

Taming Web Services from the *Wild*

M. Brian Blake and Michael F. Nowlan

Department of Computer Science, Georgetown University,

Washington, DC 20057, (202) 687-3084, {mb7,mfn3}@georgetown.edu

Abstract. Building new cross-organizational applications by combining, composing, consuming, and/or interconnecting existing services is what service-oriented computing (SOC) is all about. So, why do most composite web service-based systems currently rely on pre-established relationships that are not created by automated, dynamic discovery and integration? One perceived reason is the inconsistency in service-based interface descriptions and message names. Semantically-described interface specifications have been introduced to deal with this heterogeneity, however these approaches are tedious, unwieldy, and currently not widely used in practice. In our work, we investigate if human nature, specifically software developers' tendencies to name service descriptions in significantly consistent ways, can provide syntactical methods for service discovery. We performed a wide search for web services that are openly accessible on the Internet (*i.e. from the Wild*) and analyzed their naming tendencies. As a result, we found that the *harvested* services, although not comprehensive enough for meaningful domain-specific compositions, are more malleable than one might think when automating the search for relevant services in a general business process context.

Keywords. Web Services, Service-Oriented Computing, Service Discovery, Syntactical Analysis

1.0 Introduction

Web services are at the core of service-oriented architectures (SOA) and the SOC paradigm [7]. Web services can be defined as networked capabilities with openly accessible interfaces such that they can be discovered and executed by other machines, in real-time. Web service composition involves connecting a chain of multiple, domain-related services [8][9], however, in practice, discovering services is the first step. Several international forums have been established to encourage researchers and industry engineers to develop service discovery/composition systems that can autonomously, perhaps intelligently, compose web services (e.g. the Web Services Challenge (WSC), the Semantic Web Services Challenge (SWS),

and the Services Computing Contest). Each of these venues creates their own web service repositories (i.e. with *clean* interfaces) as opposed to using services available over the Internet.

In this work, we are interested in web services that were not created in a controlled, homogeneous environment. As such, we collected as many Web Service Description Language(WSDL) documents (i.e. service specifications) as we could acquire from various repositories and sources openly available over the Internet. By analyzing these *real* services at greater depth, it is possible to interpret the tendencies of service developers to use predictable phrases or nomenclature when creating the interfaces and service messages. A goal of this approach is to integrate the observed tendencies with straightforward syntactical approaches, such that a quick, just-in-time service management approach can be achieved. Although ambitious, success in this approach might effectively facilitate service discovery and composition without the many specifications required by full semantic techniques (i.e. ontology-based semantic description languages such as the Resource Description Framework (RDF) or the Web Ontology Language for Services (OWL-S) [6]). A fully syntactic approach cannot completely replace the need for semantic descriptions, but rather enhances semantic approaches by recommending services based on their underlying specification data prior to semantic processing and, in effect, *taming* the target services repository. An application of the approach described here is the decomposition of data from specified business process routines into keywords that can be matched to keywords extracted from candidate services. As a result, we introduce automated approaches for identifying services that are relevant to a designated, operational setting.

2.0 Related Work

Generally, this work is similar to the body of work in matching software specifications [3] [15]. More specifically, in a services-context, Wang and Stroulia [10] 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 [13], we exploit *loose* matching approaches where direct equivalence is not always necessary. By exploiting human nature and tendency, we syntactically interpret when different strings are equivalent. As an example, our approach would understand that "building", "building_name", and "bldg" are equivalent by exploiting customized natural language (NLP) approaches. While related work divides interface specifications into categories [3] [15], our work groups *all* extracted strings, irrespective of their category, together into one bag of words for matching. In contrast, we categorize the search repository. Oh et al. [5] also combine NLP approaches with semantic approaches, but their work neglects the nature of the repository being searched. In this 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 may open web pages, send messages to other stakeholders, or indirectly initiate web services. The data that propagates during business processes can be collected and used, in the background, to identify potentially, relevant services. Wang and Stroulia use candidate services as the catalyst for discovery, our approach extends their approach 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, i.e. Woogle [2], that rely on human-generated queries.

3.0 Processing WSDL Documents

Based on the W3C WSDL specification [11], a WSDL document (meta-model shown in Figure 1 as a Unified Modeling Language class diagram) describes web service attributes based on specialized XML elements such as *service*, *ports*, *bindings*, *operation*, *messages*, and *parts*. The *service* element names the web services and includes multiple physical locations, described as *ports*. Each port can have a *binding* to multiple *operations* (i.e. atomic elements of work). Operations are further defined by *messages* containing *parts* (i.e. *inputs* and *outputs*).

When analyzing naming tendencies within web services, we used three basic steps for capturing the operation names and the corresponding messages as shown in the example illustrated by Figure 1, which uses a WeatherForecast WSDL document. In the first step, our software gathers all *operation* names from all of the *ports* elements in the WSDL file. In the second step, for each *operation* name captured in the first step, our software extracts all *input* and *output* elements and their corresponding *message* and *part* elements (also shown by example in Figure 1). For some development platforms (e.g. Microsoft .Net platform) where hierarchical description is enabled, a third step is required. This third step is required when an *element* attribute appears in place of a *type* attribute within the WSDL *message* element. When this occurs, a keyword such as “parameters” or “body” is used as a placeholder in the *name* attribute. The reader should notice the difference between “WSDL Message Part from Type Elements” and “Inline WSDL Message Parts” in Figure 1. In the former situation, the low-level message information is not captured in the *message* element (i.e. as in the latter), but rather, they can be found within WSDL types. As such, the WSDL document is traversed to locate the *type*

elements that are referenced in each of the *part* elements extracted in Step 2. Similar to related work [10], we use recursive processing to capture nested/hierarchical relationships within specifications and subsequently drill-down to extract the actually message strings.

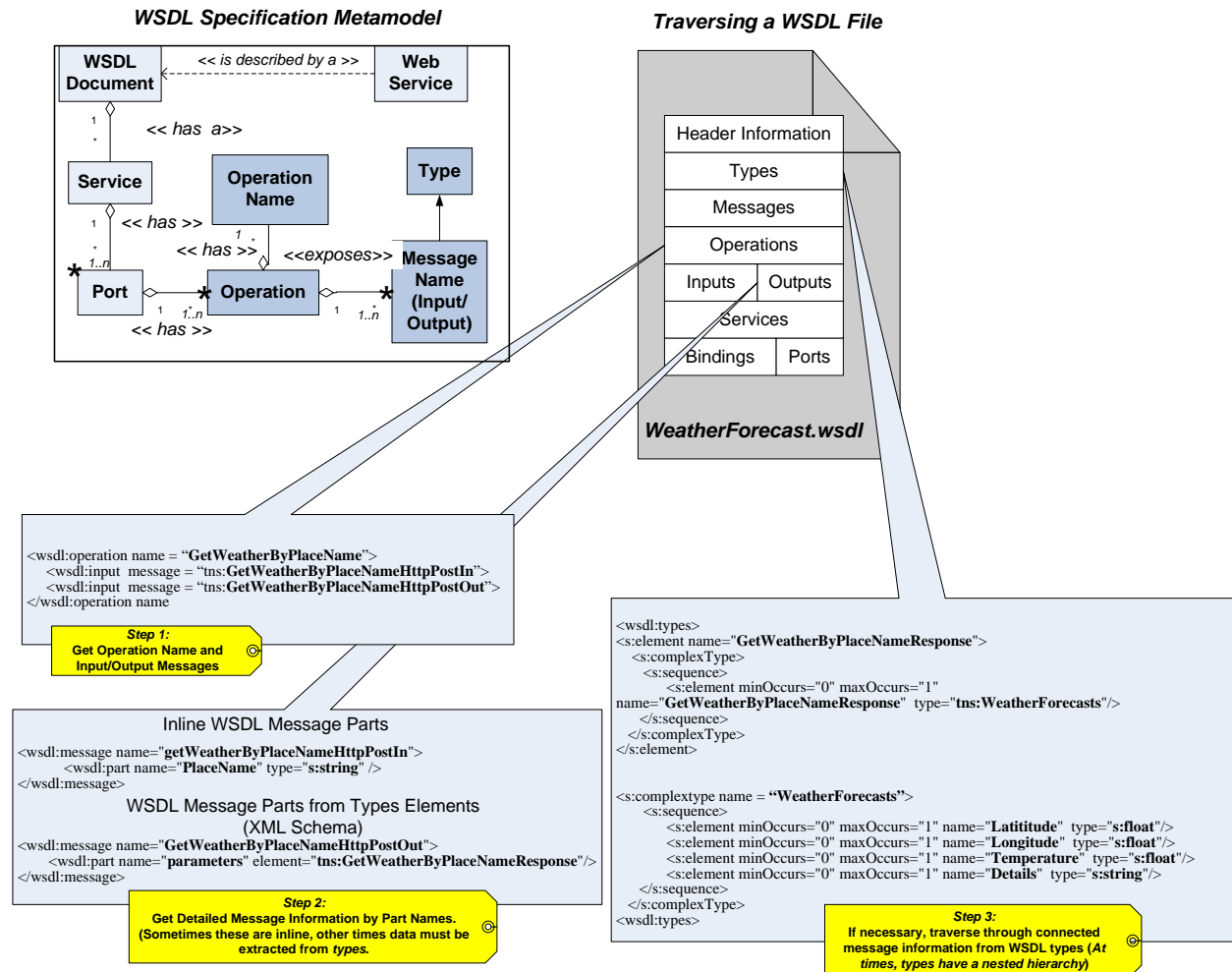


Figure 1. Three Steps for Gathering Operation Information and Messages.

Suggested Side Bar: What Do Harvested Web Services Look Like?

We manually downloaded 596 real web services (i.e., WSDL documents) from the Internet from a number of web services repositories (i.e. Woogole, XMLMethods, Salcentral, BindingPoint, and WebServiceList) [14], Amazon Corporation [1], and random Internet searches. The 596 service descriptions can be decomposed into 31,807 total message *parts* (i.e. parts are the independent parameter strings that comprise a web service message). 18,815 parts remain once duplicate entries (i.e. the same part name occurring multiple times in the same WSDL document) are removed. Finally, once all duplicates are removed, there are 5,180 unique parts across all services in the repository, regardless of input or output. There are more output parts than input parts, suggesting that input information is slightly more homogeneous than output information. Further evidence of this claim is that 20% of output parts (after removing duplicates) are unique across services, while the inputs are just 13% unique. The knowledge that developers are more likely to use the same or similar parts for input messages may be helpful, with regards to search speed, for consumers who index results from Universal Description, Discovery and Integration (UDDI) registries. Table 1 lists the quantitative details of the repository.

Table 1. Detailed Information about the Repository

Statistic Description	Input	Output	Total
Number of WSDL Documents			596
Gross Number of WSDL <i>Parts</i>	14,972	16,835	31,807
-----	-----	-----	-----
Number of WSDL <i>Parts</i> (Only keeping Unique Names within each WSDL)	8,047	10,768	18,815
Overall Unique WSDL <i>Parts</i> (195 names overlap input and output)	1,939	3,436	5,180

In evaluating the WSDL files, we were able to determine information in the header that uniquely identified which web service development platform was used. Of the repository,

Microsoft .Net services were overwhelmingly most popular. Figure 2 shows the percentage of each development platform that was represented.

WS Development Platform	Percent Representation	Most Frequent Category
Microsoft .Net	67.79%	Business and Economy
Misc/ Not Determined	16.11%	Technology
Openuri.org (NetBeans)	4.87%	Military
Borland	2.85%	News and Reference
WebMethods	2.85%	Technology
Apache SOAP	2.68%	News and Reference/Technology
Sun (SOAPInterop)	2.52%	Communication
ColdFusion	2.35%	Entertainment/Technology
Apache Axis	0.84%	News and Reference

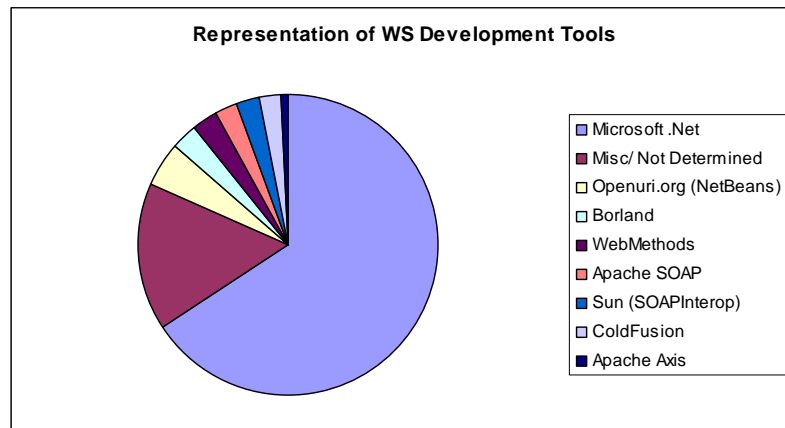


Figure 2. Representation of Various Development Environments in the Repository.

Similar to categories devised in the Woogle web service search engine [2], we divided our repository into nine categories: Calendar, Business and Economy, News and Reference, Graphics, Communication, Technology, Conversion, Entertainment and Military. The last category (i.e. Military) contains web services openly available for government defense information systems. Figure 2 also shows the most frequent type (i.e. category) of service created by each development kit.

4.0 Predicting Similarity Using Human Development Tendencies

Identifying equivalent keywords is the key to correlating relevant web services to real business operations. Table 1 shows that, of the maximum possible unique WSDL parts, only approximately 25% are truly unique within the entire repository. This is promising considering

that services were built by different organizations and suggests that web services developers, perhaps instinctively, use the same names for common types of messages. Nevertheless, identifying relevant services by exactly matching WSDL parts is not practical. We devise a more flexible approach that uses the similarity of WSDL parts 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, although they do not exactly match?” In answering this question, we sorted the repository then analyzed similar phrases using manual inspection. Then using semi-automated approaches, we determined human trends in naming the web services messages with high frequency. Based on the determined trends, we use complimentary natural language processing techniques to exploit those trends and determine equivalence. As a result of the analysis many tendencies were discovered but four examples are described here.

Tendency 1 (Subsumption Relationships). There is a strong tendency 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*.

Tendency 2 (Common Subsets). Similar to subsumption relationships, some web service parts are related by having common subsets. For example, we found equivalent WSDL parts where *first_name = user_name*.

Tendency 3 (Abbreviations in Naming). Another strong tendency was for common phrases to be shortened into abbreviations. For example, *building = bldg* or *country = cntry*.

Tendency 4 (Size constraints). Strings shorter than 3 characters and longer than 15 characters are ineffective for matching part names.

4.1 Exploiting Naming Tendencies to Find Relevant Services

Based on the previously defined tendencies, we developed an algorithm that exploits those tendencies to discover when the web service part names are equivalent. Probably the most straightforward comparison for two part names is to check to see if they are equal, which is common in related work. As previously mentioned, there are a number of occurrences when developers of different services did decide to use the same name for a similar type of message. However, a more likely occurrence are the many cases where one developer used a two-word description of an idea, while another used one word to mean the same information. Approaches to exploit tendencies 1 and 4 are equally straightforward, programmatically. Web service parts are evaluated to see if the first string is a subset of the second string or if the second string is a subset of the first string. In addition, this approach disregards part names that are smaller than 3 characters or larger than 15 characters.

Using Levenshtein Distance and Letter Pairing for Tendencies 2 and 3

Several syntactic approaches are used to exploit Tendency 2 and 3 when matching services. The Levenshtein distance (LD) (also called the *edit distance*) is a measure of similarity between two strings. The 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 $s = \text{"test"}$ and $t = \text{"test"}$, then $LD(s,t) = 0$, because no transformations are needed. The strings are already identical.
- If $s = \text{"test"}$ and $t = \text{"tent"}$, then $LD(s,t) = 1$, because one substitution (change "s" to "n") is required to transform s into t .

In our work, we adapt implementations of the LD algorithm [4]. The LD algorithm is effective when evaluating abbreviations with full strings. In addition, LD is also effective for similar strings that are changed to create uniqueness. There are a number of occurrences where zeros are substituted for the letter O . Although LD is effective for similar strings, it is not effective for

strings that have similar subsets that do not have true subsumption relations as in Tendency 1. For example, *last_name* and *surname* are equivalent but neither is a subset of the other. To account for instances of this nature, we employed the use of Letter Pairing. The Letter Pairing (LP) approach is an algorithm that can be used to match strings that have common subsets. Using the LP algorithm, two strings are separated into *letter pairs*. STR1 would be separated into *ST*, *TR*, and *RI*, and STR2 would be separated into *ST*, *TR*, and *R2*. The number of identical pairs between the two strings multiplied by two, is divided by the total number of pairs to find the LP similarity percentage. In this case the LP similarity would be 4/6, because there are two pairs shared by both strings and double two is four. The total number of pairs is 6, three from each string, and 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, these thresholds can be adjusted to values that are most effective depending on the category (i.e. domain) of the web services that are being analyzed.

4.2 Tendency-Based Syntactical Matching-Levenshtein Distance/Letter Pairing (TSM-LP)

We introduce a matching algorithm called *Tendency-Based Syntactical Matching-Levenshtein Distance and Letter Pairing (TSM-LP)*. This algorithm exploits all of our tendencies, including the four previously mentioned tendencies, using subsumption, LD, and LP. The core of TSM-LP is the Tendency-Based Thresholds, F_{T1} and F_{T2} , that we use to govern the LD algorithm, L_D , and LP algorithm, L_P , when comparing two strings, S_i and S_j . TSM-LP is governed by two thresholds that are closely tied to the uniqueness or *Sensitivity* of the category of web services. In summarizing the TSM-LP algorithm defined in Table 2, if the LD and LP are within the

Tendency-Based Thresholds or if either of the strings is a subset of the other and the strings are both greater than 3 and less than 15, then the strings are considered similar.

$TSM-LP(S_i, S_j)$:	TSM-L Function
$L_D(S_i, S_j)$:	Levenshtein Distance function
$F_{T1}(S_i)$:	Tendency-Based Threshold
$F_{T2}(S_i)$:	Tendency-Based Threshold for Letter Pairing
S_i, S_j :	Two strings for comparison
$Length()$:	String length functions
C_S :	Web Service Category (e.g. Business)
$F_{T1}(S_i)$ $temp = [(Length(S_i) * 2) / 3] - 2$ return $temp$	
$F_{T2}(S_i)$ $temp = Sensitivity(C_S)$ return $temp$	
$TSM-LP(S_i, S_j)$ if $(L_D(S_i, S_j) \leq F_{T1}(S_i))$ or $(L_P(S_i, S_j) \geq F_{T2}(S_i))$ or $(S_i \subseteq S_j \text{ or } S_j \subseteq S_i)$ and $(S_i > 3 \text{ and } S_j > 3)$ and $(S_i < 15 \text{ and } S_j < 15)$ return $TRUE$ else return $FALSE$	

Table 2. The TSM-LP Algorithm.

5.0 Experimentation: Similarity for Classification and Recommendation

As previously mentioned, our repository is separated into nine categories. It is reasonable to 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 of the differing natures (i.e. domains) of their content, 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, with the exception of one, showed high self-similarity and therefore, strong correlation between the WSDL parts of their services.

Figure 3 shows the *uniqueness* of the WSDL parts by category. For example, only 12% of all the Business and Economy parts are unique because of the nearly 16000 parts, less than 1200 are unique. This means that although there are thousands of instances of WSDL parts, the various names chosen by developers are rather limited, making syntactical searching relatively effective for this category. Military services were unique, because they seem to have distinct functionality. The effectiveness of a syntactical approach is increased when it is supplemented by a similarity approach such as the one we employed in our repository analysis. Figure 3 shows how the TSM-LP algorithm can be customized based on the uniqueness of the repository.

Using Similarity for Recommendation: An Experiment

A practical application for using TSM-LP is for recommending web services based on XML-based data being passed in context of service-oriented business processes. Automated software can capture an HTML file during an operational session and condense the file into a *bag of words*. A recommendation score can be generated by summing the number of times a word within the bag of words is similar to specification data from a particular service. The precision of the recommendation can be increased using thresholds that are calculated from the uniqueness of the category for which the service belongs (i.e. Figure 3).

Setting Thresholds Using the Nature of the Search Repository

Uniqueness Percentage	TSM-LP Sensitivity	LD Threshold	LP Threshold
10 - 25%	High	$\lceil (\text{Length}(S_i) * 2) / 3 \rceil - 3$	55.0%
25 - 50%	Medium	$\lceil (\text{Length}(S_i) * 2) / 3 \rceil - 2$	47.5%
50 - 75%	Low	$\lceil (\text{Length}(S_i) * 2) / 3 \rceil - 1$	40.0%

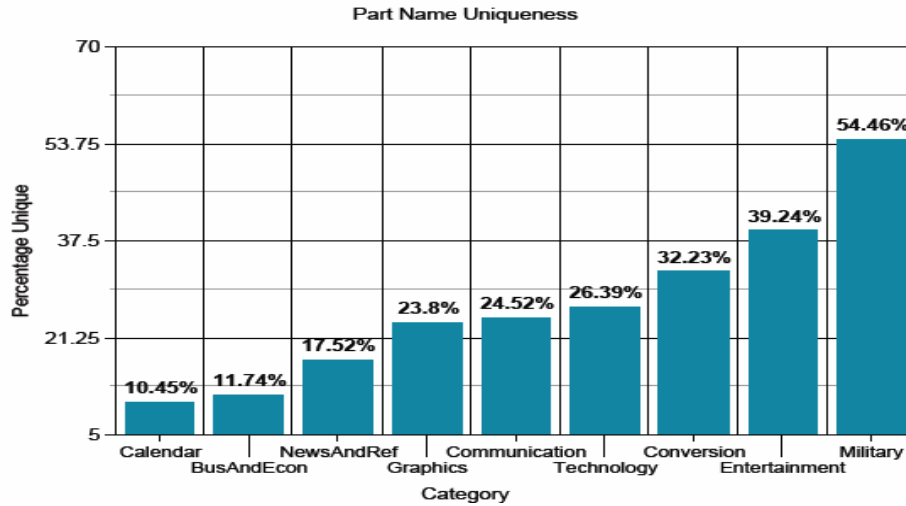


Figure 3. Percent Uniqueness by Category Type and How It Affects Matching Sensitivity.

In the experimentation shown in Figure 4, we copied the text from three different business and economy-related web pages (i.e. Google, Yahoo, and CNN). While related approaches operate at the WSDL specification level [10][13], 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 (1) equivalence and subsumption relationship (Subsumption), (2) letter pairing (LP), (3) edit distance (LD), (4) equivalence, subsumption and edit distance (TSM-L), (5) equivalence, subsumption, and letter pairing (TSM-P), (6) TSM-LP without category sensitivity, and (7) TSM-LP with category sensitivity (TSM-LP-Sens). We evaluate the algorithms based on common performance measures for information retrieval, *precision* and *recall*. Precision, Pr , (as formalized in (1)) is the percentage of returned results that were correct. This is calculated by dividing the number of

correctly returned web service operations, C_r , into the total number of web service operations that were returned, T_r . Recall, Re , considers the entire search space and is calculated by dividing the number of correctly returned web services operations, C_r , into all possible appropriate web service operations in the repository, A_c . A simple example can help to explain these metrics further. Suppose there is a repository of 10 services and of those services, 4 are relevant to one particular search. An algorithm that returns those 4 services and 2 more (for a total of six) would have 100% recall and 67% precision. The algorithm *recalled* 100% of the correct results that exist and a person could use one of the returned services with 67% assurance (i.e. *precision*) that the service is relevant to their query.

$$\text{Pr} = \frac{C_r}{T_r} \quad (1)$$

$$\text{Re} = \frac{C_r}{A_c} \quad (2)$$

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 shown by the upper graph in Figure 4. In the lower portion of the graph, 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 approximately 59% for TSM-LP-Sens. TSM-LP-Sens performs favorably with regards to related approaches of syntactic interface similarity [10][13] (i.e. average precision/recall of 20%/72% and 52%/98% respectively). Our results are further strengthened

considering that our experimentation considers 596 service descriptions while the related approaches have repositories of less than 40 service descriptions. The fact that we operate on more services than related approaches also explains why in some cases our precision is lower. In addition, our approach does not use service specifications as input (which may be more exact), but instead is initiated with a randomly collected HTML file that is applied to a relevant service category.

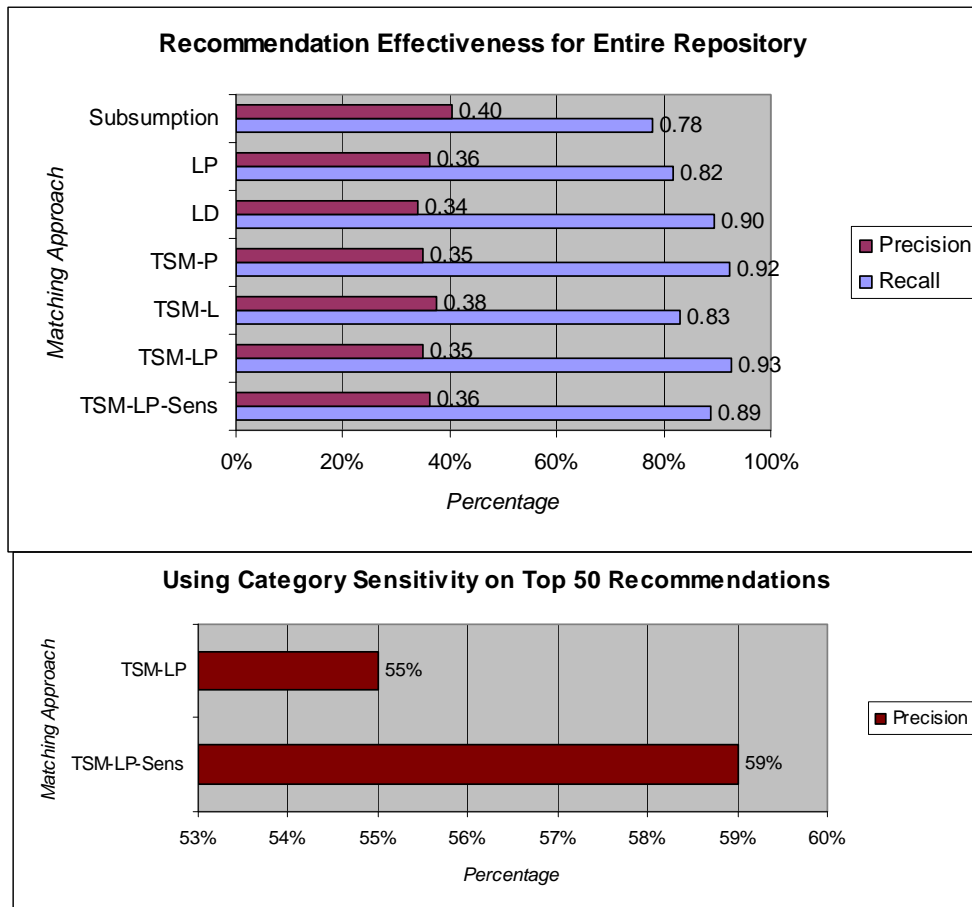


Figure 4. Precision and Recall for Recommendations from the Repository and the Top 50 Results.

6.0 Conclusions

By analyzing real web services from the Internet, we determined that operation names and messages 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 knowledgeable inference as to when two similar messages are equal. This is a “syntax-first”-type of method and the natural strength of our similarity approach is with recommendations that result in human-in-the-loop integration. At this point, it would be difficult to automatically integrate services during dynamic composition routines, but it may be practical to offer service alternatives to developers and businesses in real-time. Also, this approach may be used to augment UDDI search mechanisms with the addition of similarity-based indexing. In future work, we also plan to develop automated approaches that determine how many recommendations should be made.

References

- [1] Amazon Web Services (2006): www.amazon.com/gp/aws/landing.html
- [2] Dong, X., Halevy, A.Y., Madhavan, J., and Nemes, E., Zhang, J. Similarity Search for Web Services. VLDB 2004
- [3] M. Godfrey and L. Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities," IEEE Transactions on Software Engineering, vol. 31, pp. 166-181, 2005.
- [4] Merriam Park Software (2007): <http://www.merriampark.com/ld.htm>
- [5] Oh, S-C., Kil, H., Lee, D., Kumara, S. “WSBen: A Web Services Discovery and Composition Benchmark” In IEEE Int'l Conf. on Web Services (ICWS), pp 239-246, Chicago, IL, USA, September 2006
- [6] OWL-S (2007): <http://www.daml.org/owl-s/>
- [7] Papazoglou, M. “Service-oriented computing: Concepts, characteristics and directions. *In Proceedings of WISE '03*
- [8] Rao, J. and Su, X. "A Survey of Automated Web Service Composition Methods", *In Proceedings of the First International Workshop on Semantic Web Services and Web Process Composition, SWSWPC 2004*, San Diego, California, USA, July 6th, 2004
- [9] Sirin, E., Hendler, J., and Parsia, B. “Semi-automatic composition of Web services using semantic descriptions”, *In Proceedings of Web Services: Modeling, Architecture and Infrastructure workshop in conjunction with ICEIS2003*, 2002.
- [10] Wang, Y. and Stroulia, E., “Flexible Interface Matching for Service Discovery”, In Proceeding of the International Conference on Web Information Systems Engineering (WISE2003), pp. 147- 156
- [11] Web Services (2007): <http://www.w3.org/2002/ws/desc/>
- [12] WS-Challenge (2007): <http://www.ws-challenge.org/>
- [13] Wu, J., Wu, Z.: ”Similarity-based Web Service Matchmaking”, IEEE International Conference on Services Computing, 2005, pp. 287-294
- [14] XMethods (2007): <http://www.xmethods.com/>
- [15] Zaremski, A.M. and Wing, J. M. “Specification matching of software components”, ACM Transactions on Software Engineering and Methodology, 6(4):333–369, 1997.