Efficient Measurement of Service Similarity

Joshua Church, Department of Computer Science, Volgenau School of Engineering, George Mason University, Fairfax, VA, USA

Amihai Motro, Department of Computer Science, Volgenau School of Engineering, George Mason University, Fairfax, VA, USA

ABSTRACT

The authors define a formal model for information services that incorporates the concept of service similarity. The model places services in metric spaces, and allows for services that have arbitrarily complex inputs and output domains. The authors then address the challenge of service substitution: finding the services most similar to a given service among a group, possibly large, of candidate services. To solve this nearest neighbor problem efficiently the authors embed the space of services into a vector space and search for the nearest neighbors in the target space. The authors report on an extensive experiment that validates both their formalization of similarity and their methods for finding service substitutions.

Keywords: *Information Services, Service Similarity, Service Substitution, Side-Effect-Free Service, Stateless Service*

INTRODUCTION

The increasing deployment of the service-oriented programming paradigm, in which programs are composed by weaving together distributed, platform-independent software components, raises the problem of service substitution: How to best replace one software component with another. The substitution may be motivated by a variety of reasons: An existing component might have failed, or there could be an alternative service that is of higher quality or lower cost. As the service-oriented paradigm gains popularity, the number of available services increases substantially, making the searches for substitutions more complicated and costly, and raising the need for methods and tools that facilitate such searches (Sillito, Murphy et al. 2008).

Finding the service (or group of services) that are most similar to a given service can be stated as a k-nearest neighbors problem. As such, it requires establishing a formal notion of service similarity. In this paper we describe a formal model for information services that incorporates the concept of service similarity, and we utilize this model for addressing the challenge of finding service substitutions.

DOI: 10.4018/jwsr.2013040102

Our model has several distinguishing aspects. First, the model considers services as *mappings* of inputs to outputs. A simple example is a service that returns the current temperature for a given US Zip code. These services are stateless and free of side-effects. Our view of services is thus quite similar to that advocated by the Representative State Transfer (REST) architectural style (Zhao and Doshi 2009). Services designed according to RESTful principles make resources available to clients through Web pages: Clients navigate to a URL, provide the necessary input values, invoke the service and receive a response. Since REST describes functionality with simple URLs rather than operations enveloped in complex XML standards, RESTful service implementations are becoming a popular alternative to SOAP/WSDL technologies. Unfortunately, formal models and techniques that prove so useful for developing services with SOAP/WSDL standards are absent for this type of service. So, as more data is made available via RESTful services, a formal model for information services is expected to be useful for RESTful services.

An important feature of the model is that the input and output of a service could be *arbitrarily complex*. For example, a service that returns the value of a stock portfolio given a particular date and a set of stock symbols and their corresponding quantities — has input which is a pair comprising a single value (date) and an arbitrarily-sized set of pairs (stock symbol and quantity). Another feature of the model is that the *semantics* of services need not be represented externally — by means of natural language descriptions, sets of keywords, ontologies, and so on. Instead, when necessary, semantics are inferred from observable behavior.

Our model defines a notion of service *similarity* that is based solely on observable behavior. Similarity is measured with distance *metrics*. Each basic domain (such as real numbers or character strings) is associated with a simple metric, and these metrics are then combined to create complex metrics that could be associated with domains of arbitrary complexity. Given two services with identically structured inputs and outputs, their similarity is derived from the similarity of their outputs for identical inputs. Additionally, the model is extended to deal with service *exceptions*: instances in which services might not provide valid outputs as expected.

A naïve solution to the problem of finding substitutions is to compare the observable behavior of the given service to each of the candidate services. To avoid long computations at the time when the substitutions are needed, similarity measurements should be prepared ahead of time. The cost, however, could be high, since metrics could be complex, and the number of services could be high. To overcome this challenge we describe a method that *embeds* the given set of services into a vector space, where the complexity of measurement is reduced significantly.

Two aspects of this work require experimental validation. The first is the *quality* of the discoveries. It must be demonstrated that our metrics indeed reflect the similarity inherent among services. That is, that services that behave similarly are indeed measured to have strong similarity. Our experiments show the precision of discoveries to be between 88% and 97%. The second aspect is the *performance* of our nearest neighbor methods. Our experiments show that the metric space embedding method reduces search cost by as much as 98%. This paper expands on an earlier version (Church and Motro 2013).

Our model is described in two sections: the third section defines services and the section after defines service similarity. Our approach to the challenge of service substitutions is described in the fifth section. The following section details and analyzes the experiments that validate our methods. The next section extends the work by relaxing key assumptions in the model, thereby allowing for a wider range of substitutions. Finally, the last section summarizes the results and sketches future research directions. We begin in this next section with a brief survey of related work.

BACKGROUND

Our work introduces a model for services, and it is therefore helpful to consider other approaches to this subject. The area of service modeling is vast and mostly out of the scope of this work, and we therefore focus on a small but diverse group of formal models for service discovery. These are models that examine service descriptions that are informative for search, and address the need to retrieve a relevant service.

The most popular approach to service description is the WSDL file. These files describe service operations and parameters as elements in standard XML format. In (Dong, Halevy et al. 2004) and (Blake and Nowlan 2007), custom similarity functions are designed based on information gleaned from a statistical analysis of large WSDL collections. A bag-of-words model is used in (Dong, Halevy et al. 2004) to find services that are within some edit distance to indexed WSDL files. In a similar vein, (Blake and Nowlan 2007) recommends substitutable services based on extended string matching that incorporates tendencies found in WSDL naming conventions. Searching for service alternatives is also our goal; however, our approach builds a model from service behavior. In addition, we query our model by *example*; that is, find substitutions using a description of behavior rather than keywords. The use of metrics for defining service similarity is discussed in (Liu, Shi et al. 2010), and (Gunay and Yolum 2007). The metrics in the former work are based on the similarity of terms that are extracted from the corresponding WSDL descriptions (as well as other related terms). The latter work measures service similarity based on the commonalities in the structures of the services (e.g., the internal processes used). In neither case is the similarity derived from observable behavior.

There are different interpretations of behavior. In (Shen and Su 2005) and (Junghans, Agarwal et al. 2012), behavior is defined as data flow between service invocations, statements of logic model these connections, and similarity is defined as Boolean combinations of predicates. In (Grigori, Corrales et al. 2006), graph theory is used to model behavior on the basis of the dependencies among service operations, and similarity is based on graph edit distance. In contradistinction, we represent behavior as sets of input/output observations. This approach is facilitated by instrumenting software as it runs, and then collecting invocation traces in a repository (Haran, Karr et al. 2005). This type of information can also be gleaned from deployment logs or other histories of program executions (Hassan, 2008).

Instead of modeling behavior, (Yu and Rege 2009) presents a service retrieval approach inspired by the relational model. The goal is to optimize QoS queries using techniques from database query processing. The research in (Yu and Rege 2009) assumes sets of functionally equivalent services, whereas our work seeks to find equivalent services. In addition, (Yu and Rege 2009) emphasizes differences in QoS parameter when choosing between equivalent services, whereas we emphasize differences in behavior.

An essential concept within our model is the similarity of services. The concept of similarity, it should be pointed out, has been researched extensively in other fields, including psychology (Tversky, 1977) and (Shepard, 1987). As services are software modules, it is worth mentioning that the notion of similarity is fundamental to many applications in software engineering, including code completion (Bruch, Monperrus et al. 2009), extracting source code snippets (Bajracharya, Ossher et al. 2010), visualizing software interfaces (McMillan, Grechanik et al. 2011), and indexing source code repositories (Grechanik, Fu et al. 2010).

The challenge we address — efficiently finding the services most similar to a given service— is an example of the widely researched nearest neighbor problem (Hjaltason and Samet 2003) (Zezula and Amato 2006). Our criterion for similarity is defined by metric spaces (O'Searcoid, 2006), where "nearest to a given service" is interpreted as the service with the smallest distance to the service. Previous applications of metric spaces to service-oriented computing focused on finding geographically

close services in ubiquitous computing environments (Tang and Crovella 2003) and (Kang, Kim et al. 2007). The goal is to make frequent service lookups efficient by balancing the load among service locators. Similar to our approach, services in (Kang, Kim et al. 2007) are treated as points in a metric space and then embedded into a vector space. However, similarity is not the focus of that work and it assumes that an ontology is available to model service properties.

SERVICES

There are different views as to what is a software service. The services considered here are information services. An information service is a function that receives data values as input and returns data values as output. As such, an information service is a data object encapsulated in specific input and output protocols. Alternatively, one could consider each service as a small database that is wrapped to process one type of query.

Let A and B be two domains, a service s is therefore a function s: $A \rightarrow B$.

An example is a service that receives values that are US Zip codes and returns the current temperature in the specified location. Another example is a service that receives a combination of location (latitude and longitude) and returns the location (latitude and longitude) of the three nearest gas stations, along with the names of their oil companies. A third example is a service that receives a date and a set of an arbitrary number of pairs each consisting a stock ticker symbol and the number of shares owned, and returns the total value of the portfolio on that date.

Domain Expressions

As the examples illustrate, the domains could have different structures. A domain could be *basic* (scalar) (e.g., Zip code), it could be a *sequence* (e.g., latitude and longitude), or it could be a *set of sequences* (e.g., portfolio). To allow complex input and output we define domains as expressions in which basic domains are combined with two operators: *aggregate* and *sequence*.

Given a domain α, the aggregation of α, forms a new domain, denoted $\{\alpha\}$, where each element is a set of elements of α.

Given domains $\alpha_1, \ldots, \alpha_n$, the sequencing of $\alpha_1, \ldots, \alpha_n$, forms a new domain, denoted $(\alpha_1, \ldots, \alpha_n)$, where each element is a sequence of n elements in which the ith element is from the domain α_i.

These two operations may be combined to create arbitrarily complex expressions from basic domains. Let α denote a basic domain, then domain expressions Δ are defined with this syntax:

$$\Delta ::= \alpha \left| \{\Delta\} \right| (\Delta, \ldots, \Delta) \tag{1}$$

Denote Z the domain of Zip codes, T the domain of temperatures (e.g., in degrees Fahrenheit), L_1 and L_2 the domains of latitudes and longitudes, respectively, G the domain of oil companies, D the domain of dates, S the domain of stock symbols, and M the domain of money (e.g., US Dollars). Then the three examples of services given earlier have these domains:

$$s_1 : Z \rightarrow T$$
$$s_2 : (L_1, L_2) \rightarrow \{(L_1, L_2, G)\}$$
$$s_3 : (D, \{(S, M)\}) \rightarrow M$$

Thus, the output domain of s_2 is a three column table and the input domain of s_3 is a date and a two-column table. Similarly, the expression $\{(\alpha_1, \alpha_2), \{(\alpha_3, \alpha_4, \alpha_5)\}\}$ defines a domain whose values are sets of pairs — the first element of each pair is a pair of scalars, and the second element is a set of triplets of scalars.

Signature and Behavior

A service comprises two components, called *signature* and *behavior*. Consider a service s with input domain α and output domain β. The signature of s is the pair (α, β) of input and

output domain expressions. The behavior of s is the set of values in the domain α and their matched values in the domain β: $\{(a, s(a)) \mid a \in \alpha\}$. An individual behavior pair $(a, s(a))$ is an instance of the service s^1.

The behavior of a service could be a large set, even infinite. For example, a service that provides the total rainfall in 2012 for each Zip code would have about 43,000 instances. A service that converts degrees Fahrenheit to degrees Celsius would have an infinite number of instances[2].

SERVICE SIMILARITY

A question that often arises in service-oriented software architectures is whether two given services are "similar" to each other; i.e., whether one service could substitute for the other (Church and Motro 2011). To answer this question we must define service similarity. We limit our discussion here to the similarity of services with identical signatures. Hence, the issue is how to define the similarity of two behaviors (of the same signature).

Similarity of Domain Elements

To measure the similarity between the elements of a domain we define a metric for that domain. A metric for a domain A is a function d from the product $A \times A$ to the real numbers that satisfies:

1. $d\left(a_1, a_2\right) \geq 0$
2. $d\left(a_1, a_2\right) = 0 \Leftrightarrow a_1 = a_2$
3. $d\left(a_1, a_2\right) = d\left(a_2, a_1\right),\ and$
4. $d\left(a_1, a_2\right) + d\left(a_2, a_3\right) \geq d\left(a_1, a_3\right)$

It should be noted that, at times, these four requirements may be too restrictive, and in some applications where distances need to be measured, some of the requirements are relaxed. For example, the requirement for symmetry (the third requirement) may be inappropriate when measuring driving distances between cities, because connecting roads could be different in each direction (the measure is called a *quasi-metric*). In some applications, the requirement that the distance between different points be non-zero (the second requirement) is relaxed (the measure is called *pseudo-metric*); for example, if we wish to measure the distance between two points in the plane based only on their first coordinate: $d((x_1,x_2),(y_1,y_2))=|x_1-y_1|$. An example of relaxing the triangular inequality (the fourth requirement) is in distances based on the co-occurrence of terms in documents (the measure is called a *semi-metric*). Noting that the distances here denote *behavior similarity*, it follows that all four requirements are sensible. For example, substituting a service s_1 with s_2 and then substituting s_2 with s_3 should not be any better than substituting s_1 with s_3 directly. Note, however, that in Section 5.2 and in Section 7.1 we consider distances that are not symmetric.

Our approach is to establish distance metrics for the basic (scalar) domains, and then extend these metrics to general domain expressions[3].

In this paper we consider two basic domains only: real numbers R and character strings S. We note that extending the work to other domains should not be difficult. Distances among real numbers will be measured with the *absolute value metric*, and distances among strings will be measured with the *Levenshtein edit distance* (Gusfield, 1997). These two metrics are extended to metrics for general domains as follows.

Let x, y be elements of domain α and assume that $\alpha = (\alpha_1, \ldots, \alpha_n)$; that is, the final domain operation to create α was sequencing. Then x and y are sequences: $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$. We define the distance between x and y as the *sum* of the distances between the *sequence* components in Equation 2:

$$d\left(x, y\right) = \sum_{i=1}^{n} d\left(x_i, y_i\right) \tag{2}$$

Let x, y be elements of domain α and assume that $\alpha = \{\alpha_1\}$; that is, the final domain operation to create α was aggregation. Then

x and y are sets: $x = \{x_1, \ldots, x_n\}$ and $y = \{y_1, \ldots, y_m\}$. We define the distance between x and y as the Hausdorff distance between the sets. That is, first we calculate the directed distance from x to y: For each element in x we choose the smallest of its distances to the m elements of y, and then we choose the largest of the n distances thus obtained; we similarly calculate the directed distance from y to x; and finally — to assure symmetry — we adopt the highest of the two distances in Equation 3:

$$d(x,y) = \max\{\max_{x_i \in x}\{\min_{y_j \in y} d(x_i, y_j)\}\}, \max_{y_j \in y}\{\min_{x_i \in x}\{d(y_j, x_i)\}\}\} \quad (3)$$

As the domains α_i used to create α could themselves be complex, these definitions are reapplied recursively, until all distances are calculated in basic domains, using the two basic metrics. To illustrate, assume a domain $\alpha = (S, \{R\})$ and two values from this domain: ('Jack', {1, 3, 5}) and ('Jill', {5, 8, 11, 14}). The Levenshtein distance between 'Jack' and 'Jill' is 3; the Hausdorff distance between {1, 3, 5} and {5, 8, 11, 14} is 9; and the overall distance between ('Jack', {1, 3, 5}) and ('Jill', {5, 8, 11, 14}) is therefore 3 + 9 = 12.

We note that the eventual distance function on α thus defined is indeed a metric. This follows from (1) the distances defined on the basic domains (absolute value and Levenshtein) are both metrics, (2) the distance between sequences as defined in Equation 2 is a metric[4], and (3) the distance between aggregates as defined in Equation 3 is a metric[5].

Finally, we define the *similarity* between two elements of a domain as the *reciprocal* of the distance in Equation 4:

$$sim(x_1, x_2) = 1 / d(x_1, x_2) \quad (4)$$

Similarity of Behavior

Recall that we interpreted the similarity of two services (of the same signature) as the similarity of their behaviors. Assume two services with identical signatures:

$$s_1 : A \rightarrow B$$
$$s_2 : A \rightarrow B$$

The similarity between s_1 and s_2 is defined as the *average* of the similarity of their outputs for all possible inputs. Let $|A| = n$. Then Equation 5:

$$sim(s_1, s_2) = \frac{1}{n} \sum_{a \in A} sim(s_1(a), s_2(a)) \quad (5)$$

As an example, assume a small domain $A = \{1, 2, 3, 4\}$ with the absolute value metric, a service s_1 that multiplies its input by 5, a service s_2 that multiples its input by 6, and a service s_3 that adds 10 to its input. Their behavior tables can be combined (see Table 1):

To calculate the similarity of any two services we average the similarity of their outputs for the four possible inputs: $sim(s_1, s_2) = 1/2.5$

Table 1. Example behavior table

A	s_1	s_2	s_3
1	5	6	11
2	10	12	12
3	15	18	13
4	20	24	14

= 0.4, $sim(s_1, s_3)$ = 1/4 = 0.25 and $sim(s_2, s_3)$ = 1/5 = 0.2.

Service Exceptions

Similar to null values in database tables, behavior tables may have missing values as well, where the service cannot deliver an output value for a particular value of the input. These missing values are termed *service exceptions*. Consider this small example in Table 2 in which service exceptions are denoted with —:

For calculating distances and similarities in the presence of exceptions, our approach is to substitute each distance that cannot be calculated due to an exception with the *maximal* known distance, adopting a cautious approach that prefers to understate the similarity than to overstate it. In the example:

$$d(b_2,-) = d(-,c_3) = d(b_5,-) = d(-,-) = \max\{d(b_1,c_1), d(b_4,c_4), d(b_6,c_6)\}$$

With distances now available, similarities are obtained by reciprocity and the overall service similarity by average:

$$sim(s_1,s_2) = \frac{1}{7}\cdot(sim(b_1,c_1) + sim(b_4,c_4) + sim(b_6,c_6) + 4\cdot\min\{sim(b_1,c_1), sim(b_4,c_4), sim(b_6,c_6)\})$$

Denote A_0 the subset of A in which both services does not have exceptions. Let $|A| = n$ and $|A_0| = n_0$. Therefore the number of instances in which at least one service has an exception is $n - n_0$. Then Equation 6:

$$sim(s_1,s_2) = \frac{1}{n}\cdot(\sum_{a\in A_0} sim(s_1(a),s_2(a)) + (n-n_0)\cdot\min_{a\in A_0}\{sim(s_1(a),s_2(a))\}) \quad (6)$$

Obviously, in the absence of exceptions, Equation 6 reduces to Equation 5. A shortcoming of this equation is that when the number of valid similarities n_0 is small, the measurement of service similarity is *less* robust. In an extreme case (which will become of interest in Section 7), we might even have $n_0 = 0$; that is, there are no valid similarities and $sim(s_1, s_2)$ is not well-defined. This could happen when on each value of the input domain one of the services reports an exception, or when one of the services is entirely inoperative. Preferring a cautious approach, when one value is exception, we define:

$$d(a,-) = \max\{d(a,x) \mid x \in B\}$$

when both values are exception:

Table 2. Behavior table with exceptions

A	s_1	s_2
1	b_1	c_1
2	b_2	—
3	—	c_3
4	b_4	c_4
5	b_5	—
6	b_6	c_6
7	—	—

$$d(-,-) = \max\{d(x,y) \big| x,y \in B\}$$

SERVICE SUBSTITUTIONS

In Sections 3–4 we defined a model for services with service similarity. We now consider the following problem: Given a service s_0 and a group of services S, find the service in S that is most similar to s_0. This *nearest neighbor* problem arises frequently in service-oriented software architectures, when one of the services in the architecture fails and needs to be replaced by the most similar service from a directory of available services. Ideally, this substitution (often called healing) would be done "automatically" (i.e., without programmer's intervention) using the most similar service. In practice, however, it is safer to allow the programmer to choose from several substitution alternatives, which is a *k*-nearest neighbor problem. Before we describe our approach to the problem, we discuss two pragmatic adjustments to our model.

Reducing the Dimension of Behavior Tables

The definitions of our model assumed that behaviors are available in their entirety, and to determine the similarity of two behaviors we must know their entire sets of instances. In practice, this assumption is impractical for a variety of reasons. First, in many instances services are stored as executable code, not as tables (a simple example is the aforementioned service that converts degrees Fahrenheit to degrees Celsius). Second, even when stored in tables, these tables are "wrapped" in protocols that permit queries that access few instances at a time. Finally, even when the tables are available, their sizes may be substantial thus requiring costly calculations.

A practical approach is to obtain *samples* of behavior: Typically, a sample of *size n* is obtained by a *sequence* of n single-instance queries to the service. In (Church and Motro 2011) we showed how relatively small samples (of about 16 instances) may be obtained that convey satisfactorily the "traits" of a behavior. In the following discussion we shall assume behaviors that are samples.

Directed Similarity

Our samples are obtained by individual requests to the service, and when a service request results in an exception, we enter an "exception" into the behavior table. However, keeping in mind that our main application is service substitution, our calculation of service similarity (behavior similarity) would no longer be symmetric.

When considering a substitution of s_1 with s_2, the similarity calculation will ignore all instances in which s_1 has an exception in its behavior; and when considering a substitution of s_2 with s_1, our calculation will ignore all instances in which s_2 has an exception in its behavior. The reason is that we assume that the service to be replaced is operating correctly, and its exceptions were a result of illegal out-of-range requests. We expect the "new" service to deliver valid output only in the instances in which the "old" service delivered valid output.

Denote A_1 the subset of A in which s_1 does not have exceptions and denote A_2 the subset of A in which s_2 does not have exceptions. Recalling that A_0 is the subset for which both services do not have exceptions, we have $A_1 \cap A_2 = A_0$.

Let $|A_1| = n_1$ and $|A_2| = n_2$. Therefore the number of instances in which s_2 has exception but s_1 does not is $n_1 - n_0$. The *directed similarity* from s_1 to s_2 is shown in Equation 7:

$$\vec{sim}(s_1, s_2) = \frac{1}{n_1} \cdot (\sum_{a \in A_0} sim(s_1(a), s_2 + (n_1 - n_0) \cdot \min_{a \in A_0}\{sim(s_1(a), s_2(a))\}) \quad (7)$$

Observe that $sim(s_1, s_2)$ in Equation 7 is derived from $sim(s_1, s_2)$ in Equation 6, by substituting n_1 for n in two places.

In the previous example, when s_2 is considered a substitute for s_1 the directed similarity would be:

$$\vec{sim}(s_1, s_2) = \frac{1}{5} \cdot (sim(b_1, c_1) + sim(b_4, c_4) + sim(b_6, c_6) + 2 \cdot \min\{sim(b_1, c_1), sim(b_4, c_4), sim(b_6, c_6)\})$$

whereas when s_1 is considered a substitute for s_2 the directed similarity would be:

$$\vec{sim}(s_2, s_1) = \frac{1}{4} \cdot (sim(b_1, c_1) + sim(b_4, c_4) + sim(b_6, c_6) + \min\{sim(b_1, c_1), sim(b_4, c_4), sim(b_6, c_6)\})$$

We begin with a naïve treatment of this k-nearest neighbor problem, and we then offer a more efficient solution.

Exhaustive Search

As mentioned earlier, the optimal service substitution problem assumes a given service s_0 and a group of services S, and finds the k services in S that are most similar to s_0; that is, services $s_1, \ldots, s_k \in S$, such that for $1 \leq i < k - 1$ $sim(s_0, s_i) \geq sim(s_0, s_{i+1})$ and for every other service $s \in S$ $sim(s_0, s) \leq sim(s, s_k)$.

The simplest solution, of course, is to conduct an exhaustive search. That is, calculate the similarity between s_0 and each service in the group S, while at each point maintaining a list of the top k services discovered so far.

Assume $|S| = n$. In practice, we pre-compute all n^2 similarities and store them in an $n \times n$ matrix. When it is necessary to substitute a service s_j, the jth row in the matrix is retrieved and top k values are extracted. The positions of these values point to the best substitutions. Note, however, that even the best substitutions could be too dissimilar to s_j, so in practice, we should define a *threshold* to aid the programmer in interpreting these suggested substitutions.

This "brute-force" algorithm has complexity $O(n^2)$. Note, however, that calculating each of the similarities, especially for services over complex domains, could become costly. Our next method attempts to reduce this cost.

Metric Space Embedding

A common method to reduce the effort of calculating distances (or similarities) between elements in a given metric space is to embed the elements in another metric space, so that the distances between the mapped elements are similar to the original distances, but are (hopefully) considerably simpler to calculate (Samet, 2006). Formally, let ψ be an embedding of the metric space (A, d) into the metric space (A', d'). We want to find an embedding ψ such that for any two elements $a_1, a_2 \in A$:

$$d(a_1, a_2) \approx d'(\psi(a_1), \psi(a_2))$$

where ≈ denotes approximation. Let (A, d) be a metric space. We extend the metric d to measure the distance between an element $a \in A$ and a subset $X \subseteq A$ as follows:

$$d(a, X) = \min_{x \in X}\{d(a, x)\}$$

That is, the distance between an element and a set is the minimum distance between the element and any element in the set. Now let $X_1, X_2, \ldots, X_m$ be subsets of A. We map every element a to a *vector* of its distances to these subsets:

$$\psi : a \rightarrow (d(a, X_1), d(a, X_2), \ldots, d(a, X_m))$$

Such a mapping is called a *Lipschitz embedding*, and $X_1, X_2, \ldots, X_m$ are its *reference sets*. The intuition is that the distance between two elements a_1 and a_2 will be captured adequately by the distance between their embedded vectors:

$$\begin{aligned} d(a_1, a_2) &\approx d'(\psi(a_1), \psi(a_2)) \\ &= d'((d(a_1, X_1), d(a_1, X_2), \ldots, d(a_1, X_m)), \\ &\quad (d(a_2, X_1), d(a_2, X_2), \ldots, d(a_2, X_m))) \end{aligned}$$

A particular version of a Lipschitz embedding commonly used in situations similar to ours selects references that are singleton sets. Typical metrics for the target vector space are either the Chebyshev metric, that measures the distance between vectors as the maximum absolute-value distance between corresponding coordinates, or any of the L_p metrics (of which the Chebyshev metric is a limit).

Denote the reference points $p_1, p_2, \ldots, p_m$. Then with the Chebyshev metric we have:

$$d(a_1, a_2) \approx d'(\psi(a_1), \psi(a_2)) = \max_{i=1,\ldots,m} \left\{ \left| d(a_1, p_i) - d(a_2, p_i) \right| \right\}$$

And with the Euclidean metric (L_2):

$$d(a_1, a_2) \approx d'(\psi(a_1), \psi(a_2)) = \sqrt{\Sigma_{i=1,\ldots,m} (d(a_1, p_i) - d(a_2, p_i))^2}$$

In practice, after choosing the m reference points (also called *pivots*), we pre-compute the $n \cdot m$ distances between each of the n services and each of the m pivots. These distances are then used to calculate all service-service similarities, which are stored in an $n \times n$ matrix as before. The new cost is now $O(m \cdot n)$. Assuming $m < n$, it is an improvement over $O(n^2)$.

EXPERIMENTATION

Two aspects of the work described so far require validation. First, it must be demonstrated that the method for constructing similarity measures for domains of arbitrary complexity, described in Section 4, generates measures of good quality. Then, it must be demonstrated that the k-nearest neighbor method described in Section 5 arrives at high quality results at low computational costs. This section describes experiments that seek to validate both these aspects.

The first challenge is to design an experiment that will determine whether our similarity measures are successful in conveying the similarity *inherent* in a set of services. That is to say, the set of services has an inherent *structure*, and the issue is whether our similarity measure can detect this structure. We translated this problem to a problem of *classification*: Assume a set S of n services, and a grouping of the services into m distinct groups, where the services in each group are constructed to be inherently similar to each other. Now, given a service $s \in S$, our similarity measure was used to find the k services in S that are most similar to s. The set of k discovered services was compared to the group to which s belongs, using the measure of *precision*: the ratio of services that are indeed in the group of s to the total number k of services discovered.

Two independent experiments were conducted, each with a different output signature. In the first experiment the services had an output signature of the type (S, $\{R\}$); that is, the output of each service was a character string followed by a set of numbers (which is similar to the example shown in Section 4). In the second experiment the services had an output signature of the type (R, $\{S\}$, $\{S\}$); that is, the output of each service was a triplet: a number followed by two set of strings (for example, a service that receives the GPS coordinates of a location and returns the Zip code of that location plus a set of movie theaters and a set of restaurants in that area).

To imbue services in the same group with similar behaviors, each group of services was randomly assigned the parameters of a distribution, and then each service in the group randomly chose values for its behavior instances from that distribution. For example, assume the output includes a set of numbers. The parameters of the distribution are two ranges: a range from which the cardinality of the set is chosen, and a range from which the values of the elements of the set are chosen. Thereafter, for each service in the group, a cardinality is randomly chosen from the first range and a corresponding set of values is randomly chosen from the second range.

The parameters used in both experiments were as follows. The set S consisted of n=1,000 randomly generated services, where each ser-

vice was described in a behavior sample of 8 instances[6]. Then, m=10 groups of equal size were generated, with each group consisting of 100 services. The $n \times$ n similarity matrix was then computed, and a total of 278 queries were attempted (this number was chosen to assure statistical significance that is higher than 0.95). The precision of each search was calculated with neighborhood sizes of k = 1, 3, 5, and 10. For the embedding, we used 10 randomly chosen pivot services (this choice is justified later). The $n \times n$ matrix was constructed as described at the end of Section 5, and the same 278 queries were attempted and scored. To assure statistical significance, this entire experiment was repeated 100 times.

Quality

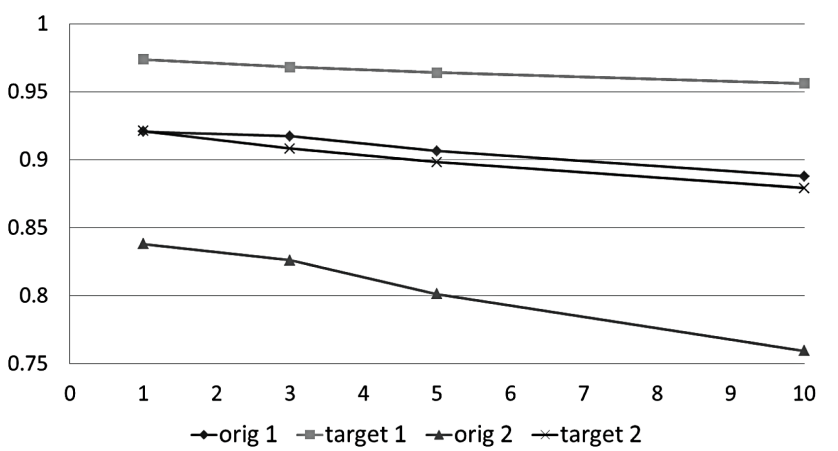
Figure 1 plots the results of finding k-nearest neighbors in both the original space of services and in the target vector space. The horizontal axis is the size of k, and the vertical axis is the overall precision. Two plots reflect first experiment and two plots reflect the second experiment.

Several interesting observations are due. First, quite surprisingly, in both experiments, the performance in the target space was significantly better than the performance in the original space. Whereas we were hoping that the embedding will improve performance (performance is discussed next) while not harming precision too much — in effect, precision improved by 5–6 percentage points.

Second, across all levels of k this precision (in the target space) is kept at the impressive range of 88%–97% — a definite validation of our methods, given that a random assignment of services to groups, since there are 10 groups, would have only 10% chance of being correct. Third, performance in the second experiment was clearly lower than in the first experiment. This can be attributed to the fact the second signature involves more string comparisons than the first signature. The services discovered and presented to the programmer bear similarity to search engine results. Good information retrieval algorithms give higher precision in earlier pages. The fourth observation is that the decline in precision with the increase in k is of a similar nature. Indeed, it is evidence that our methods percolate the better solutions to the top.

Overall, it can be concluded safely that our methods — both the design of the signature metric and the approach to discovering the best substitute services – are highly effective.

Figure 1. Precision at 4 neighborhood sizes



Performance Gain

The improvement in quality after embedding the space of services in a vector space was significant and surprising. Nonetheless, recall that the main reason for this transformation was to improve performance. We expected performance to improve because (1) fewer similarities would be calculated, and (2) the calculation of each similarity would be simpler. For validation, we measured the time (minutes of CPU time) that was required for a nearest neighbor search in the original space and in the target space. This time included 10 runs, each requiring completing the $n \times n$ similarity matrix, and, for each of 278 queries, finding the 10 most similar services. We then formed this ratio:

$$Performance\ Gain = \frac{Time\ in\ original\ space}{Time\ in\ target\ space}$$

The first experiment yielded a ratio of 83.17/1.43 = 58.16 (98.3% reduction), and the second experiment yielded a ratio of 287.00/3.29 = 87.23 (98.8% reduction). Without a doubt, such gains in performance, coupled with the previously discussed improvement in precision, testifies to the advantage of using metric space embedding.

Number of Pivots

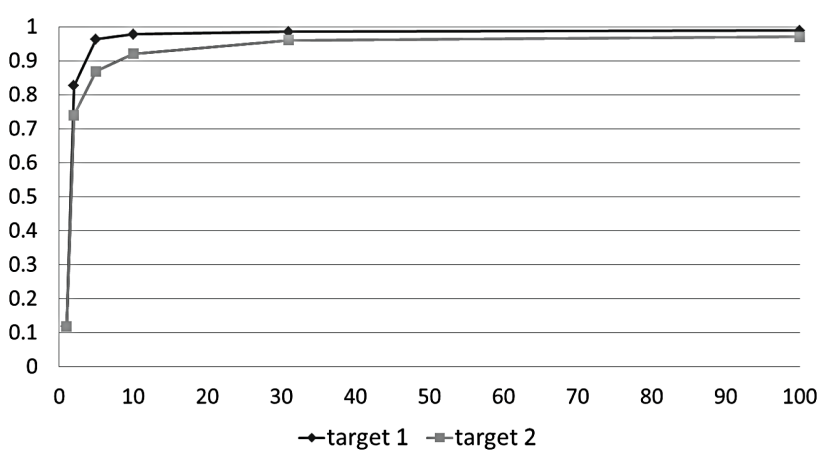
As observed at the end of Section 5, the gain in performance achieved by the embedding method is in reducing n^2 to $n \cdot m$. It is therefore beneficial to keep the number of reference points m low. We tested 8 different numbers of pivots: 1, 5, 10, 30, 100, 200, 500 and 1,000 (recall that the total number of services is 1,000). Figure 2 plots the precision in each experiment for all but the largest 3 numbers (which were essentially the same as for 100). Precision was measured for the case $k = 1$ (i.e., search for the most similar service).

As can be observed, in both experiments, precision exceeds 0.92 when only 10 pivots are used, and it exceeds 0.96 when 30 pivots are used (it is almost flat thereafter). In other words, excellent results were achieved with an embedding that uses just 1–3% of the services as its reference points. This accounts for the large performance gains observed earlier. Again, results for the first experiment were slightly better.

Exceptions

Finally, recall that our model allows for the possibility of exceptions in service behavior. Therefore, it is important to validate the robustness of our methods in the presence of excep-

Figure 2. Precision at different number of pivots



tions. Figure 3 plots, for each experiment, the search performance in both the original space of services and the target vector space when exceptions are introduced into service behavior.

The experiment was done with a neighborhood of $k = 1$ with exception percentages of 1%, 5%, 10% and 20%. For comparison, the plots also include precision when there are no exceptions (0%). As before, performance in the target vector space was consistently higher, and performance in the first experiment was noticeably better. Observe that at exception rates under 10% precision is kept above 0.79. As one is unlikely to deploy services with exception rates higher that 10%, these results are quite encouraging.

EXTENSIONS

Similarity of Services with Different Outputs

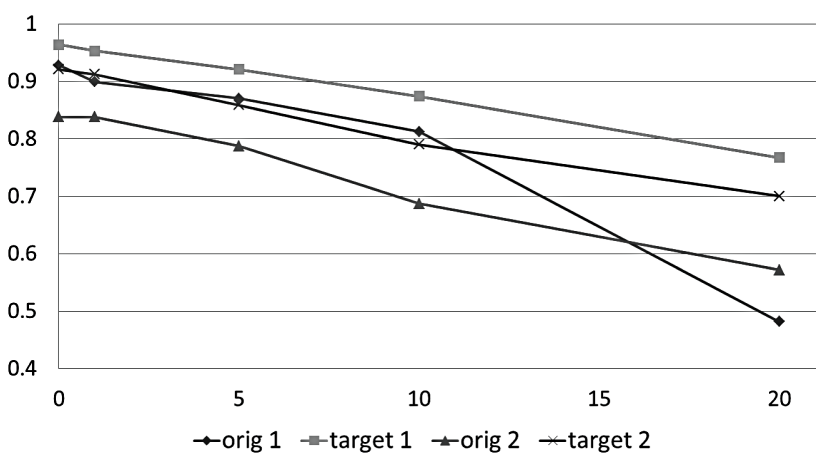
The theory and experiments presented so far made a simplifying assumption, which we shall now attempt to relax. Recall that the signature of a service s is the pair (α, β) of its input and output domain expressions. Our discussion of service similarity was limited to services with identical signatures; that is, services with identically-structured inputs and outputs. This assumption allowed us to compare two services s_1 and s_2 that have the same signature (α, β), by considering triples: an element of domain α, the element of domain β assigned to it by s_1, and the element of domain β assigned to it by s_2. The distance between the latter two is calculated with the domain metric, and then used in the calculation of the similarity between s_1 and s_2.

This assumption implies that we cannot assess the similarity of two weather services when one provides the temperature at a given location, whereas the other provides both the temperature and the barometric pressure. Possibly, simply ignoring the second output could yield a service that is more satisfactory than the stand-by services that provide temperature only.

While substituting a given service with a service that has a different input domain could be useful at times (e.g., in some situations a service that expects a Zip code could be substituted with a service that expects the name of a municipality), it would be rather difficult to calculate the similarity of such services, as there would be no common points for comparison. We therefore continue to assume that services have the same input domain.

Assume now services s_1 and s_2 with signatures (α, β_1) and (α, β_2). Our approach is to extend the definition of distances among elements of the same domain (as given in Section 4) to ele-

Figure 3. Precision at 4 percentages of service exception



ments of different domains. Once this distance has been defined, we shall continue as before: Define similarity of elements by reciprocity, and similarity of services as the average similarity of outputs. The definition of domain expressions in section 3 allowed arbitrarily complex expressions to be created from basic domains using aggregation and sequencing.

Measuring the distance among elements of arbitrarily different domains would be rather complex and of limited practicality. This is because distances are likely to be high and the plausibility that one service could substitute for another is likely to be very low. We therefore limit our discussion to β_1 and β_2 of three structures:

1. A sequence of basic domains: $(B_1, \ldots, B_n)$. Each element of the resulting domain is a sequence of scalars; for example: temperature, humidity and barometric pressure;
2. An aggregate of a sequence of basic domains: $\{(B_1, \ldots, B_n)\}$. Each element of the resulting domain is a table; for example, a set of triples each describing a gas station with longitude, latitude, and the oil company;
3. A combination of both; that is, a sequence of basic domains and an aggregate of a sequence of basic domains (in either order): $((B_1, \ldots, B_n), \{(B_{n+1}, \ldots, B_m)\})$. Each element of the resulting domain is a sequence and a table; for example, a portfolio of stocks (each described with stock symbol, price and quantity), with a date and the tax-id of its owner.

For brevity, we refer to these structures as sequence, table, and sequence+table. Measuring distances among elements of these structures involves six cases: (1) sequence vs. sequence, (2) table vs. table, (3) sequence+table vs. sequence+table, (4) sequence vs. table, (5) table vs. sequence+table, and (6) sequence vs. sequence+table:

Case 1: Assume β_1 and β_2 are both sequences of basic domains. Denote $(B_1, \ldots, B_n)$ the domains that are in both β_1 and β_2, $(C_1, \ldots, C_p)$ the domains that are in β_1 but not in β_2, and $(D_1, \ldots, D_q)$ the domains that are in β_2 but not β_1. That is:

$$\beta_1 = (B_1, \ldots, B_n, C_1, \ldots, C_p)$$
$$\beta_2 = (B_1, \ldots, B_n, D_1, \ldots, D_q)$$

Now assume $x_1 \in \beta_1$ and $x_2 \in \beta_2$:

$$x_1 = (b_1, \ldots, b_n, c_1, \ldots, c_p)$$
$$x_2 = (b'_1, \ldots, b'_n, d_1, \ldots, d_q)$$

To measure the distance from x_1 to x_2, we remove the subsequence $(d_1, \ldots, d_q)$ in x_2 and append a sequence of p exception values. To measure the distance from x_2 to x_1, we remove the subsequence $(c_1, \ldots, c_p)$ in x_1 and append a sequence of q exception values. If either $(d_1, \ldots, d_q)$ or $(c_1, \ldots, c_p)$ are empty (that is, one sequence is contained in the other), we only append the necessary sequence of exceptions. In the special case when one of the sequences is *empty*:

$$\beta_1 = (C_1, \ldots, C_p)$$
$$\beta_2 = ()$$

we proceed according to the same procedure appending a sequence of p exceptions to β_2. In each case, the measurement of distances in the presence of exceptions is done as described in Section 4:

Case 2: To measure the distance between two different tables, we apply the usual procedure described in Section 4. This involves measuring the distance between every row of one table and every row of

the other table, and then calculating the Hausdorff metric. The rows are of differently structured tables, but such distances were discussed in Case 1. A special case that needs to be considered is when one of the tables is empty. For example, $\beta_1 = \{(B_1, \ldots, B_n)\}$, and β_2 is empty. In this case we consider β_2 to be a table with a single row of n exceptions, and proceed with the calculation of the Hausdorff metric;

Case 3: To measure the distance between table+sequence and table+sequence, we, again, follow the usual procedure. This requires totaling the distance between the sequences (as in Case 1) and the distance between the tables (as in Case 2);

Case 4: To measure the distance between a sequence and a table, we consider the sequence as a table with one row, and proceed as in Case 2;

Case 5: To measure the distance between a table and a sequence+table, we convert this case to Case 3 by adding an empty sequence to the former. The distance is then the sum of the distance of the sequences and the distance of the tables. Recall that the distance between an empty sequence and a sequence was discussed in Case 1;

Case 6: To measure the distance between a sequence and a sequence+table, we, again, convert this case to Case 3, this time by adding an empty table to the former. The distance between an empty table and a table was discussed in Case 2.

As a simple example, assume that the common input domain is Zip code and the output domains are both sequences of basic domains: β_1 = (Temperature, Humidity) and β_2 = (Temperature, Pressure). To measure the directed similarity $sim(s_1, s_2)$, we modify the behavior table of s_2 to remove Pressure and insert a column Humidity with exception in every row, and proceed with the standard calculation of distances in the presence of exceptions. Similarly, to measure the directed similarity $sim(s_2, s_1)$ we modify the behavior table of s_1 to remove the column Humidity and insert a column Temperature with exception in every row.

Domain Alignment

Consider a weather service that outputs three different temperatures: current, minimal, and maximal. An alternative service might provide the same three temperatures, but in a different order: minimal, current, and maximal. Since all 6 outputs are of the same domain, and lacking any further knowledge, our methods would rely on the given order and would pair the outputs incorrectly, with negative impact on the resulting similarity. A similar issue arises with input domains.

A "brute force" approach to this problem is to calculate service similarities under all possible alignments and adopt the most favorable. While this sounds combinatorially high, the numbers involved are relatively low. As an example, consider services s_1 and s_2 both with signature:

$$((A, A, B, B)\{(C, C, D, D)\}, (X, X, Y, Y))$$

that is, both services receive a sequence of four values and a table of four columns and return a sequence of four values. Since there are 6 pairs of identical domains, altogether there are 26 = 64 different ways in which their domains could be aligned.

To reduce the complexity we may use heuristic approaches that would find good substitutions, though not necessarily the best. For example, a simple heuristic is annotate domains with their distributions (as extracted from their behavior table). In case of numeric domains, this could be simply the average.

In the example, assume that A is a numeric domain, and assume that in s_1's behavior table the average value of A is 20 in its first appearance and 60 in its second, whereas in s_2's behavior table its average is 48 and 10, respectively. This suggests calculating the similarity between the

services with the *A* columns switched in one of the services; the other similarity will probably be lower.

CONCLUSION

The increased reliance on publicly available services for creating ad hoc, low-cost applications brought about an urgent need for locating service substitutes. As services are usually discovered in large repositories, this problem is naturally stated in terms of a *k*-nearest neighbor problem, which, in turn, requires a formal method for measuring service similarity. In our opinion, it is more productive to gauge similarity on the basis of the intrinsic behavior of services than on meta-information that is associated with services.

In this paper we defined a formal, yet pragmatic method of measuring similarity, and we demonstrated that services discovered on the basis of this measure are indeed, with very high level of precision, similar to the subject of the search (even in the presence of service exceptions). Additionally, we showed how the discovery of *k*-nearest neighbors can be approximated in a vector space created by Lipschitz embedding, with enormous gains in performance.

The results have been encouraging and suggest that expanding the scope of this work may offer additional rewards. We suggest here several such opportunities.

The use of Lipschitz embedding with a rather small number of pivots brought about a large gain in performance. One research direction is to examine further methods for improving the performance of searches; for example, using *vantage-point* trees to locate the most similar services.

Our experimentation used services that were synthesized. While this allowed us to scale up searches to a very large number of services, it would be desirable to apply our solutions in an environment of actual services. An environment that we plan to explore is biomedical services (Zhang and Madduri 2011).

The model and methods that we developed can be applied to other problems in the area of service-oriented programming. For example, our methods for discovery and ranking can be adapted to provide a tool that recommends services based on examples. As another example, the similarity measures could be used to interconnect services in a semantic network, allowing users to explore large networks of services.

REFERENCES

Blake, M. B., & Nowlan, M. F. (2007). A web service recommender system using enhanced syntactical matching. In *Proceedings of ICWS 2007, IEEE International Conference on Web Services* (pp. 575–582).

Bruch, M., Monperrus, M., & Mezini, M. (2009). Learning from examples to improve code completion systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (pp. 213–222). ACM.

Church, J., & Motro, A. (2013). Efficient service substitutions with behavior-based similarity metrics. In *Proceedings of ICWS 2013, IEEE 20th International Conference on Web Services*.

Church, J., & Motro, A. (2011). Learning service behavior with progressive testing. In *Proceedings of SOCA 11, IEEE International Conference on Service-Oriented Computing and Applications* (pp. 1–8).

Dong, X., Halevy, A., Madhavan, J., Nemes, E., & Zhang, J. (2004). Similarity search for web services. In *Proceedings of the 30th International Conference on Very Large Data Bases* (pp. 372–383). VLDB Endowment.

Grechanik, M., Fu, C., Xie, Q., McMillan, C., Poshyvanyk, D., & Cumby, C. (2010). A search engine for finding highly relevant applications. In *Proceedings of ICSE 2010, ACM/IEEE 32nd International Conference on Software Engineering* (vol. 1, pp. 475–484).

Grigori, D., Corrales, J. C., & Bouzeghoub, M. (2006). Behavioral match- making for service retrieval. In *Proceedings of ICWS 2006, IEEE International Conference on Web Services* (pp. 145–152).

Gu¨nay, A., & Yolum, P. (2007). Structural and semantic similarity metrics for web service matchmaking. In E-Commerce and Web Technologies, LNCS 4655, (pp. 129–138). Springer..

Gusfield, D. (1997). *Algorithms on strings, trees and sequences: Computer science and computational biology*. Cambridge University Press. doi:10.1017/CBO9780511574931.

Haran, M., Karr, A., Orso, A., Porter, A., & Sanil, A. (2005). Applying classification techniques to remotely-collected program execution data. *ACM SIGSOFT Software Engineering Notes*, *30*(5), 146–155. doi:10.1145/1095430.1081732.

Hassan, A. E. (2008). The road ahead for mining software repositories. In FoSM 2008, Frontiers of Software Maintenance (pp. 48–57)..

Hjaltason, G. R., & Samet, H. (2003). Properties of embedding methods for similarity searching in metric spaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *25*(5), 530–549. doi:10.1109/TPAMI.2003.1195989.

Junghans, M., Agarwal, S., & Studer, R. (2012). Behavior classes for specification and search of complex services and processes. In *Proceedings of ICWS 2012, IEEE International Conference on Web Services* (pp. 343–350).

Kang, S., Kim, D., Lee, Y., Hyun, S. J., Lee, D., & Lee, B. (2007). A semantic service discovery network for large-scale ubiquitous computing environments. *ETRI Journal*, *29*(5), 545–558. doi:10.4218/etrij.07.0106.0281.

Liu, F., Shi, Y., Yu, J., Wang, T., & Wu, J. (2010). Measuring similarity of web services based on WSDL. In *Proceedings of ICWS 10, International Conference on Web Services* (pp. 155–162).

McMillan, C., Grechanik, M., Poshyvanyk, D., Xie, Q., & Fu, C. (2011). Portfolio: Finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering* (pp. 111–120). ACM.

O'Searcoid, M. (2006). *Metric spaces*. Springer..

Samet, H. (2006). *Foundations of multidimensional and metric data structures*. Morgan Kaufmann..

Shen, Z., & Su, J. (2005). Web service discovery based on behavior signatures. In *Proceedings of the 2005 IEEE International Conference on In Services Computing*, (vol. 1, pp. 279–286). IEEE.

Shepard, R. N. (1987). Toward a universal law of generalization for psychological science. *Science*, *237*(4820), 1317–1323. doi:10.1126/science.3629243 PMID:3629243.

Sillito, J., Murphy, G. C., & De Volder, K. (2008). Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, *34*(4), 434–451. doi:10.1109/TSE.2008.26.

Sushil, K., Bajracharya, J. O., & Lopes, C. V. (2010). Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the 18th ACM SIGSOFT international Symposium on Foundations of Software Engineering* (pp. 157–166). ACM.

Tang, L., & Crovella, M. (2003). Virtual landmarks for the internet. In *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement* (pp. 143–152). ACM.

Tversky, A. (1977). Features of similarity. *Psychological Review*, *84*(4), 327. doi:10.1037/0033-295X.84.4.327.

Yu, Q., & Rege, M. (2009). A relational approach for efficient service selection. In *Proceedings of ICWS 2009, IEEE International Conference on Web Services* (pp. 719–726). IEEE.

Zezula, P., Amato, G., Dohnal, V., & Batko, M. (2006). *Similarity search*. Springer..

Zhang, J., Madduri, R. K., Tan, W., Deichl, K., Alexander, J., & Foster, I. T. (2011). Toward semantics empowered biomedical web services? In *Proceedings of ICWS 2011, IEEE International Conference on Web Services* (pp. 371–378). IEEE.

Zhao, H., & Doshi, P. (2009). Towards automated RESTful web services composition. In *Proceedings of ICWS 2009, IEEE International Conference on Web Services* (pp. 189–196). IEEE.

ENDNOTES

1 This distinction between signature and behavior is similar to the distinction between intension and extension, common in logic and in databases.

2 For now we ignore the fact that this type of service is better stored as a formula, such as C = (F-32) * 5/9.

3 Note that metrics measure distance, and eventually we shall convert distance to similarity.

4 This metric is termed the product metric, as the new space is a product of the component spaces.

5 The final step in that process, taking the maximum of the two directed distances, assures the required symmetry.

6 As mentioned in Section 5, tables of this magnitude have been shown to be effective samples.

Joshua Church is a Ph.D. Candidate in Information Technology with concentration in Software Engineering at the Volgenau School of Engineering at George Mason University. His research interests include applying information retrieval techniques to software engineering problems.

Amihai Motro is a Professor of Computer Science at the Volgenau School of Engineering at George Mason University. He received a B.Sc. degree in mathematics from Tel Aviv University, a M.Sc. in computer science from the Hebrew University at Jerusalem, and a Ph.D. in computer and information science from the University of Pennsylvania. He has published over 100 research papers in leading journals, conferences and other venues. His current research interests are in database systems, information retrieval systems, virtual enterprises, and service-oriented software architectures.