Service-oriented computing is about building new cross-organizational applications by combining, composing, consuming, or interconnecting existing services. So, why do most composite Web service-based systems currently rely on pre-established relationships that aren’t created by automated, dynamic discovery and integration? One perceived reason is the inconsistency in service-based interface descriptions and message names. Here, the authors investigate whether human nature — specifically, software developers’ tendencies to name service descriptions in significantly consistent ways — can provide syntactical methods for service discovery.

Web services are at the core of service-oriented architectures (SOAs) and the service-oriented computing (SOC) paradigm. We can define Web services as networked capabilities with openly accessible interfaces that other machines can discover and execute in real time. Web service composition involves connecting a chain of multiple, domain-related services, but, in practice, discovering services is the first step. Several international forums encourage researchers and industry engineers to develop service discovery and composition systems that can autonomously, perhaps intelligently, compose Web services (for example, the Web Services Challenge [WSC], the Semantic Web Services Challenge [SWS], and the Services Computing Contest). Each of these venues creates its own Web service repository (that is, with clean interfaces) as opposed to using services available over the Internet.

In this article, we’re interested in Web services that weren’t created in such controlled, homogeneous environments. We collected as many Web Service Description Language (WSDL) documents (service specifications) we could acquire from various repositories and sources openly available on the Internet. By analyzing these real services at greater depth, we can interpret service developers’ tendencies to use predictable phrases or nomenclature when creating interfaces and service messages. This will help us integrate the observed tendencies with straightforward syntactical approaches, such that we can achieve a quick, just-in-time service management approach for Web service discovery and composition.

Although ambitious, this approach’s success could effectively facilitate service discovery and composition without requiring all the specifications of full semantic techniques (that is, ontology-
Based on semantic description languages such as the Resource Description Framework (RDF) or the Web Ontology Language for Services (OWL-S; www.daml.org/owl-s/). A fully syntactic approach can’t completely replace the need for semantic descriptions but can enhance semantic approaches by recommending services based on their underlying specification data prior to semantic processing—in effect taming the target services repository. Here, we describe applying our approach to decompose data from specified business process routines into keywords that we can match to keywords extracted from candidate services. Consequently, we introduce automated approaches for identifying services that are relevant to a designated, operational setting.

### Processing WSDL Documents

Based on the W3C’s WSDL specification (www.w3.org/2002/ws/desc/), a WSDL document describes Web service attributes on the basis of specialized XML elements such as services, ports, bindings, operations, messages, and parts (Figure 1 shows a metamodel as a Unified Modeling Language (UML) class diagram). The service element names the Web services and includes multiple physical locations, described as ports. Each port can bind to multiple operations (that is, atomic elements of work). Operations are further defined by messages containing parts (inputs and outputs).

When analyzing naming tendencies within Web services, we used three basic steps for capturing the operation names and corresponding messages, as Figure 1 illustrates using a WeatherForecast WSDL document. In step 1, our software gathers all operation names from all port elements in the WSDL file. In step 2, for each operation name captured in step 1, our software extracts all input and output elements and their corresponding message and part elements. On development platforms (such as Microsoft’s .NET) in which hierarchical description is enabled, a third step is necessary. When an element attribute appears in place of a type attribute within the WSDL message element, a keyword such as “parameters” or “body” is substituted for the name attribute. Note the difference between “WSDL message part from type elements” and “Inline WSDL message parts” in Figure 1. For the former, we wouldn’t capture the low-level message information within the message element (as we would with the latter) but rather within WSDL types. As such, in step 3 our software traverses the WSDL document to locate the type elements referenced in each part element extracted in step 2. Similar to related work, we use recursive processing to capture nested or hierarchical relationships within specifications and subsequently drill down to extract the actual message strings.

### What Do Harvested Web Services Look Like?

To capture the true random nature of Web services naming, we manually downloaded 596 real Web services (that is, WSDL documents) from Amazon (www.amazon.com/gp/aws/landing.html), numerous Web services repositories (Woogle, XMLMethods, Salcentral, BindingPoint, and WebServiceList, via www.xmethods.com), and random Internet searches. We decomposed the 596 service descriptions into 31,807 total message parts (parts are the independent parameter strings that comprise a Web service message); 18,815 parts remained once we removed duplicate entries (that is, the same part name occurring multiple times in the same WSDL document). After also removing duplicates that spanned multiple WSDL files, we had 5,180 unique parts across all services in the repository, regardless of input or output. There were more output parts than input parts, suggesting that input information is slightly more homogeneous than output information. As further evidence, note that 20 percent of output parts (after removing duplicates) were unique across services, whereas the inputs were just 13 percent unique. The knowledge that developers are more likely to use the same or similar parts for input messages might be helpful, with regards to search speed, for consumers who index results from UDDI registries. Table 1 lists the repository’s quantitative details.

In evaluating the WSDL files, we determined that header information uniquely identified the Web service development platform used. Microsoft’s .NET services were overwhelmingly the most popular out of those repositories. Figure 2 shows the percentage of each development platform represented.

Similar to categories devised in the Woogle Web service search engine, we divided our repository into nine categories: calendar, business and economy, news and reference, graphics, communication, technology, conversion, enter-
tainment, and military. The military category contains Web services openly available for government defense information systems. Figure 2 also shows the most frequent type (category) of service created with each development kit.

Predicting Similarity Using Human Development Tendencies
Identifying equivalent keywords is vital to correlating relevant Web services to real business operations. Table 1 shows that, of the maximum
possible unique WSDL parts, roughly 25 percent are truly unique within the entire repository. This is promising, considering that different organizations built the services—it suggests that Web service developers, perhaps instinctively, use the same names for common message types. Nevertheless, identifying relevant services by exactly matching WSDL parts isn’t practical. We devised a more flexible approach that uses the WSDL parts’ similarities to determine equivalence, although some loss of accuracy occurs. The major question that we answer is, “What are the properties that make two WSDL parts equivalent, even though they don’t match exactly?” In answering this question, we sorted the repository and then analyzed similar phrases manually. Then, using semiautomated approaches, we determined human trends in naming the Web services messages occurred with high frequency. Based on the determined trends, we used complimentary natural language processing techniques to exploit them and determine their equivalence. Our analysis revealed many tendencies, but we describe only four examples here:

- **Subsumption relationships** (tendency 1). A strong tendency exists for Web service developers to use WSDL parts based on common phrases. When using common phrases, similar messages tend to have strong subsumption relationships. For example, we found equivalent messages where name = lname, name = first_name, and name = user_name.
- **Common subsets** (tendency 2). Similar to subsumption relationships, some Web service parts had common subsets. We found equivalent WSDL parts where first_name = user_name, for example.
- **Abbreviations in naming** (tendency 3). Another strong tendency was shortening common phrases into abbreviations. For instance, building = bldg or country = cntry.
- **Size constraints** (tendency 4). Finally, we determined that strings shorter than three characters and longer than 15 characters were ineffective for matching part names. These tendencies occurred most frequently, which compelled us to determine specific processing techniques that enhanced syntactical matching.

### Exploiting Naming Tendencies to Find Relevant Services
Based on the previously defined tendencies, we developed an algorithm that exploits those tendencies to discover when Web service part names are equivalent. Probably the most straightforward comparison for two part names is to check

<table>
<thead>
<tr>
<th>WS development platform</th>
<th>Representation (%)</th>
<th>Most frequent category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microsoft .NET</td>
<td>67.79</td>
<td>Business and Economy</td>
</tr>
<tr>
<td>Misc./not determined</td>
<td>16.11</td>
<td>Technology</td>
</tr>
<tr>
<td>Openuri.org (NetBeans)</td>
<td>4.87</td>
<td>Military</td>
</tr>
<tr>
<td>Borland</td>
<td>2.85</td>
<td>News and Reference</td>
</tr>
<tr>
<td>WebMethods</td>
<td>2.85</td>
<td>Technology</td>
</tr>
<tr>
<td>Apache SOAP</td>
<td>2.68</td>
<td>News and Reference/Technology</td>
</tr>
<tr>
<td>Sun (SOAPInterop)</td>
<td>2.52</td>
<td>Communication</td>
</tr>
<tr>
<td>ColdFusion</td>
<td>2.35</td>
<td>Entertainment/Technology</td>
</tr>
<tr>
<td>Apache Axis</td>
<td>0.84</td>
<td>News and Reference</td>
</tr>
</tbody>
</table>

| Figure 2. Representation of development environments in the repository. The table and graph show the percentage of Web services by implementation type that were represented in our open Internet search for WSDL files. The table additionally shows which development environment was most popular based on the classification of the Web service. |
We introduce a matching algorithm called the tendency-based syntactic matching-LD/LP (TSM-LP) algorithm, which exploits all Web services development tendencies that we discovered, including the four we mentioned. TSM-LP uses subsumption, LD, and LP, but its core is the tendency-based thresholds, FT1 and FT2, which we use to govern the LD algorithm, LD, and LP algorithm, LP, when comparing two strings, Si and Sj. TSM-LP is governed by two thresholds that are closely tied to a Web service category’s uniqueness. Table 2 defines the TSM-LP algorithm – to summarize, if the LD and LP to see if they’re equal, which is common in related work (see the sidebar). As previously mentioned, numerous occurrences exist in which developers of different services used the same name for similar message types. However, more commonly, one developer used a two-word description of an idea, whereas another used one word to convey the same information. Approaches for leveraging tendencies 1 and 4 are equally straightforward, programmatically. For tendency 1, our software evaluates Web services parts to see if the first string is a subset of the second string or if the second string is a subset of the first. In addition, our approach disregards part names that are smaller than three characters or larger 15.

We used several syntactic approaches to exploit tendencies 2 and 3 when matching services. The Levenshtein distance (LD; also called the edit distance) is a measure of similarity between two strings. In our work, we adapt implementations of the LD algorithm (www.merriampark.com/ld.htm). LD is the smallest number of deletions, insertions, or substitutions required to transform a source string, s, into a target string, t. The greater the LD, the more different the strings are. For example, if $s = “test”$ and $t = “test,”$ then $LD(s, t) = 0$ because no transformations are needed. The strings are already identical.

- if $s = “test”$ and $t = “tent,”$ then $LD(s, t) = 1$ because one substitution (change “s” to “n”) is required to transform $s$ into $t$.

The LD algorithm is effective when evaluating abbreviations with full strings and for similar strings that are changed to create uniqueness. There are numerous occurrences discovered within our repository in which developers substitute zeros for the letter 0. Although LD is effective for similar strings, it isn’t effective for strings with similar subsets that don’t have true subsumption relationships as in tendency 1. For example, $\text{last_name}$ and $\text{surname}$ are equivalent, but neither is a subset of the other. To account for instances of this nature, we used the letter pairing (LP) approach, which matches strings that have common subsets. Using the LP algorithm, we separate two strings into letter pairs – for example, we’d separate $\text{STR1}$ into $\text{ST}$, $\text{TR}$, and $\text{R1}$, and $\text{STR2}$ into $\text{ST}$, $\text{TR}$, and $\text{R2}$. The letter pairing algorithm multiplies the number of identical pairs between the two strings by two and divide it by the total number of pairs to find the LP similarity percentage. In this case, the LP similarity would be 4/6 because both strings share two pairs and double two is four. The total number of pairs is six, three from each string, so the percentage is 66.7. An innovation in our approach (described later) is using the self-similarity of specific categories within the repository to determine the LD and LP thresholds (sensitivity) dynamically. As such, we can adjust these thresholds to values that are most effective depending on the category (domain) of the Web services being analyzed.

### Table 2. The tendency-based syntactic matching-Levenshtein distance/letter pairing algorithm.

<table>
<thead>
<tr>
<th>TSM-LP($Si$, $Sj$)</th>
<th>TSM-L function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$LD(Si$, $Sj)$</td>
<td>Levenshtein distance function</td>
</tr>
<tr>
<td>FT1($Si$)</td>
<td>Tendency-based threshold</td>
</tr>
<tr>
<td>FT2($Si$)</td>
<td>Tendency-based threshold for letter pairing</td>
</tr>
<tr>
<td>Si, Sj</td>
<td>Two strings for comparison</td>
</tr>
<tr>
<td>Length()</td>
<td>String length functions</td>
</tr>
<tr>
<td>CS</td>
<td>Web services category (such as Business)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FT1($Si$)</th>
<th>$temp = [(\text{Length}(Si) \times 2) / 3] - 2$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>return $temp$</td>
</tr>
<tr>
<td>FT2($Si$)</td>
<td>$temp = \text{Sensitivity (CS)}$</td>
</tr>
<tr>
<td></td>
<td>return $temp$</td>
</tr>
<tr>
<td>TSM-LP($Si$, $Sj$)</td>
<td>if ($LD(Si$, $Sj) &lt;= FT1(Si)$) or ($LP(Si$, $Sj) &gt;= FT2(Si)$) or ($Si \subseteq Sj$ or $Sj \subseteq Si$) ($Si &gt; 3$ and $Sj &gt; 3$) and ($Si &lt; 15$ and $Sj &lt; 15$) return $\text{TRUE}$ else return $\text{FALSE}$</td>
</tr>
</tbody>
</table>
are within the tendency-based thresholds, or if either string is a subset of the other and they’re both greater than three and less than 15, then our approach considers them similar.

**Similarity for Classification and Recommendation**

As previously mentioned, we separated our repository into nine categories. We can assume that the WSDL parts of Web services within the same category should be similar, equivalent, or, in some cases, the same. We anticipated that the categories would differ in their percentages of uniqueness because their content has differing natures (domains), but we still expected signs of overall self-similarity by category.

**Assessing Self-Similarity within Categories**

Through analysis, we found that all of our categories except one showed high self-similarity and thus strong correlation between their services’ WSDL parts. Figure 3 shows the WSDL parts’ uniqueness by category. For example, only 12 percent of all the business and economy parts are unique (fewer than 1,200 out of nearly 16,000 parts). This means that although thousands of instances of WSDL parts exist, the various names developers chose are rather limited, making syntactical searching relatively effective for this category. Military services were unique because they seemed to have distinct functionality in which each service has a uniquely different capability. A syntactical approach’s effectiveness increased when we supplemented it with a similarity approach such as the one we employed in our repository analysis. Figure 3 shows how we can customize the TSM-LP algorithm based on the repository’s uniqueness.

**Using Similarity for Recommendation: An Experiment**

A practical application for TSM-LP is using it to recommend Web services to a user in the context of their business processes (that is, our software can examine XML-based messages and compare them against available Web services). More specifically, automated software can capture an HTML file during an operational session and condense the file into a bag of words. Our software can then generate a recommendation score by summing the number of times a word within the bag of words is similar to specification data representing a particular service.

We can increase the recommendation’s precision using thresholds that we calculate from the uniqueness of the category to which the service belongs (as in Figure 3).

In the experiment Figure 4 shows, we copied the text from three different business- and economy-related Web pages (Google, Yahoo, and CNN). Although related approaches operate at the WSDL specification level, our approach evaluates at the level of the underlying operations (some WSDL files can contain as many as 50 independent operations). We evaluated our recommendation technique using the entire repository of 6,019 operations and varied the underlying matching algorithm across seven approaches. We used

- equivalence and subsumption relationship (subsumption),
- letter pairing (LP),
- edit distance (LD),
- equivalence, subsumption, and edit distance (TSM-L),
- equivalence, subsumption, and letter pairing (TSM-P),
- TSM-LP without category sensitivity, and
We evaluate the algorithms based on common performance measures for information retrieval, precision, and recall. Precision, $Pr$, (as formalized in Equation 1) is the percentage of returned results that were correct. We calculated this by dividing the number of correctly returned Web services operations, $Cr$, into the total number of returned Web services operations, $Tr$, Recall, $Re$, considers the entire search space; we calculated it by dividing the number of correctly returned Web services operations, $Cr$, into all possible appropriate Web services operations in the repository, $Ac$:

$$Pr = \frac{Cr}{Tr} \tag{1}$$

$$Re = \frac{Cr}{Ac} \tag{2}$$

Let’s use a simple example to further explain these metrics. Suppose we have a repository of 10 services, and of those services, four are relevant to one particular search. An algorithm that returns those four services and two more (for a total of six) would have 100 percent recall and 67 percent precision. The algorithm recalled 100 percent of the correct results that exist, so a person could use one of the returned services with 67 percent assurance (that is, precision) that the service is relevant to his or her query.

We recorded the average recall and precision measures after running the recommendation algorithm separately for each of the three input Web pages. Considering precision and recall, TSM-LP and TSM-LP-Sens are comparatively more effective than all other approaches, as Figure 4a shows. In Figure 4b, we further evaluate the precision of just the top 50 returns. Again, we take the average precision score, within the top 50, for each of the three HTML pages. In practice, our application would only use a smaller subset of the most relevant returns as recommendations to the user. As such, we find the precision in this subset to be roughly 59 percent for TSM-LP-Sens, which performs favorably with regard to related approaches of syntactic interface similarity\cite{4,6} (which have average precision/recall of 20 percent/72 percent and 52 percent/98 percent, respectively). Our results are further strengthened given that our experimentation considers 596 service descriptions whereas the related approaches have repositories of fewer than 40 service descriptions. The fact that we operated on more services than related approaches also explains why, in some cases, our precision is lower. In addition, our approach doesn’t use service specifications as input (which might be more exact) but instead is initiated with a randomly collected HTML file that’s applied to a relevant service category.

By analyzing real Web services from the Internet, we determined that operation names and message names are reasonably self-similar. We experimented with natural language processing techniques to further determine equivalency among the services. Our approach attempts to mimic human tendencies to make a knowledge-able inference as to when two similar messages are equal. This is a “syntax first” type of method, so the natural strength of our similarity approach is with recommendations that result in human-in-the-loop integration. At this point,
Related Work in Syntactical Web Service Discovery

Generally, the work described in the main text is similar to the body of work in matching software specifications.1,2 More specifically, in a services context, Yiqiao Wang and Eleni Stroulia3 combine syntactical approaches with information retrieval techniques to find new Web services that match a provided service. With favorable results, their approach identifies direct equivalence between the data underlying potentially equivalent services. Unlike Wang and Stroulia’s work and other approaches that match services based on equating strings and structure,4 we exploit loose matching approaches in which direct equivalence isn’t always necessary. By exploiting human nature and tendency, we syntactically interpret when different strings are equivalent. For example, our approach would understand that “building,” “building name,” and “bldg” are equivalent by exploiting customized natural language processing (NLP) approaches. Whereas related work divides interface specifications into categories,1,2 our work groups all extracted strings, regardless of their category, together into one bag of words for matching. In contrast, we categorize the search repository. Seog-Chan Oh and his colleagues3 also combine NLP approaches with semantic approaches, but their work neglects the nature of the repository being searched. In our work, we analyze the uniqueness of strings in the search repository (by category) to set the sensitivity of our underlying matching algorithms.

During business processes, users can open Web pages, send messages to other stakeholders, or indirectly initiate Web services. To identify potentially, relevant services, our approach can collect and use, in the background, the data that propagates during business processes. Yiqiao Wang and Eleni Stroulia use candidate services as the catalyst for discovery; our approach extends theirs by also exploiting business process data in the form of random HTML files, text messages, documents, or SOAP messages. This extension also gives our approach more flexibility than Web service search engine approaches, such as Woogie,4 which rely on human-generated queries.

References


M. Brian Blake is an associate professor and chair in the Department of Computer Science at Georgetown University. He conducts applied research in the areas of service-oriented computing and enterprise integration. Blake has a PhD in information and software engineering from George Mason University. He’s a senior member of the IEEE. Contact him at Blakeb@cs.georgetown.edu.

Michael F. Nowlan is an undergraduate research assistant in the Department of Computer Science at Georgetown University pursuing a BS in computer science. He conducts research in the intelligent exchange of messages and data for Web services in service-oriented architectures. Contact him at mfn3@georgetown.edu.

it would be difficult to automatically integrate services during dynamic composition routines, but it might be practical to offer service alternatives to developers and businesses in real time. Also, we can use our approach to augment UDDI search mechanisms with the addition of similarity-based indexing. In future work, we plan to develop automated approaches that determine how many recommendations we should make. [2]

References