A Query-based System for Automatic Invocation of Web Services

Chaitali Gupta, Rajdeep Bhowmik, Michael R. Head, Madhusudhan Govindaraju, Weiyi Meng
Department of Computer Science, State University of New York (SUNY) at Binghamton, NY
{cgupta1, rbhowmi1}@binghamton.edu {mike, mgovinda, meng}@cs.binghamton.edu

Abstract

There is a critical need to design and develop tools that abstract away the fundamental complexity of XML based Web services specifications and toolkits, and provide an elegant, intuitive, simple, and powerful query based invocation system to end users. Web services based tools and standards have been designed to facilitate seamless integration and development for application developers. As a result, current implementations require the end user to have intimate knowledge of Web services and related toolkits, and users often play an informed role in the overall Web services execution process We employ a set of algorithms and optimizations to match user queries with corresponding operations in Web services, invoke the operations with the correct set of parameters, and present the results to the end user. Our system uses the Semantic Web and Ontologies in the process of automating Web services invocation and execution.[1]

Key Words: Automatic Invocation, Web services, Query matching, Semantic Web, Ontology.

1. Introduction

The Web services model has emerged as a standard for representation, discovery, and invocation of services in a distributed environment. A Web service can be defined as an interface to application functionality that is accessible using well-known Internet standards and is independent of any operating system or programming language. The widespread adoption of Web services is enabled by a set of flexible and extensible XML based standards including the Web Service Description Language (WSDL), which is the widely used specification to describe Web services. Web services are widely expected to simplify the design of distributed applications that are amenable to automated discovery, composition, and invocation. The use of XML facilitates in moving towards loosely-coupled applications that provide greater interoperability in distributed heterogeneous environments. However, the current XML based specifications provide only syntactical descriptions of the functionality provided by Web services. Even though a wide variety of tools are available to invoke Web services, the lack of semantics associated with Web service descriptions requires user intervention in the decision make process and understanding of the complex interfaces, contexts, composition, and invocation. It is currently required for application developers to have some knowledge of the intricacies of Web services: know-how about the list of available operations, the input and output parameter types, the set of ports and service endpoints, apart from usage details of their particular Web services toolkit (WSIF [1], gSOAP [2], Axis [3], for example). Though an important motivation of Web services is to promote ease-of-use for application developers, the requirement that end users also be familiar with the design and some implementation details makes its usage difficult for end users. Our work addresses this problem by simplifying the user interaction with Web services. We have developed several algorithms and optimization techniques that map user queries to relevant operations in domain specific Web services. Our system presents a simple interface, similar to HTML based search engines, which accepts user queries and presents the end user with results after invoking and executing relevant Web services. We employ several query matching techniques including Semantic Web [4] and ontology technologies such as OWL [5], as well as tools such as WordNet [6], to retrieve contextual information from queries and determine the set of Web services that need to be invoked for any user query. The details of Web services specification and implementations are hidden from the user. For example, suppose a user wants to check the weather for a trip from Boston to Chicago. In our system, the user needs to enter a query "*weather for travel from Boston to Chicago*". Our system will employ various

[1] Supported in part by NSF grants IIS-0414981 and CNS-0454298

2007 IEEE International Conference on Web Services (ICWS 2007)
0-7695-2924-0/07 $25.00 © 2007 IEEE

algorithms to understand the query and obtain the required information by invoking the appropriate Web services. Unlike other Web service implementations, the user does not have to fill detailed forms for each service. Our system takes into consideration results of past Web service invocations and utilizes it to improve performance for subsequent user queries in the same domain. Our system also supports memorized optimization, which uses the knowledge of certain or entire parts of previously made queries for the benefit of future queries.

2. Design and Implementation

Figure 1 shows the components and control flow of our system. We describe each component in detail.

2.1. WSDL Processor

The WSDL Processor populates the data structures for all the WSDL files stored in the WSDL Repository with operation names, comments and annotations, part names, input and output parameters, the port types and the service endpoints corresponding to a particular WSDL file. The WSDL Processor is invoked only once at the start of the system and all the aforementioned WSDL file details are cached to improve performance for each subsequent client query.

2.2. User Query Interface

Our approach is to provide a simple user interface for invoking Web services. We have designed a general interface, as opposed to the use of forms specific to each domain as is currently implemented by many Web services invocation systems. In our system, depending on the results of the query matching algorithms, relevant Web services are invoked and executed without further involvement of the user in the overall process.

2.3. Query Processor

The Query Processor processes each user query and updates its learning engine whenever it encounters a new query. Since a user query cannot be predicted in advance, for the matching algorithms to succeed, the query has to first be normalized [7]. Words denoting a single concept can be depicted differently in a user query. We considered using Levenshtein Algorithm [9] to determine the similarity factor between the user query and the operation names in the WSDL files. Our results showed that it does not produce desirable matching results due to the diversity of user queries.

The normalization steps in our approach are:

- Non-content or stop words (such as "add", "fetch", "find", "check", and "what") are removed from the query string because they do not add any semantic value in our current processing system, and instead waste valuable processing time.
- All non-alphanumeric characters are not replaced with a space character. For example, a query like "*list of available hotels @ Salt Lake City on Monday*" is converted to "*list of available hotels at Salt Lake City on Monday*", taking into account the significance of '@' in context with the user query. In a query such as "today's $ to # conversion rate", the characters $ and # contribute to the contextual information. The query processor stores a list of non-alphanumeric characters that can be replaced with alphanumeric characters to better understand the query.
- Contents in parentheses are not removed. The information in parentheses provided by the user query can play a vital role in the OWL vocabulary for more accurate determination of the relevant Web services. For example, in a query string like "*Today's forecast (weather) at San Francisco*", the word "forecast" can be interpreted in different senses as a noun or a verb, which have different meanings. Since the user has provided the term "weather" in parentheses, it can be inferred that she is referring to today's weather condition at San Francisco.
- Our system uses word variants or stemming technology [8], which not only searches for the words present in the user query but also for similar words. This is implemented by domain independent technologies like thesaurus matching as well as by the use of Semantic Web and ontology technologies.
- Unlike popular search engines for HTML based Web pages; we do not automatically exclude common words like "to", "from" and "at" because such words in a user query can help determine the context of the query. For example, in the query "*Will it rain at Washington tomorrow?*" the preposition "at" indicates that the user wants to find out tomorrow's weather condition at some location. So we argue that prepositions including "to", "from", "at" and others can be vital indicators of the context of a particular query.
- Abbreviations are extended: for a query string like "flight booking to NYC", our system interprets it as "flight booking to New York City".

After the processing is done, the query words are stored in the Query Words Repository.

2.4. Lexicon

The Lexicon Block is built using the WordNet 2.0 Dictionary [6]. Our system uses JWNL 1.3 API [10] to

access the WordNet Dictionary. The functionalities provided by this block are extensively used by the Match Processor and the Relevance Checker in later stages. Apart from the dictionary methods, we also utilize the glossary feature provided by WordNet in the Relevance Checker module.

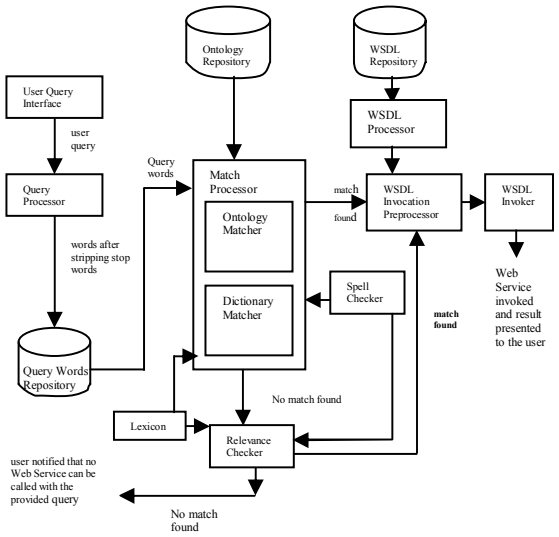

Figure 1: Design of the query processing system.

2.5. Spell Checker

The Spell Checker is taken into account to verify the correctness of the query words by both the Match Processor and the Relevance Checker, when no matches can be found for a particular query string. This helps in refining the user query. For example, consider the case when a query is "*chaepst fare from Chicago to Seattle*" instead of "*cheapest fare from Chicago to Seattle*". In this case, if the correct spelling of the word "cheapest" is found, it will result in a match, which otherwise would not have been possible. The Spell Checker itself can be implemented by invoking the *doSpellingSuggestion()* operation of the Google Web service API. However, this will incur an expensive overhead of making a remote invocation. The efficient alternative is to locally incorporate the Soundex [11] and Metaphone [12] algorithms.

2.6. Match Processor

The query words from the Query Words Repository are then fed to the Match Processor. The Match Processor consists of an ontology based processor namely *Ontology Matcher* and a thesaurus based processor called *Dictionary Matcher*.

2.6.1. Ontology Matcher. We have built ontologies to define popular search domains including "travel", "location", "currency", and "weather" in the initial experiments. We chose OWL [5] over RDF [13]/RDFS [14] because RDFS imposes loose constraints on vocabularies. For example, in RDFS, we can neither provide information that two properties are disjoint nor can we restrict the cardinality of a specific property. OWL, on the other hand, provides more accurate, precise, and complete representation of a domain. We have currently chosen OWL Lite to keep the ontology definitions simple. We plan to compare the performance with other OWL types in future work.

The Ontology Repository stores the vocabularies for a wide variety of domains. The ontology definitions are modeled in the Ontology Matcher using Jena [15]. Jena is a framework for building Semantic Web [4] applications. It provides a simple OWL API for processing vocabularies. The query words, fed to the Match Processor, are searched in the ontology models. These models consist of statements where each statement is made up of Subject, Predicate and Object. We do not search a query word in the ontology models if the word is a preposition. Instead, we indicate the presence of the preposition such as "at", "in", "of", "on", "from", "to" by a flag because these words can be vital indicators of the context of a particular query. For example, if we have a query string like "*Best price for flight from Los Angeles to San Francisco on Sunday*", we may infer from the prepositions that Los Angeles and San Francisco are geographic locations. If an ontology model is lit up by the query words that are searched in the models, then the corresponding sentence is returned and the query words are stored against the ontology domain. For example, Figure 2 shows how a simple query string ("*flight from Los Angeles to San Francisco*") lights up the travel ontology model. We can also deduce from the use of the prepositions "from" and "to" in the client query that Los Angeles is the originating location and San Francisco is the destination.

Within the Ontology Matcher, the Lexicon block is used and its features are employed to obtain better contextual information relevant to the client query. Initially, the query words are matched using direct keyword matching with the Subject, Object and Predicate of each ontology statement. Irrespective of the matches found, we use the Lexicon block to employ synonym, hypernym[2], and hyponym[3] matching

[2] Hypernym – A word whose meaning denotes super class.

techniques. By taking into consideration different senses of a particular word, we can ensure that the selected ontology file has the closest relevance to the client query string. We plan to study the effectiveness of Metanym Matching [16] in future work.

We consider only root words in the user query to avoid redundancy. For example, if a user query is "*Will it be snowing in Buffalo tomorrow?*" the word "snowing" in the client query string gets stemmed to the root word "snow", which is then retrieved from the Lexicon block. This approach is adopted because we presume that both the Lexicon and our ontology files will contain the corresponding root word.

For synonym matching, four different search outcomes are possible.

- Neither the query word nor the synonym words are present in any of the ontology models.
- Some of the synonyms are present, but not the query word.
- The query word is present, but not its synonyms.
- Both the query word and its synonyms are present in the ontology model.

We collect these outcomes, shown in Figure 3, and use them to extend the ontology models, thus enriching the model. We have designed a learning module that stores the knowledge and information of a previously made query (the semantics of which are not in our ontology) to later queries for predicting more accurate results. If both the query word and its synonym are not found, the ontology model does not get extended. The same condition applies when the query word and its synonym are both found within the ontology model. However, the ontology file is extended when a synonym of a particular word yields a match. We can infer that since a synonym of the query word is present in the ontology file, the query word very likely has contextual relevance to the ontology model.

Suppose we have a query "*temperature at Binghamton*" and we do not have the keyword "temperature" in our present ontology model. Further assume that from the Lexicon we can infer that "weather" is a synonym of temperature and "weather" is already present in the ontology model. It can then be inferred that "temperature" has a meaning which is semantically similar to "weather" and should be included in the ontology model. So we regenerate the weather ontology model and incorporate the keyword "temperature" in the ontology file. Each time any of the ontology models is updated, we create and read the new ontology model again so that the changes are incorporated. However, if a keyword from the user query string is present in the ontology model, every synonym of it does not qualify to be incorporated into the ontology model. For example, for the query string "*weather at Binghamton*", instead of "*temperature at Binghamton*", we will get "endure" as a synonym for weather from Lexicon. Since "endure" is not present in the ontology file, we cannot extend the model because the word "endure" carries a different sense that is not relevant to the present context of "weather".

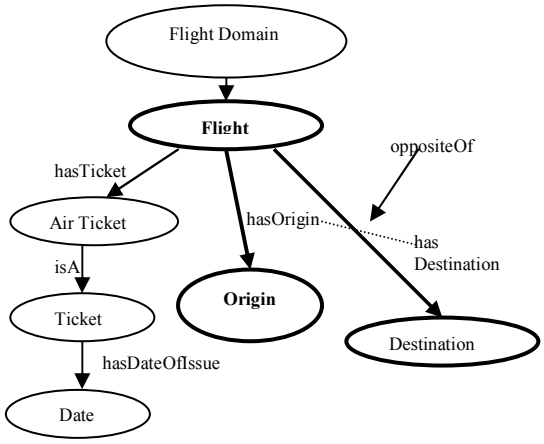

Figure 2: A query "flight from Los Angeles to San Francisco" lights up the travel ontology model.

In the first case of Figure 3, where the query word and its synonyms do not yield any positive result, the word is searched in a cached list of valid locations with the assumption that the word could be a valid location or a proper noun. We also inspect the first character of a query word to detect if it is in upper case. If so, it can possibly be inferred as a proper noun. If the word is found in the list of locations then the actual context information of the word can be obtained. Otherwise, we look up in Lexicon or invoke a location Web service [17] to determine if the query word denotes a corresponding location.

If the result is positive, the word is added in the cached list of locations. If the result is not positive, the Spell Checker is used to determine whether the query word is correctly spelled or not. We take into consideration varied senses of each query word to minimize the chances that more than one ontology model is selected for each search string.

2.6.2. Dictionary Matcher. If it is not possible to determine the context of the user query based on the ontology files, the Dictionary Matcher is used to obtain

[3] Hyponym – A word whose meaning denotes a subordinate or subclass. Cat is a hyponym of animal. It is the opposite of hypernym.

appropriate Web service operations. The matching algorithms it uses are:

- Stop words like "get", "add" and "fetch" are stripped from operation names and then the client query is matched with the operation names.
- Each value string is matched, ignoring its case, with the operation names. For example, *getTemperature* is matched with "temperature".
- Direct and Dictionary Level Matching is employed.

Query Word	Synonym	Result
0	0	Continue
0	1	Extend OWL file with query word
1	0	Continue, Synonyms may have different senses
1	1	Continue

Figure 3: Ontology Extension Table.

Our matching algorithms employed in the Dictionary Matcher works as follows:

Direct Matching – This is implemented by direct keyword matching. The client query is matched directly with the operation names and the stripped operation names. The stripping of the operation names is achieved by removing the stop words from the operation names. On a positive match, the operation names are stored in a specific data structure along with their WSDL file names.

Stripped Matching – The client query words are stripped and matched with the actual operation names as well as the stripped operation names. On a positive match, the WSDL files and associated operation names are stored.

Dictionary Level Matching - The matching algorithm uses Dictionary Level Matching, which includes synonym, hypernym, and hypernym of a particular key word. Here, synonyms of the query words are fetched and matched with the operation names and the stripped operation names. If the synonym match fails to provide a positive result then hypernyms and hyponyms of the query words are retrieved and matched with the operation names. On a positive match, operation names and corresponding WSDL files are stored.

2.7. Relevance Checker

If no match is found by the Match Processor, the system checks the glossary provided by the Lexicon as well as the input and output parameters of the methods, the part names and the comments and annotations in the WSDL files. This aspect of the system flow is executed by the Relevance Checker. The Lexicon component provides a detailed glossary for each word. It is possible that the query word cannot be found using ontology or direct, synonym, hypernym, and hyponym matching, but may be present in the glossary. For example, consider the query string "*forecast for Binghamton today*". We consider the query word "forecast" which contains the main contextual information in the user query. Suppose the word "forecast" is neither in the ontology model nor can it be found by dictionary level matching, because the Lexicon block does not provide any word related to weather in both cases. However, if the glossary of "forecast" is considered, "a prediction about something (as the weather) will develop" will be found. Therefore it can be deduced that "forecast" is related to weather, which helps in the invocation and execution process. Others features like input and output parameters and part names can also be beneficial in the matching process.

A feature that is often ignored but has relevance for query processing is comments and annotations in the WSDL file. If the operation names in the WSDL file do not truly reflect their actual cause but are annotated with suitable comments in the WSDL file, then it helps in determining which method to invoke to get an appropriate response. To facilitate automatic invocation of Web services, we recommend WSDL developers to use meaningful names and annotations in the WSDL file. For example, a weather-related WSDL file with an operation name *getWeatherForecast*, input message named as *getWeatherForecastRequest,* and output message named as *getWeatherForecastResponse*, will very likely be accessed by our query mapping system.

2.8. WS Invocation Preprocessor

After the Match Processor and Relevance Checker generate match results, we then search the selected WSDL files to check if there is enough information available in the query words so that an operation can be invoked. The semantics of operations present in the WSDL files are helpful in this case. The domain information gathered from the user query is matched with the semantic definitions of the signature of operations declared in the WSDL file. For example, the user query "*weather at Detroit*", when processed, gives us valuable information that the weather domain has been selected and the query is related to a geographic location. So any operation that takes a location attribute such as State, City or ZipCode as input parameter can possibly be invoked. In case of failure, text based searches are performed on the operation names. When all the inputs necessary to invoke a Web

service, such as Operation Name, its Input Messages, Output Messages, PortTypes, Service Endpoints are made available, they are passed on to the WS Invoker module.

2.9. WS Invoker

The WS Invoker is used to invoke and execute the selected operation and Web service. This module, which uses the Axis toolkit, creates the SOAP payload necessary to invoke the chosen Web service and presents the parsed results to the user.

2.10. Fallback Invocation Behavior

We compare query words with WSDL file names and if we cannot determine the set of operations to invoke, the user is presented with a form with links to methods for invocation. The user can then select any of the operation names and invoke the Web service. The information provided by the user is auto-filled in the form. If none of these techniques are successful and no matches can be found, the user is notified and requested to refine her query. This default behavior is similar to the capabilities of existing Web services invocation tools.

3. Experimental Results

We conducted experiments on a Dell D620 with an Intel T2300 processor @ 1.66 GHz and 1 GB of RAM running Microsoft Windows XP. We created a list of representative queries of varying non stop word lengths (from 1 to 10): "Seattle", "Binghamton's weather", "Temperature at Chicago", "How hot is it at Boston today?", "Will it be raining at Dallas tomorrow?", "Flight Details from Los Angeles to San Francisco", "Today's conversion rate from Euro to Dollar", "Stopovers from NYC to Philadelphia when traveling by bus", "Cost of hotel booking for one-night stay at NYC", and "Best price for flight from Pittsburgh to Buffalo on Sunday".

Figure 4 shows that if query results in a match in the ontology model, the total time required for invocation and execution of a Web service is less as opposed to a scenario where the user query does not yield a match in the ontology model and has to flow through secondary back-up matching techniques like dictionary level matching in the Match Processor. We expect that in most of the cases, an ontology model will be selected for a client query. In cases when the ontology model does not help in matching the query, the additional overhead of the Lexicon block will be incurred and increase the response time.

We present the performance results for memoized query strings in Figure 4. If a certain part of the user string, or the string in its entirety, is fed to the system for subsequent queries, performance improvement is significant as the Ontology Level and Dictionary Level matching blocks can be skipped.

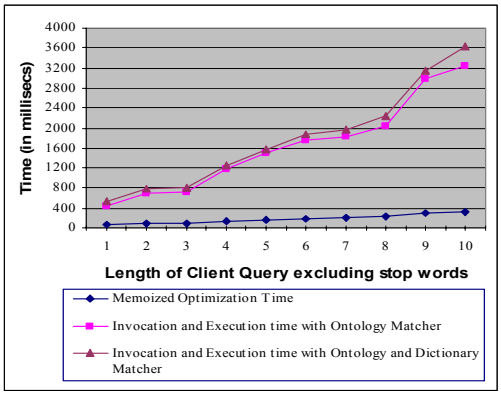

Figure 4: Queries that hit in the Ontology Matcher resolve more quickly than those requiring both Ontology and Dictionary Matcher.

Figure 4 also illustrates that the time required for Ontology and Dictionary Level Matching increase proportionally with the length of the client query strings. Since a larger query string with more non stop words provides better context information, it also results in more accurate processing.

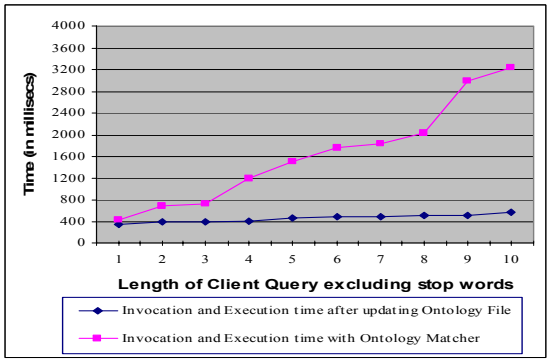

Figure 5: The Lexicon block adds a significant overhead to the query matching process.

Figure 5 illustrates that the time spent in finding the synonyms of client query words and the synsets[4] for extending the ontology models is a major

[4] A synset (synonym set) represents a concept and contains a set of words; each of which is synonymous with the other words in the synset.

bottleneck in the overall process. Our analysis showed that a major portion of the time is spent in loading the JWNL API implementation for generation of the synsets for each client query word. Implementing and incorporating the self-learning capabilities of the system, with the extension of the ontology models, is time consuming. Though the overhead ranges from 85% to 90%, it results in faster response time and enhanced accuracy.

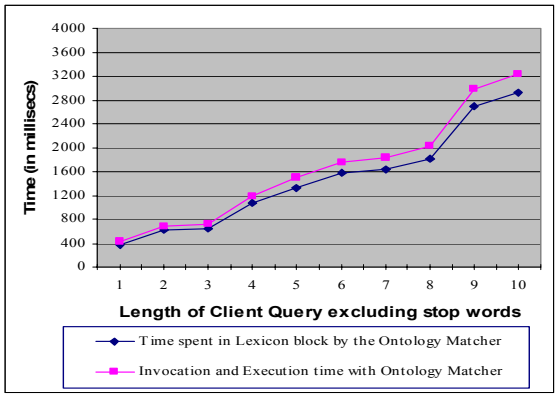

Figure 6: When the query words are extended to the Ontology Model, the query performance significantly improves. The performance improves as the length of the query is increased.

Figure 6 shows significant performance improvement, when the new query words are extended to the ontology model. The performance enhancements range from 20% to 82%. Note that the improvement is more pronounced with the increase in the length of the query string. This is because the time to generate synonym sets for each client query word is a significant overhead that multiplies proportionately with the increase in the size of the query string. So, the incorporation of self-learning features like extension of the ontology models plays an important part in our query matching system.

4. Related Work

Eberhart at al. describe WSDF [24], a representative mechanism and a runtime system architecture, which allows a client to invoke a service based solely on ontology without prior knowledge of the API. This work overcomes the drawback of the approaches presented in OWL-S and BPEL4WS. WSDF provides semantic annotations to Web services allowing ad-hoc invocation of a service. Patil et al. have developed MWSAF, a Web service annotation framework [18] that performs both element and structural level matching for Web services. The element level matching is bound on a combination of Porter Stemmer algorithm for root word selection, WordNet dictionary for synonyms, abbreviation directory to handle acronyms, and NGram algorithm for linguistic similarity of the names of two concepts. Sycara et al. have developed one of the earliest ontology-based semantic matchmaking engines, MatchMaker [19], which uses capability-based semantic match and various IR-based filters. Another related effort is Racer [20], which focuses solely on service capability-based semantic matches for application in e-commerce systems. Syeda-Mahmood et al. [21] explore the use of domain-independent and domain-specific ontologies for finding matching service descriptions. Domain-independent relationships are derived using an English thesaurus after tokenization and part-of-speech tagging, while domain-specific ontological similarities are derived by inferring semantic annotations associated with Web service descriptions. A combination of the matches due to the two cues is done to determine an overall semantic similarity score. Our work extends the work by Syeda-Mahmood et al. [21], but dynamically learning from previous match making results, extending the ontological vocabulary, and applying the knowledge to subsequent queries Agarwal et al. propose a solution Synthy [22] for the composition of Web services using domain-dependent ontologies. The system provides semantic reasoning and planning but it does not include domain-independent cues such as thesaurus and text analysis techniques such as stop word filtering. Syeda-Mahmood et al. describe Minelink in [23], which uses bipartite graph for modeling Web service compositions and solves a maximum matching problem using domain-independent cues and text analysis techniques.

5. Conclusions and Future Work

We presented a system that matches user queries with operations in Web services. The system uses lexical analysis, domain-independent matching techniques, domain-specific ontologies and a set of specialized algorithms and optimizations to match simple free-form queries to WSDL operations. The system also provides a self-learning mechanism that utilizes the knowledge of previously made queries and enhances the efficiency of the system by a range of 20% - 82%. Our system provides the ease-of-use of popular Web search engines, enhanced with the ability to combine and retrieve information related to user queries.

In future work, we plan to enrich the vocabulary and ontology, and extend the number of domains in our experiments. We also plan to conduct a detailed accuracy study of our system.

6. References

[1] M. J. Duftler et al., "Web Services Invocation Framework (WSIF)" in *OOPSLA Workshop on Object Oriented Web Services, October 2001*

[2] R. van Engelen, "A Framework for service-oriented computing with C and C++ Web service components", ACM Transactions on Internet Technologies, 2007.

[3] Axis: "WebServices – Axis" Web Page. Available: http://ws.apache.org/axis/

[4] "Semantic Web" Web Page. Available: http://www.w3.org/2001/sw

[5] "OWL Web Ontology Language Overview" Web Page. Available: http://www.w3.org/TR/owl-features/

[6] G. A. Miller, "WordNet: A Lexical Database for the English Language" in *Comm. ACM 1983*

[7] Hai He et al., "An Automated Integrator of Web Search Interfaces for E-commerce" in *VLDB Journal, Vol.13, No.3, pp.256-273, September 2004*

[8] M. F. Porter, "An algorithm for suffix stripping", in *Program, 14(3) pp 130−137, 1980*

[9] V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions and Reversals" in *Soviet Physics Doklady, Vol. 10, p.707, 1966.*

[10] "JWNL 1.3" Web Page. Available: http://jwordnet.sourceforge.net/

[11] Charles P. Bourne, Donald F. Ford, "A Study of Methods for Systematically Abbreviating English Words and Names" in *Journal of the ACM Volume 8, Issue 4 (October 1961), Pages: 538 – 552.*

[12] "Metaphone" Philips L, "Hanging on the Metaphone." *Computer Language 1990, 7:39-43.*

[13] "Resource Description Framework (RDF)" Web Page. Available: http://www.w3.org/RDF/

[14] "RDF Vocabulary Description Language 1.0: RDF Schema" Web Page. Available: http://www.w3.org/TR/rdf-schema/

[15] "Jena - A Semantic Web Framework for Java" Web Page. Available: http://jena.sourceforge.net

[16] David Heckerman, Eric Horvitz, "Inferring Informational Goals from Free-Text Queries: A Bayesian Approach" in *Proceedings of the Conference on Uncertainty in Artificial Intelligence, July 1998*

[17] "Geoplaces.wsdl" Web Service. Available: http://www.codebump.com/services/placelookup.asmx?wsdl

[18] A. Patil et al., "METEOR-S Web Service Annotation Framework" in *Proc. WWW Conference, pp. 553-562, 2004*

[19] K. Sycara et al., "Dynamic service match making among agents in open information environments" in *Journal of the ACM SIGMOD Record, 1999*

[20] L. Li, I. Harrocks, "A Software Framework For Matchmaking Based on Semantic Web Technology" in *Proc. WWW Conference, 2003*

[21] Syeda-Mahmood et al., "Searching Service Repositories by Combining Semantic and Ontological Matching" in *Proc. of the IEEE International Conference on Web Services, 2005*

[22] V. Agarwal et al., "Synthy. A System for End to End Composition of Web Services" in *Journal of Web Semantics, Vol. 3, Issue 4, 2005*

[23] Syeda-Mahmood et al., "Minelink: Automatic Composition of Web Services through Schema Matching" *Poster paper WWW Conference, 2004*

[24] A. Eberhart, "Ad-hoc Invocation of Semantic Web Services" in *Proc. of the IEEE International Conference on Web Services, 2004*