Learning Service Behavior with Progressive Testing

Joshua Church Amihai Motro
Computer Science Department
Volgenau School of Engineering
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 solutions do not rely on the service descriptions provided by their authors and they avoid common information retrieval techniques. Our work addresses a variety of challenges raised throughout the process. These include: (1) the generation of unbiased test samples based on individual domains and their statistical properties; (2) the use of progressive sampling and Rand index convergence to minimize sample size; (3) the classification of services by their input and output structures (single values, sets of values, sequences of values, and tables), and the development of corresponding similarity measures; (4) the optimal alignment of services that have multiple inputs and outputs of the same type; (5) the management of two types of service exceptions (*null* values); (6) the selection of clustering methods that are most appropriate to the sets of services being clustered; and (7) the caching of tests, results, similarities, clusters and other statistical information to enable cluster evolution. Initial testing with a prototype implementation validated our methodology, yielding high accuracy at surprisingly small test sizes.**

I. Introduction

Web services are distributed, platform-independent software components, often limited in their functionalities, but easily available to other applications through standard protocols. Service orientation is an approach to software design that accomplishes more complex functionalities by integrating such services. 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, 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. The problem with service descriptions is that they do not adhere to a single standard. As a consequence, service descriptions are often inaccurate, ambiguous, and at times even purposely misleading. Generally, 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 and 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 challenge to finding optimal substitutions automatically is how to discern the *behavior* of services, where comprehension of behavior is defined as the ability to *classify* services reliably by *clustering*. In this paper we propose to discover and learn the behavior of services by means of *testing*.

When the services in the repository have been clustered in this manner, 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). Software that can automatically adjust to environmental changes are often referred to as *self-adaptive*.

The main advantage of testing-based discovery of behavior is that it avoids relying on service descriptions. Thus, our methodology does not assume the existence of ontologies, semantic tagging, or other representations. Indeed, the only information our methodology 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.

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 dif-

ferent 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 does not). 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.

Our methodology is described in detail in Section III. It was implemented in prototype form and was used in experiments of appreciable scale. The implementation and experimentation are described and discussed in Section IV. The results indicate that our approach is viable, with cluster accuracy exceeding 75% using as few as 16 sample tests points. Additionally, statistical significance tests validate that the resulting service clusters are representative of service behavior and do not occur by chance. The paper concludes in Section V with a brief summary and description of ongoing research. We begin with a brief review of related work.

II. Related Work

The essence of the work described here 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*. We discuss here briefly prior research in these subjects.

A. Testing Services

Testing is a familiar means for assessing the behavior of software; usually to ascertain that the software performs according to its specification [2]. In the context of SOA, testing has been similarly used to gain confidence that the service integrating application performs as expected [3]. In this vein, [4] suggests verifying and validating Web services with techniques of data perturbation. In contradistinction, our objective here is to generate a suite of tests that *elicits the behavior* of services.

B. Clustering Services

Several works have explored the subject of service clustering [5], [6], [7], [8]. Invariably, the requisite information for clustering is derived from the files that describe the Web services. Typically, the keywords extracted from WSDL files are processed with information retrieval techniques (e.g., singular vector decomposition) to create vector representations of services. The similarity of these representations is then measured with standard IR measures (e.g., cosine). In contradistinction, our methods ignore WSDL descriptions entirely, and the similarity measures that we introduce have been designed to estimate the similarity of *behaviors*, rather then the similarity of *descriptions*.

The goal of the four works discussed above is to locate services that satisfy a need; and this need is assumed to be specified with IR-style *keywords*. Somewhat different, [9] attempts to locate services based on *non-functional attributes*, such as security or reliability. Our purpose here is different from these all: We aim to locate substitutions. Hence, our search goal is specified with an *example* (not keywords): Find services that performs like a particular service.

C. Substituting Services

Self-adaptive systems emphasize the selection of alternative services in situations when the runtime environment affects the continued operation of software [10]. In particular, self-healing approaches initiate service substitution when stated Quality of Service (QoS) requirements are no longer met. The issue of *when* to initiate substitutions is addressed in [11]; in contradistinction, our work addresses the issue of *which* services should be selected as substitutions. Similarly, the works of [12], [13] adapt service compositions at run-time by selecting the highest quality substitute from a service registry. Optimal quality is measured by [12] as the greatest similarity between the QoS of a candidate substitution and the QoS expected of a given service composition. In the same vein, selecting the optimal substitute is the subject of [13]. Accordingly, domain experts define utility functions that represent a service composition's QoS goals. The utility functions are given run-time performance data and the system computes the current utility of the application. If the utility falls below a threshold an adaptation is initiated. QoS is a primary factor considered during substitution in [12], [13]; on the other hand, behavior is our primary factor.

Finally, we discuss three works that are most closely related to the work described in this paper. Finding substitutable services is also the subject of [14]. It shares an important principle with our work: Services are described with tables that specify their inputs and outputs, and table-to-table similarities are used to determine substitutability. The approach is elegant, but it ignores many critical issues, including the automatic generation of such tables with unbiased samples, the determination of minimal samples that meet an accuracy threshold, the clustering of services into equivalence sets, and the appropriate handling of null outputs of various kinds.

The subject of [15] is a process for evaluating component replaceability. It, too, adopts the approach that service behavior is observed through its input-output mappings. But, like the previously discussed paper, it has a single focus: the generation of a test suite and the analysis of its results.

The goal of the approach described in [16] is to locate Web services. Users must describe their needs with keywords, which are then used to search through WSDL files for appropriate matches. Tests are then generated automatically, and the results are logged, sorted and delivered to the user. While the approach involves automatic testing, it requires considerable involvement of the programmer in providing semantic search information and in selecting the best service from the results.

III. Methodology

Our methodology involves two principal tasks: *eliciting* service behavior and *clustering* services according to that behavior. Service elicitation begins with the partitioning of the service registry according to the input and output types of the services; the services in each partition are then tested with a common sample of test points, and the responses are saved in *behavior tables*. Clustering begins with the measurement of similarity between every two behavior tables in each partition. The results are the basis for forming clusters of services.

An important objective of the methodology is to use a low number of tests to generate service clusters of high quality. To this end, we repeat these two tasks, each time incrementing the number of sample tests geometrically, until *convergence* is reached. Convergence occurs when the difference between successive clusterings is small. After convergence, the artifacts of our methodology (e.g., test values, behavior tables, similarity matrix, and clusters) are saved to the service registry. Figure 1 illustrates this architecture.

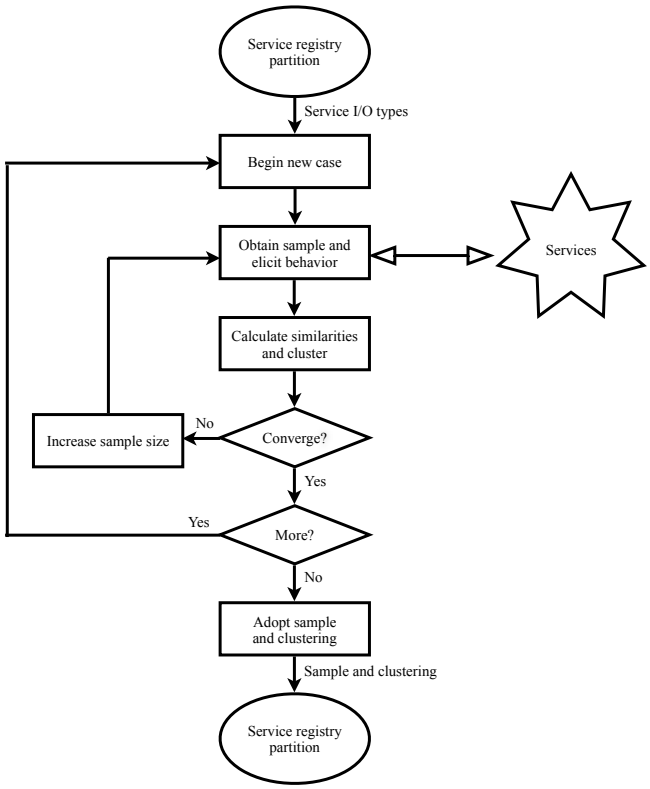

Fig. 1. Methodology for clustering services by testing

We begin by formalizing the environment in which we operate. 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 is requested to associate a domain with each of the inputs and outputs of the service (the list of domains can be expanded if necessary).

Each service is provided with a *wrapper*. The wrapper is used to interface with the service: It processes the user input, formats it to fit the service requirements, invokes the service, and formats the service output in a table (as if it were an answer from a relational database).

A. Partitioning Services by Their I/O Domains

We assume that all the services have been pre-classified into *partitions* based on their input and output domains. That is, the services in each partition have the same number of inputs, the same number of outputs, and their inputs and outputs are associated with the same domains. This classification is automatic: When new services are registered, they are assigned to their partitions. 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"; a service that receives the title of a movie and return pairs of theaters and show times is "cell-to-table"; and a service that receives a combination of stock symbol and date and returns the opening, closing, low and high prices for that date is "row-to-row".

B. 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, as shown in Table I.

1) Managing Different Types of Result: Note that each sample point d_i might result in three types of output: (1) a valid value, (2) an error value (illegal input), or (3) a missing value (no response). These outputs are all encoded in the behavior table. Both error values and missing values are *exceptions*. A missing value reflects a service invocation without a response.

TABLE I
SERVICE BEHAVIOR

d_1	r_1
d_2	r_2
⋮	
d_p	r_p

It resembles a database *null* of the type *not available* [17]. In data mining applications such values are often *imputed* from other values [18]. On the other hand, an error value indicates that the input is outside the expected domain. It resembles a database *null* of the type *not applicable* [17] (and imputation is entirely unjustified). Indeed, that one service responds to an input with a proper value, whereas another issues an error message, is an indication of service dissimilarity.

2) Selecting Unbiased Random Sample of Inputs: As already mentioned, the repository includes a list of domains. For accurate sampling, additional information must be available on each domain and its distribution. The service repository represents domains either as numeric ranges (e.g., temperatures or salaries), or as enumerated sets (e.g., Zip codes or stock symbols). Very large discrete domains (e.g., last names or movie titles) are represented with sufficiently large samples. In addition, information is kept on the *distribution* of the domains, to be used as sampling guidelines for minimizing *selection bias*. Although missing values may be imputed, when the sample test includes a high proportion of such values, the sample is rejected and a new sample is obtained. This is intended to minimize *non-response bias*.

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

C. Comparing the Services

To cluster a set of elements, an element-to-element similarity measure must be adopted. In our case 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 (p is number of test input values):

$$\boxed{\Phi(S_1, S_2) = 1/p\, \Sigma_{i=1}^{p} \phi(r_i^1, r_i^2)} \quad (1)$$

The definition of ϕ 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. As explained, missing values are handled with imputation (unless their proportion is high), but error values are retained as a valid part of the service behavior. Therefore, ϕ should gauge the similarity of outputs when both services respond, but also consider the number of inputs which one service can handle whereas the other cannot.

We omit further discussion of the similarity measure ϕ, but in Section IV we describe the particular measures used in the experimentation. We emphasize that, as is common in information retrieval systems and search engines, the definition of ϕ could be subject to continuous evolution and fine-tuning.

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 $n \times n$ similarity matrix.

D. 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) [19]. Generally, clustering algorithms fall into two main categories: hierarchical and partitional. Each approach has its advantages and disadvantages and there is no single algorithm that performs well in all situations. Factors such as element types (qualitative or quantitative), dimensionality, and cluster structure (overlapping or non-overlapping) guide the selection of the algorithm to choose. For instance, hierarchical algorithms generate nested clusters that form a tree structure, whereas partitional algorithms generate sets of disjoint clusters.

1) Choosing the Clustering Algorithm: Since our methodology must deal with different types of data, we do not adopt a single clustering algorithm, but determine for each partition of services the clustering method that works best. The experiments described in Section IV choose between two clustering algorithms: Hierarchical Agglomerative Clustering (HAC) and k-Means partitional clustering. Both algorithms are extensible; that is, their clustering can be amended when the set of elements changes. This is important for our methodology as we expect service repositories to be dynamic, with new services added and existing services modified or removed.

2) Considering Different Orderings of Inputs and Outputs: Consider now two services, each with two inputs and two outputs. Assume that all four inputs are of the same domain, and all four outputs are of the same domain. Imposing a strict ordering on the inputs and outputs implies that there is a single way to compare these services. However if, we are willing to accept that two services that expect the same two inputs but in reverse order (or two services that produce the same two outputs but in reverse order) are comparable, then each service has four different "versions". In other words, fixing the orders of inputs and outputs in one service as the "correct" ones, requires four similarity comparisons to the other service. Note, however, that permutations of the inputs require additional testing, whereas permutations of the outputs do not.

E. 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*. We note that previous work in this area ignored the issue of choosing test cardinality judiciously.

1) Detecting Convergence: Our approach is to *iterate* the process of testing and clustering the services in a partition 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* [20], [21].

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* [22]. Consider two clusterings C_1 and C_2 of n elements. Intuitively, this index considers all $\binom{n}{2}$ pairs of elements and calculates the proportion of "agreements" among the two clusterings on their placement. An "agreement" is placing the elements of a pair in the same cluster or in two different clusters in *both* C_1 and C_2. The index generates values between 0 (C_1 and C_2 are in complete disagreement) and 1 (C_1 and C_2 are identical).

A sequence of k clusterings will produce $k-1$ Rand index values. Theoretically, a high value could be followed by a low value; i.e., a clustering is very similar to its predecessor, but very different from its successor. In practice, however, this rarely happens, and high values are usually followed by high values; i.e., the plot of Rand index values reaches a "plateau". Hence, it should be sufficient to get a high value (two similar clusterings in succession) and then adopt the sample size that was associated with the first of these clusterings. We adopt a Rand index value α that is considered sufficiently high.

2) Calculating Optimal Sample Size: 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 corresponding sample and clustering are stored in the service repository partition to be used in future occasions (e.g., to classify new services or to reclassify current services).

Our methodology is summarized in Algorithm 1 below.

Algorithm 1 ELICITBEHAVIOR(α, β)

```
Partition P ← {service_1, service_2, ..., service_n}
Domain D ← {d_1, d_2, ..., d_d}
Test Schedule S ← {p_1, p_2, ...}
Sample Size p_t ← S_0
Execution e_l ← 0
while e_l ≤ n do
    Previous Clusters c_{k-1} ← null
    Rand Values c_{k-1} ← null
    repeat
        Inputs I ← RandomlySelect(D, p_t)
        for all Service s_k ∈ P do
            for all Input d_i ∈ I do
                Output r_j ← s_k.invoke(d_i)
                s_k.table.add(d_i, r_j)
            end for
        end for
        Clusters c_k ← DoClustering(P, φ)
        Rand Index r ← 0
        if c_{k-1} ≠ null then
            r ← CalculateRand(c_{k-1}, c_k)
        end if
        p_t ← nextvalue ∈ S
        c_{k-1} ← c_k
    until r > α
    CheckConverge(e_l, r, β)
    e_l ← e_l + 1
end while
```

IV. EXPERIMENTATION

For this experiment we focused on two categories of services: cell-to-cell and cell-to-row.

1) Weather services that receive a Zip code and return a numeric value.
2) Stock quote services that receive a stock symbol and return a pair of numeric values.

Notice that input domains are rather specific: They are assumed to be included in the list of domains of the service repository. On the other hand, we assumed that the output domains are generic, which adds the challenge of output matching.

A. Classifying Weather Services

For the first experiment we considered 11 weather services of 4 different functionalities: Four services return the temperature in degrees Fahrenheit, one returns the temperature in degrees Celsius, three return the relative humidity, and three return the barometric pressure. Our goal is to identify these clusters in the given set of services. The input domain was fully enumerated (all 27,238 Zip codes), and the sampling principle was random (without replacement): Each value in the domain had equal likelihood of being selected.

Our progressive sampling started with a sample size of 2 and doubled it repeatedly to 4, 8, 16, 32 and 64. We

used the following distance function (which was subsequently converted to a similarity measure ϕ by subtracting distances from the maximal distance): The distance between two values is their absolute difference; when both values are exceptions, the distance is 0; and when only one value is an exception, the distance is taken as the average distance. The overall similarity Φ of the two behavior tables is the average of the ϕ values over the entire sample.

Given these similarity measurements, we clustered the services using both HAC and k-Means partitional clustering. We repeated this process 6 times with increasing sample sizes, computing 5 Rand index values for each clustering method (between the clustering obtained at 4 and the clustering obtained at 2; between the clustering obtained at 8 and the clustering obtained at 4, and so on). Our termination condition was set to be a Rand index in excess of $\alpha = 0.7$ (if that threshold is not met, then the clustering of the final test is adopted).

For added statistical significance, each of the experiments was replicated $n = 40$ times, and the minimal proportion of successful convergences was set at $\beta = 0.8$.

TABLE II
WEATHER: RAND CONVERGENCE

Sample	k-Means	HAC
2	0	0
4	0.5	0.23
8	0.73	0.55
16	0.95	0.73
32	0.95	0.9
64	1	0.95

1) Results: Table II summarizes the results of this experiment. The columns k-Means and HAC show, for each clustering method, the percentage of experiments where the Rand index exceeded the 0.7 convergence threshold. As can be observed, for k-Means clustering, to meet the threshold of 80% successful convergence, a sample of size 16 should be used (the rate of successful convergence is then 95%). For HAC, a sample size of 32 would be necessary to meet the threshold of 80% successful convergence (the rate of successful convergence is then 90%). This suggests that for this class of services, k-Means clustering with samples of size 16 should be preferred.

2) Validation: Any clustering algorithm will produce a set of clusters, even in random data. Hence it is important to validate that our service clusters characterize the data accurately. Towards this end, we measured the *accuracy* of the clustering recommended by the methodology (i.e., k-Means clustering with a sample of 16 tests), by comparing it to the true clustering (as defined by experts). Each of the two clusterings was represented in symmetric matrix in which each service is represented by a row and column, and a cell (i, j) is 1 if the services in row i and column j are in the same cluster; otherwise it is 0. High correlation between the two matrices indicates high accuracy [23]. The average accuracy in the 40 trials that were performed was 75%.

The cluster accuracy score thus obtained should be evaluated for its statistical significance. Cluster accuracy is valid only if it is unusual; that is, the clusters correspond to service behavior better than by chance. The significance of the accuracy score is tested using a statistical hypothesis test. The null hypothesis states that the cluster assignments are random (and the accuracy was obtained by chance). The alternative hypothesis states that the cluster assignments are not random (and the accuracy could not have been obtained by chance) [23]. 300 cluster assignments were generated at random. Each was compared to the matrix of the true clustering and the accuracy score was computed. Figure 2 shows the distribution of these random accuracy scores. Next, we compared our accuracy to this distribution to decide if it is unusual (i.e., greater than a critical value). In this case 99% of the accuracy scores were less than 0.5; hence the accuracy score of 0.75 is statistically significant.

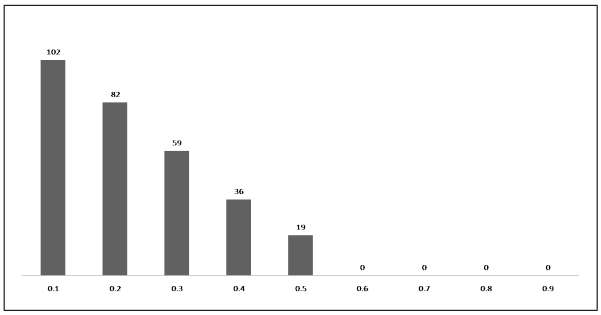

Fig. 2. Weather: accuracy distribution

B. Classifying Stock Quote Services

For the second experiment we considered 12 stock quote services of 4 different functionalities: Two services return the opening and closing prices of the stock, four return the daily change and the market capitalization, four return the current price and the earnings per share, and two return the price per share and the traded volume. Again, our goal is to identify these clusters in the given set of services. The input domain was assumed to be comprehensive, though not necessarily exhaustive (2,809 stock symbols), and the sampling principle was again random (without replacement): Each value in the domain had equal likelihood of being selected.

Our experiment followed the same procedure as the first experiment, with the exception of the similarity measure ϕ, which used Euclidean distance to measure the difference between two results (as each result is now a *pair* of values).

1) Results: Table III summarizes the results of the second experiment. As can be observed, for k-Means clustering, to meet the threshold of 80% successful convergence, a sample of size 16 should be used (the rate of successful convergence is then 83%). For hierarchical agglomerative clustering, a sample size of 32 would be necessary to meet the threshold of 80% successful convergence (the rate of successful convergence is then 80%).

TABLE III
STOCK: RAND CONVERGENCE

Sample	k-Means	HAC
2	0	0
4	0.58	0.28
8	0.75	0.45
16	0.83	0.7
32	0.85	0.8
64	0.98	0.98

2) Validation: The average accuracy of the clustering proposed by this methodology (k-Means clustering with samples of size 16), as measured by the Rand index, was 84%. The significance of this accuracy score was validated with the same hypothesis test described earlier. As evident in Figure 3, this accuracy score is significant.

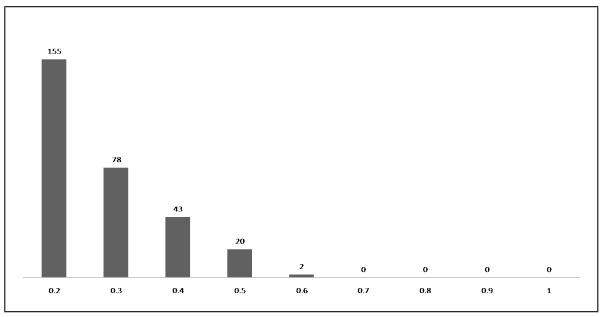

Fig. 3. Stock: accuracy distribution

We note that in both experiments k-Means clustering proved superior. This may be due to the fact that in both cases the output of the services is quantitative. Altogether, the results of these preliminary tests are promising, showing that our methodology is capable of providing accurate clusters (75% and 84%) with a surprisingly low number of tests (in each case, 16 service invocations).

V. CONCLUSION

We presented a comprehensive methodology for discovering service similarity (substitutability) by testing. Our solutions do not rely on the service descriptions provided by their authors and they avoid common information retrieval techniques. Rather than concentrate on a single issue, our work addresses a variety of challenges raised throughout the process. These include: (1) the generation of unbiased test samples based on individual domains and their statistical properties; (2) the use of progressive sampling and Rand index convergence to minimize sample size; (3) the classification of services by their input and output structures (cells, columns, rows and tables), and the development of corresponding similarity measures; (4) the optimal alignment of services that have multiple inputs and outputs of the same type; (5) the distinction between two types of service exceptions (*null* values): "no response" values (which must be considered aberrations, and can be imputed or ignored) and "input error" values (which must be considered part of the behavior, and hence must be handled appropriately by the similarity measures); (6) the selection of clustering methods that are most appropriate to the sets of services being clustered; and (7) the caching of tests, results, similarities, clusters and other statistical information to enable cluster evolution.

We implemented this methodology with a prototype system, and we conducted two tests: one with a set of weather services (cell-to-cell) and another with a set of stock quoting services (cell-to-row). Both tests validated our methodology yielding high accuracy at surprisingly small test sizes.

Our research is in its preliminary stages and there are many additional potentially rewarding directions for research. We mention here four such directions.

A. More Extensive Testing

We plan to perform more comprehensive testing; in particular, conduct tests with *larger sets of services*, with services that use *non-numeric* inputs and outputs, and with services that have *more complex* inputs and outputs (i.e., columns, rows and tables).

B. Further Performance Improvements

For the solution to be viable, its cost must be kept low, and we intend to explore various cost-cutting opportunities. Two such opportunities are sketched here. Our methodology requires that all services are invoked with increasing samples of test points until convergence. When the number of services in a partition becomes large, the total number of invocations could become substantial. This problem could be mitigated in two ways. First, new test samples could be obtained by *incrementing* existing samples with additional test points. When the test size increases by a factor of r, this would cut the total number of service invocations by about $1/r$ (e.g., when the test size doubles, the total number of service invocations would be cut in half). Additionally, the iterative procedure illustrated in Figure 1 could be performed with only a *sample* of the services. Once convergence is reached, the remaining services would be invoked just once for the final clustering. When the test size increases by a factor of r, and the service sample proportion is α, the number of service invocations would be reduced by about $(1-\alpha)/r$ (e.g., with a sequence that doubles and a service sample of 20%, the number of invocations would be reduced by 40%).

C. Discover Services that May be Transformed to Substitutes

In section III we partitioned the entire repository based on the inputs and outputs of the services, and clusters of equivalent services were identified *within* each partition. We note that frequently a service that has been assigned to one partition could easily be *adapted* to a service that is comparable to services in *other* partitions. This could be done by an adaptation of its I/O domains. The simplest adaptation is to ignore ("clip") one of the service outputs. The "new" service then becomes "eligible" for membership in another partition. The subject of discovering equivalent services among services that are not input/output compatible is currently being investigated.

D. Incorporate the Methodology into a Larger System

The Web is highly dynamic, and so are the services made available to service integrators: Services are added, updated or withdrawn, and their quality of service parameters (e.g., reliability, performance, or cost) are continuously changing. What worked yesterday might not work today, and what works today can be made to work better tomorrow. We view the problem addressed in this paper as part of a larger goal. Namely, how to construct a system that (1) facilitates the development of *optimal* compositions of Web services, and (2) monitors their operation continuously, repairing and optimizing them as needed.

REFERENCES

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

[2] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[3] S. R. Tilley, X. Bai, and G. A. Lewis, "Proceedings of soat-2009, first international workshop on service-oriented architecture testing," in *Proceedings of ICSM-2009, IEEE International Conference on Software Maintenance*, 2009, pp. 583–584.

[4] J. Offutt and W. Xu, "Generating test cases for web services using data perturbation," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 5, pp. 1–10, 2004.

[5] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang, "Similarity search for web services," in *Proceedings of VLDB-2004, Thirtieth International Conference on Very Large Data Bases*, 2004, pp. 372–383.

[6] W. Liu and W. W.W̃ong, "Discovering homogenous service communities through web service clustering," *Service-Oriented Computing: Agents, Semantics, and Engineering*, pp. 69–82, 2008.

[7] M. Sellami, W. Gaaloul, and S. Tata, "Functionality-driven clustering of web service registries," in *Proceedings of SCC-2010, IEEE International Conference on Services Computing*, 2010, pp. 631–634.

[8] Q. Yu and M. Rege, "On service community learning: A co-clustering approach," in *Proceedings of ICWS-2010, 8th IEEE International Conference on Web Services*, 2010, pp. 283–290.

[9] G. R. Santhanam, S. Basu, and V. Honavar, "Web service substitution based on preferences over non-functional attributes," in *Proceedings of SCC-2009, IEEE International Conference on Services Computing*, 2009, pp. 210–217.

[10] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic *et al.*, "Software engineering for self-adaptive systems: A research roadmap," *Software Engineering for Self-Adaptive Systems*, pp. 1–26, 2009.

[11] S. S. Pillai and N. C. Narendra, "Optimal replacement policy of services based on markov decision process," in *Proceedings of SCC-2009, IEEE International Conference on Services Computing*, 2009, pp. 176–183.

[12] D. Ardagna and B. Pernici, "Adaptive service composition in flexible processes," *IEEE Transactions on Software Engineering*, pp. 369–384, 2007.

[13] D. Menascé, J. Ewing, H. Gomaa, S. Malek, and J. Sousa, "A framework for utility-based service oriented design in SASSY," in *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering*. ACM, 2010, pp. 27–36.

[14] M. D. Ernst, R. Lencevicius, and J. H. Perkins, "Detection of web service substitutability and composability," in *Proceedings of WS-MaTe-2006, International Workshop on Web Services–Modeling and Testing*, 2006, pp. 123–125.

[15] A. Flores and M. Polo, "Testing-based process for evaluating component replaceability," *Electronic Notes in Theoretical Computer Science*, vol. 236, pp. 101–115, 2009.

[16] Y. Park, W. Jung, B. Lee, and C. Wu, "Automatic discovery of web services based on dynamic black-box testing," in *Proceedings of 33rd Annual IEEE International Computer Software and Applications Conference*, 2009, pp. 107–114.

[17] A. Motro, *Management of Uncertainty in Database Systems*. Addison-Wesley/ACM Press, 1994, pp. 457–476.

[18] D. B. Rubin, *Multiple Imputation for Nonresponse in Surveys*. Wiley, 2004.

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

[20] F. Provost, D. Jensen, and T. Oates, "Efficient progressive sampling," in *Proceedings of Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1999, pp. 23–32.

[21] G. H. John and P. Langley, "Static versus dynamic sampling for data mining," in *Proceedings of Second ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1996, pp. 367–370.

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

[23] P. Tan, M. Steinbach, V. Kumar *et al.*, *Introduction to data mining*. Pearson Addison Wesley Boston, 2006.