in which $O(f)$ represents the complexity of the operation similitude function employed $f : op_i \times op_j \rightarrow V_{sim}$. When using the Hungarian algorithm at step 2, the computational complexity needed to calculate the best possible assignations is $O(n_c^3)$. Finally, the integrability metric has complexity $O(n_c)$. Therefore, the computational complexity of the proposed algorithm is:

$$O(max(n_c \times n_s, f, n_c^3, n_c)) \tag{3}$$

The next section explains the usage of the proposed algorithm in four relevant integrability scenarios.

4 Integrability scenarios

Table 1. Example of normalized operations similitude matrix for the $n_c = n_s$ scenario

μM	$op_{s,1}$	$op_{s,2}$
$op_{c,1}$	0.21	0.07
$op_{c,2}$	0.57	0.14

For exemplification purposes, let us assume that the interface C has two operations, namely $op_{c,1}$ and $op_{c,2}$, and the interface S has also two operations: $op_{s,1}$ and $op_{s,2}$. This is the scenario when $n_c = n_s$. Also, let us assume that an specific operation similitude function returns the normalized operations similitude matrix μM shown in Table 1, and that the relevance of all operations is the same, e.g. $r =< 1, 1 >$. Furthermore, for simplicity, the constants α and β are set to 1.

By basing on μM, the best pair of assigned operations are $(op_{c,1}, op_{s,2})$ and $(op_{c,2}, op_{s,1})$, then the resulting assignations vector is $a =< 2, 1 >$ and $v =< 0.07, 0.57 >$. Having μM, a, and $n_c = n_s = 2$, the IM is calculated as $(0.07 + 0.57) \times (2/2) \times 1 - [(2-2)/2] = 0.64 \times 1 \times 1 = 0.64$.

Now, for exemplifying the $n_c < n_s$ scenario, let us assume that an interface S' containing all operations in S plus another operation $op_{s,3}$. The new operations similitude matrix μM is shown in Table 2. For the sake of simplicity, we have chosen the same relevance vector than before, i.e. $\forall i, i \varepsilon [1, n_c], r_i = 1$. Then, due to the operation added into S', the best assignation pairs are represented by the assignations vector $a =< 3, 1 >$ and the operation similitude values vector by $v =< 0.15, 0.3 >$. Then, the IM for C and S' is $(0.15 + 0.3) \times (2/2) \times 1 - [(3-2)/3] = 0.45 \times 1 \times (0.66) = 0.29$. As the reader

Table 2. Example of normalized operations similitude matrix for the $n_c < n_s$ scenario

μM	$op_{s,1}$	$op_{s,2}$	$op_{s,3}$
$op_{c,1}$	0.11	0.03	0.15
$op_{c,2}$	0.3	0.07	0.29

can see, the IM between C and S' falls with regards to the IM between C and S, because of the fact that having a "dangling" operation on one interface is penalized.

Table 3. Example of normalized operations similitude matrix for the $n_c > n_s$ scenario

μM	$op_{s,1}$	$op_{s,2}$
$op_{c,1}$	0.11	0.3
$op_{c,2}$	0.03	0.07
$op_{c,3}$	0.15	0.29

Let us consider the scenario of $n_c > n_s$, by incorporating a new operation $op_{c,3}$ into C, which results in interface C'. Table 3 presents the resulting operations similitude matrix. The assignation vector is $a =< 2, 0, 1 >$, which means that operation $op_{c,2}$ could not be matched onto any operation of S. As a consequence, the operation similitude values vector is $v =< 0.3, 0.15 >$. The IM for C' and S is computed as $(0.3+0.15) \times (2/3) \times 1 - [2-2]/2 = 0.45 \times (0.66) \times 1 = 0.29$. Likewise the $n_c < n_s$ scenario, a penalization is introduced in the IM because one operation of interface C' could not be matched.

As the reader can see, the previous two scenarios result in the same IM value. This stems from the fact that both constants α and β were established to the same value. Therefore, to penalize in a harder way the fact that there were non matched operations from C, i.e. $n_c < n_s$ scenario, then $\alpha > \beta$ should hold.

Finally, in this example will we show how the AASII algorithm can be employed for determining which interface is more integrable into another interface C when there are two candidate interfaces S and S'. Although the example may seem trivial, it allows generalizing the algorithm to deal with the comparison of n interfaces, in terms of integrability. Table 4 shows two pairs of matrixes. The $M_{c,s}$ and $M_{c,s'}$ matrixes represent the operation similitude values between the C interface operations and S and S' interfaces operations, respectively. Instead, the $\mu M_{c,s}$ and $\mu M_{c,s'}$ matrixes stand for the globally normalized operation similitude values. In this context, the term "globally" refers to knowing the overall operation similitude values achieved among all pairs of operations.

Once we have calculated the normalized operation similitude matrix for $< C, S >$ and $< C, S' >$, the next step is to compute an assignation vector for each pair of interfaces. The assignation vectors are $a_{c,s} =< 2, 1 >$ and $a_{c,s'} =< 1, 2 >$, and the operation similitude values vectors are $v_{c,s} =< 0.24, 0.03 >$ and $v_{c,s'} =< 0.22, 0.16 >$. The

Table 4. Example of operations similitude matrixes for three interfaces: C, S and S'.

$M_{c,s}$	$op_{s,1}$	$op_{s,2}$
$op_{c,1}$	0.3	0.1
$op_{c,2}$	0.8	0.2

$M_{c,s'}$	$op_{s',1}$	$op_{s',2}$
$op_{c,1}$	0.4	0.53
$op_{c,2}$	0.75	0.2

$\mu M_{c,s}$	$op_{s,1}$	$op_{s,2}$
$op_{c,1}$	0.09	0.03
$op_{c,2}$	0.24	0.06

$\mu M_{c,s'}$	$op_{s',1}$	$op_{s',2}$
$op_{c,1}$	0.12	0.16
$op_{c,2}$	0.22	0.06

next step consists on calculating the integrability metrics $IC_{c,s}$ and $IC_{c,s'}$, which results in 0.27 and 0.38, respectively. Accordingly, S' is more integrable into C than S.

5 Related work

Previous research in the area of service-oriented computing has emphasized on the importance of services interfaces and more specifically their non-functional concerns, although to the best of our knowledge the integrability quality attribute has not been specifically treated until now.

In [5] the author describes a metrics suite that consists of different kinds of metrics, ranging from common size measurements such as lines of code and number of statements, to metrics for measuring the complexity and quality of services interfaces. All the involved metrics can be statically computed from a service interface in WSDL, since the metric suite is purely based on WSDL schema elements occurrences. The most relevant complexity metrics included in the suite are Interface Data Complexity, Interface Relation Complexity, Interface Format Complexity, Interface Structure Complexity, Data Flow Complexity (Elshof's Metric), and Language Complexity (Halstead's Metric). On the other hand, the proposed quality metrics are Modularity, Adaptability, Reusability, Testability, Portability, and Conformity. Metrics results are expressed as a coefficient on a scale 0 to 1. For complexity metrics, 0 to 0.4 indicates low complexity, 0.4 to 0.6 indicates average complexity, 0.6 to 0.8 indicates high complexity and over 0.8 indicates that the service is not well designed at all. Instead, for quality metrics 0 to 0.2 indicates no quality at all, 0.2 to 0.4 indicates low quality, 0.4 to 0.6 indicates medium quality, 0.6 to 0.8 indicates high quality, and 0.8 to 1.0 indicates very high quality.

Regarding services interfaces complexity, [14] presents a metric suite whose cornerstone is that the effort required to understand data flowed to and from the interfaces of a service can be characterized by the structures of the messages used for exchanging and conveying the data. Basing on this statement, the authors define a suite comprising 4 main metrics, namely Data Weight (DW), Distinct Message Ratio (DMR), Message Entropy (ME) and Message Repetition Scale (MRS). These metrics can be statically computed from a service interface in WSDL, since this metric suite is purely based on

WSDL and XML Schema Definition (XSD), a language to structure XML content that offers constructors for defining simple types (e.g. integer and string), restrictions and both encapsulation and extension mechanisms to define complex elements.

There are more related efforts from within the component-based engineering area. For example, in [15] the authors attempted to design an interface complexity metric for components to quantify the complexity of a component-based system. Software complexity not only affects maintenance activities (software reusability, understandability, modifiability, testability) but also integrability. Given a software component, the proposed measure takes into account the interactions among other components, by borrowing notions from graph theory to illustrate interaction among software components and to compute complexity. The proposed metric relies in turn on two basic metrics, namely the Average Incoming Interactions Complexity (AIIC), and the Average Outgoing Interactions Complexity (AOIC).

The AIIC and AOIC metrics represent links among components, where a link means that one component sends a message and other receives it. For example, if a component C has an operation or method m_c, and m_c has a call statement to a method m_s of a component S, then there is a directed link from C to S. Then, the Average Interface Complexity of a Component Based System, i.e. AIC(CBS), is computed as:

$$AIC(CBS) = \frac{\sum_{i=1}^{m} AIIC_i}{m} + \frac{\sum_{i=1}^{m} AOIC_i}{m}$$

, where m is the number of components of the software system. Clearly, to compute AIIC and AOIC, one should have access to component implementations or have an specification of components dependencies. Having such "glassy" components is not the case in SOC, where black-box reuse is highly encouraged and services are third-parties components.

Similarly, in [10] the author defines measures for assessing the maintainability and integrability of a component-based system. The proposed metrics are Total number of components (TNC), Average number of methods per component (ANMC), Total number if implemented components (TNIC), Total number of links (TNL), Average number of links between components (ANLC), Average number of links per interface (ANLI), Total number of interfaces (TNI), Average number of interfaces per component (ANIC), Depth of the composition tree (DCT), and Width of the composition tree (WCT). The author arranges these metrics in four groups according to their purpose. The goal of TNC, ANMC, and TNIC is to characterize the components in the system. The TNL, ANLC, ANLI metrics are intended to characterize the connectors between components. To characterize interfaces in the system, TNI and ANIC are employed. Finally, DCT and WCT characterize the composition tree of the system. The proposed set of metrics help to assess quality attributes of the overall system, but not to assess whether two particular components could be integrated. Moreover, the second group of metrics requires to access components implementations, which as mentioned above is not always feasible in the context described in Section 2.

6 Conclusions and future research opportunities

The growing acceptance of the Service-Oriented Computing paradigm promotes the research on programming models and software metrics to develop and assess service-oriented applications, respectively. A service-oriented application can be seen as a component-based software system having internal components, those locally implemented, and external components, those that have been outsourced to a third-party service. During the life cycle of a service-oriented application, the selection and integration of external services into applications represent essential and non-trivial tasks.

This paper presented a novel algorithm for assessing the effort needed to integrate a service into an application that adheres to the programming model presented in [4]. The algorithm assumes that only services functional interfaces descriptions are known, as occurs in practice, though different approaches have been proposed to add semantically richer descriptions to services, but these have not been adopted by the industry yet. Conceptually, the algorithm regards on the degree of similitude between the operations of the external service and those expected by the internal components, the difference on the number of operations provided by the service interface and internal components expectations, the number of matched operations and the number of non-matched ones, the relevance of non-matched operation in terms of how many internal components call them. These aspects are assessed by the Integrability Metric (IM), a new software engineering metric. The algorithm outputs an integrability value, which can be used for determining which service is more integrable into an application from a set of candidate services. Therefore, this represents a step towards alleviating the selection and integration of external services into service-oriented applications.

The computational complexity of the proposed algorithm was presented. Besides, a framework for evaluating software engineering metrics was employed to evaluate the IM. Additionally, the usage of the proposed algorithm was illustrated using relevant integrability scenarios.

In the near future, we plan to conduct empirical evaluations of the proposed algorithm. Currently, we are gathering a data set of real-world, open source service-oriented applications. Besides gathering the applications, we are implementing the most relevant operations similitude functions found in current literature [6]. The ultimate goal is to analyze the implications of different similitude functions on the overall effectiveness of the algorithm, and whether the effort needed to integrate external services into gathered applications is reduced or not. Different measures, such as manpower and lines of code, will be calculated.

In another research direction, we plan to integrate the IM into the approach to discover services known as Query-By-Example (QBE) for services, which is surveyed in [6]. The QBE approach allows developers to find services by using a structured interface as query. Then, our hypothesis is that assessing the IM between a list of potential candidates and the query, i.e. an interface, could modify results ranking with regard to the effort needed to integrate the services.

References

1. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, page 1,

1990.

2. Hans-Gerhard Gross and Nikolas Mayer. Built-in contract testing in component integration testing. *Electronic Notes in Theoretical Computer Science*, 82(6):22 – 32, 2003. TACoS'03, International Workshop on Test and Analysis of Component-Based Systems.
3. Ivica Crnkovic, Judith Stafford, and Clemens Szyperski. Software components beyond programming: From routines to services. *IEEE Software*, 28:22–26, May 2011.
4. Marco Crasso, Cristian Mateos, Alejandro Zunino, and Marcelo Campo. Easysoc: Making Web Service outsourcing easier. *Information Sciences*, to appear, 2010.
5. Harry M. Sneed. Measuring Web Service interfaces. In *12th IEEE International Symposium on Web Systems Evolution (WSE), 2010*, pages 111 –115, September 2010.
6. Marco Crasso, Alejandro Zunino, and Marcelo Campo. A survey of approaches to Web Service discovery in Service-Oriented Architectures. *Journal of Database Management*, 22(1):103–134, 2011.
7. Woralak Kongdenfha, Hamid Reza Motahari-Nezhad, Boualem Benatallah, Fabio Casati, and Regis Saint-Paul. Mismatch patterns and adaptation aspects: A foundation for rapid development of Web Service adapters. *IEEE Transactions on Services Computing*, 2:94–107, April 2009.
8. Zvi Galil. Efficient algorithms for finding maximum matching in graphs. *ACM Computing Surveys*, 18:23–38, March 1986.
9. Luca Cavallaro and Elisabetta Di Nitto. An approach to adapt service requests to actual service interfaces. In *SEAMS'08*, pages 129–136. ACM, 2008.
10. Noel Salman. Complexity metrics as predictors of maintainability and integrability of software components. *Journal of Arts and Sciences*, 5:39–50, 2006.
11. E.J. Weyuker. Evaluating software complexity measures. *Software Engineering, IEEE Transactions on*, 14(9):1357 –1365, sep 1988.
12. L.C. Briand, S. Morasca, and V.R. Basili. Property-based software engineering measurement. *Software Engineering, IEEE Transactions on*, 22(1):68 –86, jan 1996.
13. G. Costagliola, F. Ferrucci, G. Tortora, and G. Vitiello. Class point: an approach for the size estimation of object-oriented systems. *Software Engineering, IEEE Transactions on*, 31(1):52 – 74, jan. 2005.
14. D. Baski and S. Misra. Metrics suite for maintainability of extensible markup language Web Services. *IET Software*, 5(3):320–341, 2011.
15. Usha Kumari and Shuchita Upadhyaya. An interface complexity measure for component-based software systems. *International Journal of Computer Applications*, 36(1):46–52, December 2011. Published by Foundation of Computer Science, New York, USA.