2011 IEEE International Conference on Services Computing

Discovering Service Similarity by Testing

Joshua Church Amihai Motro
Computer Science Department
George Mason University
Fairfax, VA 22030-4444, USA
Email: jchurch2@gmu.edu

***Abstract*—We describe a comprehensive methodology for discovering service similarity (substitutability) by testing. Our solution does not rely on the service descriptions provided by their authors, it does not assume the existence of ontologies, semantic tagging, or other representations, and it avoids common information retrieval techniques. The only information our method expects is that every service in the repository will be annotated with its inputs and outputs, and that each input and output will be associated with a domain from a published list.**

I. Introduction

In recent years, the *Service Oriented Architecture* (SOA) has gained considerable popularity, notably for achieving architectural flexibility at low cost [1]. A shortcoming of this architecture is its *instability*. Instability may be due to a variety of reasons, including unreliable network connectivity, flawed understanding of the exact semantics of a published service, the unexpected evolution in service behavior, or the discontinuation of a service. These can be summarized succinctly in this manner: What worked yesterday might not work today. In addition, one might want to consider situations in which new services become available that outperform services currently being used. In other words, what works today, could be made to work better tomorrow.

To address these shortcomings, methods are needed that *substitute inoperable or sub-par services with compatible services*. Assuming that the available services are registered at a service repository, the problem is to locate a repository service that would be a proper substitute to (or even an improvement upon) a service that is currently being deployed.

The challenge, of course, is to find optimal substitutions *automatically*; that is, without programmer's involvement. The difficulty lies in the fact that descriptions of services are inadequate for this purpose. A service might be described in natural language text, with semantic tags, or with the specification of its interface; i.e., the numbers of its inputs and outputs and their data types. Natural language descriptions can only be understood by humans, and tags or interface descriptions are not sufficiently accurate to guarantee substitutability. For example, there could be several stock quote services that, given a stock symbol, return a price; but the price returned could be the most recent closing price, the highest price in the last 90 days, or the 52-week average price.

The main issue in meeting this challenge is how to discern the *behavior* of services, where comprehension of behavior is defined as the ability to *classify* services reliably using *clusters*. We propose to discover and comprehend the behavior of services and classify them in clusters by means of *testing*.

When the services in the repository have been clustered, any service may be substituted by any other service in its cluster. One could also choose among the cluster members the service that optimizes some utility function (such as the lowest in cost, or the highest in reliability or some combination thereof).

Any new solution should be judged based on its *quality* and its *cost*; that is, to be acceptable as viable, it should perform with both accuracy and efficiency. Our methodology attempts to reconcile these two conflicting factors.

We highlight here some of the challenges that our methodology addresses. First, methods must be developed for generating unbiased tests automatically and for handling different service exceptions (e.g., no response, error message) gracefully. Second, tests must be of the smallest size that assures clustering accuracy. This is because testing entails costly network traffic but also because excessive testing might cause servers to block future access. Third, the tests must discover how to properly match services that have multiple inputs and outputs of the same type. Fourth, clustering requires the adoption of an appropriate measure for evaluating the similarity of services. The measure must handle the structure of the output (a single value, a set of values, a sequence of values of different attributes, or even a table), and it must gauge both *output* similarity (when both services produce an output) and *coverage* similarity (when one service produces an output and the other doesn't). Also, the clustering method must be chosen optimally for the particular output produced by the services being clustered (e.g., numeric or alphabetic). Finally, tests, results, similarities, clusters and other statistical information must be cached to maintain performance, when services are added, removed or modified. Altogether, we believe that this work offers a comprehensive solution to the problem of service substitutability.

II. Methodology

We assume a service repository with a potentially large number of registered services. The repository has a list of available domains. When a service is registered, the submitter must associate a domain with each of the inputs and outputs of the service. This provides for automatic pre-classification of the services into *partitions* based on their input and output domains: The services in each partition have the same number of inputs, the same number of outputs, and their inputs and

978-0-7695-4462-5/11 $26.00 © 2011 IEEE
DOI 10.1109/SCC.2011.91

outputs are associated with the same domains. Obviously, the commonalities among services in the same partition are more syntactical than semantical. Our goal is to further divide each partition into clusters of semantically equivalent services.

Web services may be conveniently viewed as *mappings* from input values to output values. Both the input and the output can be classified into four categories based on their structure: (1) a single value (a *cell*); (2) an *array* of values (in conformance with relational database terminology it will be referred to as a *row*); (3) a *set* of values (a *column*); and (4) a set of rows (a *table*). When applied to both input and output, these classifications yield 16 types of mappings. For example, a service that receives a Zip code and returns the current temperature and humidity in this location is "cell-to-row"; and a service that receives the title of a movie and returns pairs of theaters and show times is "cell-to-table".

1. Eliciting Service Behavior Tables. Consider a repository partition with n services. Our first step in clustering this partition is to obtain a common set of input values $d_1, d_2, \ldots d_p$ for testing its services. The test points d_i are determined by *sampling* the declared domain *randomly*. Each service is then invoked with these test points, returning output values $r_1, r_2, \ldots, r_p$. Note that d_i and r_j could be single values, rows, columns or even tables. The result of each test is a table that denotes the *behavior* of the service:

d_1	r_1
d_2	r_2
$\vdots$	
d_p	r_p

This phase ends when all the partition services have been tested, resulting in n behavior tables.

2. Comparing the Services. To cluster a set of services we need a measure that would calculate the similarity between two behavior tables. Consider two services S_1 and S_2. The i'th row of each table is the output of the service for the test point d_i. Let the second column of that row in each of the tables be r_i^1 and r_i^2, respectively. We define the similarity of the tables in two steps. First, we define a similarity measure $\phi(r_i^1, r_i^2)$ between two corresponding results for the same test point d_i; then, we define the overall similarity $\Phi(S_1, S_2)$ of the service behaviors as the *average* similarity of the corresponding results for each of the test points: $\Phi(S_1, S_2) = 1/p\, \Sigma_{i=1}^{p} \phi(r_i^1, r_i^2)$. The definition of ϕ is must cope with three difficulties. First, r_i could be single values, rows, columns or tables; second, the scalar components of each r_i could be numbers, dates, strings, and so on; finally, in addition to proper values, r_i could also be error values and missing values.

We now compute the similarity between every two services in the given partition. The resulting $n \cdot (n-1)/2$ similarity values are stored in a symmetric $n \times n$ matrix.

3. Clustering the Services. Clustering Web services can be viewed as discerning their behavior, to the extent that we can assert that one service behaves like another. The service-to-service similarity matrix obtained in the previous phase is the basis for forming service clusters. In general, good clustering algorithms create clusterings with low cohesion (inter-cluster distance), and high separation (intra-cluster distance) [2]. Since our methodology must deal with different types of data, we do not adopt a single clustering algorithm, but determine for each partition the clustering method that works best.

4. Determining Minimal Test Cardinality. At this point, we have clustered the services in the partition according to their behavior. Yet, we have no assurance that our clustering is accurate. Obviously, with larger tests better clustering can be expected, but excessive testing incurs both costs and risks, so we should avoid tests that are unnecessarily large. In other words, for each partition we must determine an "ideal" test size: the *smallest test* that will *exceed a predetermined threshold of clustering accuracy*.

Our approach is to *iterate* the process of testing and clustering the partition services with increased test sizes, until the clustering *converges*; that is, additional increases in test size do not yield significant changes in the clustering. This technique is often referred to as *progressive sampling* [3].

To illustrate, suppose we attempt test sizes $p_1, p_2, p_3, \ldots$, and beginning with p_k the derived clusterings are identical. We can then determine that there is no need to conduct tests that are larger than p_k: The clustering *converges* at p_k. In practice, we do not seek a point beyond which the clusterings are *identical*, but merely a point beyond which their differences are sufficiently small, and we measure differences between successive clusterings with the well-known *Rand index* [4].

Experience shows that the point of convergence is often detected by the first Rand index which is sufficiently high, and we adopt a suitable index value α. For statistical robustness, this entire procedure is repeated n times. At each sample size, we calculate the number of times that the clustering converged successfully (i.e., the number of times in the n experiments that the Rand index exceeded α). We adopt a threshold value β for the minimally acceptable rate of successful convergence. Finally, when testing ends, we choose the *lowest sample size* in which the successful convergence rate exceeded β. The results are stored in the partition for future use.

III. Conclusion

We outlined a comprehensive methodology for discovering service similarity (substitutability) by testing. The essence of this work is to (1) elicit the behavior of Web services by judicious *testing*; then (2) use the results of testing to measure service *similarity* and derive service *clusters*; with the final objective of (3) using the clusters to facilitate *substitutability*.

References

[1] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, 2005.

[2] P. Berkhin, "A survey of clustering data mining techniques," Accrue Software, San Jose, CA, Tech. Rep., 2002.

[3] F. Provost, D. Jensen, and T. Oates, "Efficient progressive sampling," in *Proceedings of Fifth ACM SIGKDD*, 1999, pp. 23–32.

[4] W. M. Rand, "Objective criteria for the evaluation of clustering methods," *J. of the American Statistical Society*, vol. 66, no. 336, pp. 846–850, 1971.