Semi-Automatic Management of Web-Based Software Services

By: Michael Fitzgerald Nowlan

Advisor: M. Brian Blake

Senior Thesis
Department of Computer Science
Georgetown University
Washington, DC 20007
May, 2009
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents</td>
<td>2</td>
</tr>
<tr>
<td>Foreword</td>
<td>3</td>
</tr>
<tr>
<td>Abstract</td>
<td>4</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>5</td>
</tr>
<tr>
<td>2. Background on the SOC Domain and Related Work</td>
<td>9</td>
</tr>
<tr>
<td>2.1 Web Services and XML</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Related Work</td>
<td>10</td>
</tr>
<tr>
<td>3. SAMS Approach: Similarity Method</td>
<td>12</td>
</tr>
<tr>
<td>3.1 Initial Investigation</td>
<td>13</td>
</tr>
<tr>
<td>3.2 Developer Tendencies</td>
<td>13</td>
</tr>
<tr>
<td>3.3 Syntactical Similarity Methods</td>
<td>16</td>
</tr>
<tr>
<td>3.4 Evaluation of the Similarity Method</td>
<td>16</td>
</tr>
<tr>
<td>4. SAMS Approach: Component-Based Architecture</td>
<td>20</td>
</tr>
<tr>
<td>4.1 SAMS Architecture: Component View</td>
<td>21</td>
</tr>
<tr>
<td>4.2 Repository Reading Component</td>
<td>22</td>
</tr>
<tr>
<td>4.2.1 Defining Instrumentability</td>
<td>22</td>
</tr>
<tr>
<td>4.2.2 Assessing Instrumentability</td>
<td>23</td>
</tr>
<tr>
<td>4.3 Categorizing Component</td>
<td>25</td>
</tr>
<tr>
<td>4.3.1 Categorization On Pairing</td>
<td>26</td>
</tr>
<tr>
<td>4.3.2 Assessing the COP Method</td>
<td>26</td>
</tr>
<tr>
<td>4.4 Relation Identification Component</td>
<td>29</td>
</tr>
<tr>
<td>4.5 Implementation Details</td>
<td>31</td>
</tr>
<tr>
<td>5. Applications of the SAMS Approach: Evaluative Simulation and Case Studies</td>
<td>31</td>
</tr>
<tr>
<td>5.1 Web Service Recommendation</td>
<td>32</td>
</tr>
<tr>
<td>5.2 Intelligent Agent Collaboration</td>
<td>34</td>
</tr>
<tr>
<td>5.2.1 Defining Agent Behavior</td>
<td>36</td>
</tr>
<tr>
<td>5.2.2 Evaluating Agent Behavior</td>
<td>37</td>
</tr>
<tr>
<td>5.3 Web Service Mashup</td>
<td>39</td>
</tr>
<tr>
<td>6. Discussion and Future Work</td>
<td>44</td>
</tr>
<tr>
<td>7. Conclusion</td>
<td>45</td>
</tr>
<tr>
<td>8. Acknowledgements</td>
<td>46</td>
</tr>
<tr>
<td>9. References</td>
<td>46</td>
</tr>
</tbody>
</table>
Foreword

I would like to thank my parents for the support they have provided and the sacrifices they have made in order to give me so many opportunities. They have given me enough that I may do anything I like, but not enough that I may do nothing. I would like to thank Dr. M. Brian Blake for his valuable input, knowledge, willingness to help, professionalism and kindness throughout my undergraduate research work. He has been more than generous in his support and advice on topics ranging from computer science to life. Finally, the faculty and students in the Department of Computer Science at Georgetown University have my best wishes and sincerest thanks for a wonderful experience.
Abstract

Service-Oriented Computing (SOC) offers the promise of modular software, called services, produced and consumed by networked stakeholders, to be combined, ordered and executed to create higher-level functionality. The modularity fosters growth and adaptivity, which caters to the changing needs of its environment. Currently, there are problems in organizing these modular software components because of the distributed nature in which they are created. In this work, the new Semi-Automatic Management of Services (SAMS) architecture is introduced as an overlay to any entity's service execution protocol. I also introduce a syntactical similarity approach to identify relationships between the data provisions of the software services. In addition to evaluating the similarity approach on open services, the architecture was implemented for various SOC applications (i.e. recommendation, service mashup).
1. Introduction

Service-Oriented Computing (SOC) is a paradigm [27] in which software is easily and openly accessible via modules across a network. These software modules act as utilities for the users of the network by receiving some input data provisions and returning output data provisions. A vision for the SOC concept is that the Internet will act as a hub for these software modules, enabling users to receive information by way of executing multiple service queries. Service-Oriented Computing supports the notion that millions of services comprise both the body of knowledge and the actions that can be taken within a network. In recent years, many tasks or actions that were traditionally carried out by human agents, have been transformed into software services, which has helped to fuel the explosion of e-commerce on the Internet. The term Web Services is used to denote any software of this type that is made available over a network (or the Internet). Software of this nature abounds on the Internet from booking travel, to ordering food, to simple informational queries through search engines. Great promise lies in the concept of Web Services because they have the power to transform the Internet into a repository of relevant information for any individual user and places that information at the fingertips of a user. In the earliest years of web services, business transactions accounted for much of their utilization. Now, with the growth of Web 2.0 [25] and its applications, business services are not the only services being used. Informational services abound and make it such that every user is a consumer in the web services sense. In the happy case, everyone has their own unique Internet experience, populated with all the information and tools a user specifically wants, and driven by the underlying services that relay the information.

In order to bring about the picture of automatic service consumption described above, there must be a way for users to find and organize useful web services. Service discovery refers
to identifying services or a particular service that adequately fulfill(s) an informational request. In addition to service discovery, the discovered services must be organized in such a way that they are easily presented and executed. This can take many forms, but at the basic level, a user must be able to separate services based on similar content. Also, many researchers have attempted to compose services to form higher-order functionality. Service composition is a type of organization characterized by matching the output data provisions of one service with the input data for another service so that together, the chain of services provides a new software capability. Discovery, and particularly, composition, are two traditional issues that researchers have investigated because of the powerful ability that such tools would have in transforming information exchange on the Internet. Specifically, automatic methods for discovery and composition would aid immensely in this transformation because intelligent agents could use such tools to process information with more efficiency, while maintaining the intentions of a user. However, there are severe limitations to efficient and accurate autonomous methods for discovery and organization of services.

The primary problem with service discovery with random services over the network, is the lack of a definitive data library that disambiguates or maps data provisions used across multiple services. There are no automatic, machine-interpretable means for identifying services or their functionality. Some projects [14] have attempted to solve the problem by gathering and categorizing hundreds of services. While these repositories are effective at providing a central location for discovering services, they require human manipulation and traversal, and thus, they will always be limited by their requirement for human intervention. Recall that in a true SOC environment, the system should be able to identify and present services for consumption autonomously. The only human element occurs, if at all, at the end of the process, in deciding
whether or not to invoke the service. A repository that simply lists service offerings does not completely realize the vision of SOC. Other approaches [26][13] for organizing services use ontologies and semantics, but these approaches introduce further complications. Semantic techniques consist of creating classifications, relationships and an ordering of terms that can be referenced during parsing of a document in order to gain deeper knowledge of the underlying meaning of its strings. While these techniques ultimately provide the most accurate solution, they lack widespread use. In the current SOC environment, syntactical methods, that is, methods that deal with string manipulations of the actual characters present in a term, are much more practical because they do not require adherence to a common ontology or classification system [38]. With the ever-increasing size and breadth of the Internet, the entry requirements for a semantic system may cause it to be infeasible. Therefore, any system for managing web services must be two-fold. It must be "in-place", that is, adaptable enough to handle a changing environment of service offerings without requiring developers to adhere to a common classification standard or ontology. Furthermore, it must be scalable and autonomous so that it fosters machine-interpretable methods for managing services with minimal human input. Any system with these two components will be able to grow and adapt in parallel with the Internet and it will increase efficiency by eliminating superfluous human interaction during stages of the service consumption lifecycle that can be performed without human input.

At the lowest level, services transmit their data in named messages passed via the Simple Object Access Protocol (SOAP) [32]. Although the message names are merely strings, in some cases, they are descriptive enough to identify the type of data being transmitted. There is a fundamental need to compare and discern meaning from these descriptive strings. This task of organizing, comparing and defining services based on their "in-place" data attributes is the
primary goal of this work. Formally, this research seeks to address the following questions for the SOC domain:

- Is there an efficient method for discerning meaning from the in-place data within a service specification?
- Can an in-place solution perform effectively in a real-time environment?
- Can this solution be realized in multiple applications within the SOC domain?

This approach, entitled Semi-Automatic Management of Services (SAMS) involves using syntactical means for discerning the content a web service manipulates. The implementation of the approach is in-place and rooted in the current (at the time of execution) state of the service domain, so that it can follow any changes in the creation and modification of services. The approach is customized to fulfill requirements in the area of service discovery and similarity, as well as other applications.

This paper proceeds in the following order. The next section briefly introduces the technology of the SOC domain and discusses the related work in the areas of web service management, composition and discovery. Furthermore, other syntactical approaches towards service organization and recommendation are discussed. The third section explains the syntactical approach for comparing and understanding services using their in-place data attributes. This syntactical approach is exploited and extended in the fourth section, which presents and describes the SAMS architecture for service lifecycle management. The fifth section evaluates an implementation of the SAMS approach on its effectiveness for various applications within the SOC domain. The last section is a discussion on the contributions of this research and future work involving this approach.
2. Background on the SOC Domain and Related Work

There are a large number of related projects in the area of web service management. These projects comprise a thriving research area concerned with traditional service problems, such as service discovery and composition. Before discussing these projects, it is necessary to understand two technologies essential to Service-Oriented Computing: Web Services and the eXtensible Markup Language (XML). A Web Service Description Language file (WSDL) [36] is an XML file specification for defining web services.

2.1 Web Services and XML

The WSDL file describes the nature of the service and the method for invoking the service. In many cases, a WSDL file can be automatically generated by the software that was used to create the service itself. The WSDL file is a specification document in that it explains how to explicitly interact with the service (i.e. port number, Uniform Resource Locator, data format, etc.). Thus, a WSDL document is not a simple, textual description, but rather a technical specification for the protocols used in the service's execution. Although a WSDL file usually will not provide long sections of comments that describe the content the service manipulates in normal human language, it does contain the variable names, or descriptive strings, that the developer of the service used to represent the data provisions. These descriptive strings are called parts, because they appear in an xml "part" element. Each part has a "name" attribute, which is referred to as a part name. A web service is comprised of various operations, which actually receive the input data and return the output data via an input message and an output message, respectively. Therefore, in many cases, the part names in the WSDL document describe and indicate the type of content that the service manipulates or transports. Furthermore,
a web service could be represented as a bag of words containing all the descriptive information of a service, pulled from a WSDL file. These descriptive strings include the file name, operation names, port names, message names and part names. Taken as a set, this collection of strings represents a service in much of this work's experimentation. The set of descriptive strings for any service can be represented as:

\[ S_d = \{S_n \cup \{S_p\} \cup \{O_n\} \cup \{M_n\} \cup \{P_n\}\} \]

where:

\[ S_n \equiv \text{the service name} \]

\[ S_p \equiv \text{the set of service port names} \]

\[ O_n \equiv \text{the set of service operation names} \]

\[ M_n \equiv \text{the set of service message names} \]

\[ P_n \equiv \text{the set of service part names} \]

### 2.2 Related Work

Two traditional research problems for web services are the notions of web service discovery and composition. Discovery is simply the task of identifying web services of a specific nature; while composition attempts to order services such that they can be executed sequentially to provide a complete chain of new functionality. Techniques for the discovery and composition of web services are the target of many related projects for Service-Oriented Computing. Srivastava and Koehler [18] and Rao [29] detailed the progress that has been made in the field of service composition and detailed the two competing approaches, semantic and
syntactic techniques. Semantic techniques require the use of ontologies and multi-level classification systems, such as OWL-S [26] and DAML [13]. While accurate search and discovery (i.e. with a high level of confidence) may require semantic approaches with technologies such as RDF, OWL-S, and WSDL-S, currently these technologies are not widely used in practice. Woole [14] is a project that allows users to create a query to find relevant web services. This approach is similar to the search that takes place in this work; however, in this work, WSDL files and SOAP messages are used to create customized queries that traverse through the service-oriented architecture. This work is more closely related to syntactical projects. In previous work, some approaches have shown that text manipulation can help to identify terms with similar meanings [7][23]. Other syntactical projects include, Rocco [30], who uses rigorous string manipulation software to help equate web services messages and Pu [28], who uses an XML type-oriented rule-based approach. Lastly, collaborative filtering [3] is another method for augmenting service discovery and uses collaborating agents to identify services. This research differs from the related projects above because it uses natural language tendencies of developers when creating and naming the services.

Two other intersecting threads within this work are web service recommendation and intelligent agent communication. Recommender systems have traditionally been associated with the electronic commerce domain [5][9]. Furthermore, several papers addressing recommender systems for web services either incorporate web services in their implementation [4] or suggest web services at the human-user level [21]. In most cases, a recommender system uses the input of a consumer or user and finds relevant web services. The work of this paper uses descriptions of a user's operational domain as the search input and returns relevant services based on similarity between the input and the service descriptions. When combined with intelligent agents
the knowledge for recommendation may increase significantly. It is also of note that for recommendation, composition and discovery, Universal Description Discovery and Integration (UDDI) [34] and the Business Process Execution Language (BPEL) [19] have shown to be effective for organizing services in strictly defined scenarios. Both of these technologies could be combined with the work presented in this paper to provide a more robust system.

The final important application of this work within the SOC domain is towards the problem of web service mashup. A web service mashup is the simultaneous execution of two or more services that are related in some aspect of the content that they manipulate, but have different functionality. Web service mashup is a relatively newer application of the larger topic of data integration. Although the area of data integration has had a longstanding background, the usage of these techniques to accomplish mashups has only just recently started to be addressed. A majority of the work in this area addresses the tools and environments that support the visualizations that presents mashup results [17]. Other projects describe enabling techniques for preparing services for mashup [20][31]. These are also projects that investigate the policy for protecting data in mashup environments [40] and instituting enterprise policy for modernizing systems using the information resulting from successful mashups [10]. This paper presents one of the first approaches for predicting potential mashups accurately. In the work of this paper, the management of web service data facilitates the discovery of potential web service mashups, but stops short of actually executing and composing the mashups.

3. SAMS Approach: Similarity Method

This section presents the syntactical similarity method devised from the descriptive strings pulled from service specifications. This section also presents results for the effectiveness
of this approach in determining string similarity.

### 3.1 Initial Investigation

In order to understand the current practice in the creation and offering of web services, it is necessary to observe the nature of developers and their software. By scrutinizing the ways in which services are created and presented, knowledge about the best ways in which to compare and match services becomes apparent. A goal of this work was to identify a syntactical method for comparing web services so that they might be organized and managed quickly. In order to do this, nearly 550 web service description files were downloaded from various Internet repositories [39][37]. The size of the repository and the duplicate files found across different Internet repositories lead the researcher to believe that the repository is one of the largest repositories of WSDL files that represent real, functional, services. This repository is used as the dataset for all of the experimentation of this work.

**Table 1. Statistical breakdown of the web service repository**

<table>
<thead>
<tr>
<th>Number of WSDL files</th>
<th>545</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of descriptive strings</td>
<td>14919</td>
</tr>
<tr>
<td>Number of unique descriptive strings</td>
<td>3966</td>
</tr>
</tbody>
</table>

### 3.2 Developer Tendencies

By representing a service as a collection of strings, it is possible to compare two sets of strings for similarity. In this comparison, service discovery is reduced to the task of determining similarity between a service's descriptors and a target set of strings. A major hypothesis of this work is that two strings may differ in characters slightly, but can hold the same meaning. If true, this suggests that for any set of strings (representing a service), there can exist slight
modifications of this set that retain the set's inherent meaning. The only way to verify this in the context of the SOC domain is to verify sets of descriptive strings for services that are known to be similar, and see if the sets show similarity. Initial results showed that there were indeed cases where similar services used similar, but not identical descriptive strings. Examining further showed that, in general, developers of services, without collaboration, exhibited certain tendencies in naming the data provisions of their services (ie. the service's descriptors). The most prevalent tendencies are as follows:

1. Some developers use part names that are very specific or very general, such as "getCityStateLocationResponse" or "id"
2. Developers use subsets of certain terms where longer versions are similar, such as "zip" and "zipCode"
3. Developers use different endings for the same root word, such as "stockSymbol" and "stockName"
4. Developers change descriptive strings for uniqueness, such as "city" and "city1"

These tendencies provided useful insight into the habits of developers for web services because this knowledge could be used to syntactically manipulate the descriptive strings. The first tendency suggests that strings that are too long or too short provide too much or too little information to be useful. This can be handled programmatically quite simply by ruling out strings that are not within a given length. Tendency two suggests that two strings may be similar when one is a subset of the other. This is handled via subsumption relationship string checks. Tendencies 3 and 4 require slightly more complex string manipulations in the form of the Letter-Pairing Similarity and Levenshtein Distance algorithms. These two algorithms specialize in identifying similarity between two strings. The Letter-Pairing algorithm breaks a string into
character pairings and calculates the number of shared letter pairings between two strings as a percentage of total string pairs. The Levenshtein Distance calculates the smallest number of deletions, insertions or modifications needed to transform a string, s, into a target string, t. In the case of both algorithms, there must be a cutoff for determining when two strings are similar. For instance, Letter-Pairing returns the percentage of the pairs between two strings that are the same. Levenshtein Distance returns the number of modifications necessary to transform the string into the target. Determining the proper cutoffs for these two methods is presented in detail in other work [6][24], but this work uses the general cutoffs of 45% and (2/3 * stringLength - 2) for Letter-Pairing and Levenshtein Distance, respectively. Figure 1 presents these algorithms.

**Figure 1.** Explanation of Levenshtein Distance and Letter-Pairing Algorithm (left) TSM-LP psuedocode (right)
3.3 Syntactical Similarity Methods

These different syntactical methods of string manipulation account for the four observed tendencies of developers and form a similarity algorithm rooted in actual web service creation techniques. This similarity method is called *Tendency-based Syntactical Matching - Levenshtein Pairing* (TSM-LP). In addition to TSM-LP, TSM-L and TSM-P versions of the algorithm were created with the Letter-Pairing and Levenshtein methods absent, respectively. In many of the experiments in this work, the enhanced syntactical methods are compared against simple equating and subsumption-only matching approaches. These two approaches generally yield high precision, but lack the recall of the enhanced methods. Figure 1 defines the TSM-LP approach and its use of several different algorithms that target the observed tendencies. The four stages of the algorithm specifically target the developers' tendencies.

3.4 Evaluation of the Similarity Method

As mentioned, in the experiments, simple subsumption relationships were also evaluated alongside the previous three methods. As previously mentioned, this research work is concerned with a fast, accurate method for discovery and management of web services. In order to evaluate the similarity method(s), accuracy and performance were measured to determine the most viable candidate algorithm for inclusion in further research work. As such, the accuracy (precision) of the various methods was tested and recorded, as well as the time for each method. The timing measurement was a comparative one, rather than a running time analysis. As services are reducable to a collection of descriptive strings, this experiment analyzed the ability of the different methods to identify similar services based on their descriptive strings. In comparison to the related work of Wang and Stroulia [35], which operates on smaller repositories with 50%
precision, TSM-L performs comparably or better. Additionally, 6 milliseconds per service comparison extrapolates to 1000 service comparisons in under 6 seconds on a 1.6GHz single processor Pentium M 730. This measurement yields a reasonable response time (less than 6 seconds) for any search within a repository of up to 1000 services using TSM-LP. Each service comparison consists of applying the TSM-LP algorithm to the two sets of descriptive strings and calculating a percentage of strings that are similar between the two services.

An effective syntactical string comparison method is the cornerstone to the tools described in this work within the SOC domain. The syntactical methods are tailored to the SOC domain by taking into account SOC developer tendencies in naming their services. Furthermore, these methods are fast, such that they are much more practical than semantic methods. Four distinct algorithms are evaluated in Figure 2 on their ability to identify string similarity in strings pulled from web services. Because these methods stem from the nature of web services creation, this experiment identifies the best method for syntactical similarity for the SOC domain. In each method, exact string equivalence is the starting point. However, simple string equivalence misses many valuable similar strings and so these syntactical methods were made to provide similar matches beyond exact equivalence. In this manner, the TSM-L, TSM-P and TSM-LP (TSM*) methods are called enhanced methods, because they provide more matches than simple equating. Figures 2 and 3 show the results of evaluating various syntactical methods for precision and performance.
In the experiments, ambiguous strings were eliminated from the set of descriptive strings for a service. Ambiguous strings are strings such as "return", "result", "response", "body" and "string". These terms are programming terms included in WSDL documents and sometimes are included in a service's descriptive strings collection. In all comparisons in this work, these ambiguous terms were ignored. In analyzing the results above, TSM-P and TSM-LP both employ the use of the Letter-Pairing algorithm, which specializes in finding similar substrings between strings, and these two methods were noticeably lower in precision than TSM-L. This is because in finding a shared substring between strings, the methods often identified an ambiguous or unimportant shared substring. For example, "candidates" and "endDate" were returned as similar because of the shared substrings "nd" and "date".

Figure 4 shows the estimated number of good matches combined with the estimated number of incorrect similar matches by method. Notice how the number of correct matches gained with TSM-P and TSM-LP is much smaller in comparison to the number of incorrect
matches they identify. In light of these results, the rest of the experiments in this work exclusively employ the TSM-L algorithm, as it shows an ability to return a large number of "similar" matches beyond simple equating, yet maintains a high precision: above 80%. It is believed that TSM-P and TSM-LP are useful syntactical methods and that greater research can be conducted to determine their applicability in this domain.

**Figure 4.** Estimated number of overall positive and negative matches

The usefulness of TSM-L in this task suggested that it would augment software tools for managing web services in the SOC environment. The plethora of work in the area of service composition signals that it is a popular research problem in the community, whereas, service discovery and recommendation are somewhat less researched. This work focuses on the latter two topics. As mentioned above, the experiments for this work were conducted on a dataset of services pulled openly from the Internet. As such, the recall metric is extremely difficult to measure because all the possible matches that should be returned could only be identified by manual inspection of the repository of 545 services. Where noted in this work, a subset of a set
of results is evaluated for recall where applicable.

4. SAMS Approach: Component-Based Architecture

The second innovation of this research work is the SAMS component-based architecture for organizing web services. This approach implements the most effective uses for the similarity method proposed in Section 3. This system is an end-to-end solution for managing web services from the point of discovery to the point of execution, with features at both ends for updating and learning. The architecture is described in this section by the components that comprise its functionality. These components were then applied to different tasks within the SOC domain.

SAMS adheres to the principles outlined at the beginning of this work of "in-place" and "adaptive". As the nature of the domain changes, or the use of the system is modified, the system keeps pace because its components for mashup and recommendation are rooted in an effective observation of the domain itself. The SAMS approach is distinct because of its use of syntactical methods for similarity and categorization and because of its adaptability to automatic or intelligent systems. The SAMS architecture is comprised of three major components. The first component is a web service repository reader. This feature of the system analyzes a repository of services and extracts descriptive strings for each service, organizes all descriptive strings in the repository and compares services based on their descriptive strings. The second component is the automatic categorization component, which divides the repository into a specific or open number of categories. This component further analyzes each category by the metrics of uniqueness and service similarity. The final component of the SAMS system is the ability to identify connections between services based on shared or complementary data. The relation identification component presents a novel method for identifying complementary
services among a collection of services that share high similarity. This novelty is described in
detail in Section 5.3. The SAMS system uses the TSM-L enhanced syntactical approach for its
service comparisons, but the system supports the use of any enhanced syntactical method such as
TSM-P or TSM-LP.

4.1 SAMS Architecture: Component View

This section presents a high-level picture of the SAMS approach and how it provides
different components for managing web service data. In Figure 5, services are gathered from
Intranet sources, such as internal company data services, and Internet sources, such as publicly
available services for free or with subscription. This SAMS architecture gathers the WSDL
documents that describe the services collected. With this repository of data, the system organizes
the services in different ways depending on the management view desired. As new services are
introduced or offered to the system, the changes ripple down the chain, manifesting themselves
in an altered management view of the service data. The management views of the data enable a
controller to gain greater insight into the nature of the services and the connections between them
via shared attributes.
4.2 Repository Reading Component

This component focuses on extracting knowledge from a repository of services and it provides this data to the later steps of the SAMS approach. For each service in a repository, the reader extracts the service name, port names, operation names, message names and part names. In previous experiments, services were reduced to a set of descriptive strings. This component carries out this reduction. During the processing of each service, the descriptors are recorded and sorted by number of appearances and number of similar strings. Useful knowledge about the overall nature of a repository is gained from examining the most popular strings and the strings that have the highest number of similar terms.

4.2.1 Defining Instrumentability

One measurement of the connection or commonality within a repository of services is the *instrumentability*. Instrumentability describes the likelihood that any service, operation or
individual string shares similarity with other elements of the repository. For instance, suppose the descriptor "city" appears in every service description (very unlikely) and "arg2" appears in just one description. The instrumentability for the term "city" is much higher than that of "arg2" because the term "city" links services together through its appearance in both, it can be used as an instrument to create inter-service connections. Identifying the instrumentability of a collection of services effectively describes how easily the repository lends itself to achieving higher-order functionality, which is the promise of the SOC concept. After reducing services into their sets of descriptive information, the Repository Reading component parses all unique strings, conducts service similarity testing and measures the instrumentability of the repository at the service, operation and string descriptor levels. The service similarity component is conducted using TSM-L. The experiments described below help to outline the process of determining instrumentability.

4.2.2 Assessing Instrumentability

Instrumentability describes how similar an element is to all other elements of its type in a repository. This applies to service descriptors, or sets of descriptive strings. These sets vary in size depending on what element of a web service they are representing. This section describes the experiments used to measure instrumentability within the repository of 545 web services. In each of these three experiments, the similarity matching approach TSM-L was used. The first experiment measures instrumentability between services. In this experiment, the sets of strings contained all part names, message names, operation names within the service, as well as the service name. In this primary experiment (Figure 6), for each service it was determined how many other services shared at least one similar string. In the second experiment (Figure 7), instrumentability between operations was measured. Just as services can be reduced to a
collection of descriptive strings, an operation can be reduced to a set of strings containing the operation name, the message names and the part names that it contains. With these sets, an operation was compared to every other operation to see the number of operations with which the original operation shared a descriptive string. The last experiment (Figure 8) shares a similar procedure as the first two, as it measures instrumentability between part names. In this test, each set of descriptive strings contained only one string: the part name itself. These experiments show the spread of instrumentability throughout the repository among the entities of services, operations and part names (descriptive terms).

Figures 6, 7 and 8 show the instrumentability within the repository of the three different web service elements. The elements are ordered left to right from most instrumentable to least instrumentable. Notice in the latter two cases, the most instrumentable example is noticeably more instrumentable than the second most instrumentable example. Examining these results shows that services overall are more instrumentable than part names or operations. The services have the largest set of descriptive strings of the three different elements. Operations showed a generally linear decline in instrumentability, while part names were the least instrumentable and experienced a noticeable drop in instrumentability after the first quarter of examples.
4.3 Categorizing Component

This component of the SAMS approach is responsible for separating the repository into categories of similar services. Creating categories within a repository is useful for targeting
certain types of services during discovery or mashup. It is also useful for identifying which
types of services are most common, or which types of services are most descriptive. In many
cases, a developer will create services in generally the same area, such as finance services [33].
A strength of categorization is that a statistical representation or description can be generated for
each category. As such, if a category grows or shrinks in size, only the affected category's
statistics need to be recalculated.

4.3.1 Categorization On Pairing

The SAMS approach implements categorization using TSM-L (or any enhanced
syntactical method) and a novel categorization approach named *Categorization-on-Pairing*
(COP). The COP method compares each service to every other service. For each comparison, a
pairing is created with links to the two services and a similarity value which represents the
percentage of how similar the two services are to each other. These pairings are then sorted in
descending order from most similar service pairing to least similar pairing. Then, for the single
pairing with the greatest similarity of all pairs, a new category is created. The sorted list of
pairings is then traversed. If both services in a pairing are already in a category, no action is
taken. If just one service is categorized, then the uncategorized service is added to the same
category as the categorized service. If neither service is categorized, a new category is created.
In the case where a specific number of categories is desired and this number of categories has
already been created, then a pair in which neither service is already categorized is handled as
follows: each service is added to the category with which it has the greatest similarity. The
experiment described below further highlights the COP method.

4.3.2 Assessing the COP Method

This experiment compares the automatic categorization of a repository using the COP
method with manual classification in which each service is placed into a category after inspection of its content and functionality. In this experiment, 9 categories were used in accordance with the 9 categories used by Halevy, et. al. in the Woogle project [14]. In order to begin, the repository was first manually stratified into 9 categories. Then, the COP method was used to create an "open" number of categories, which means that there was no limit to the number of categories that could be created. The first version of this experiment compares the COP method of categorization to the manual method outlined above. Table 2 shows some attributes of the categories created by the different methods. The "% Unique Descriptors" corresponds to the average percentage of all the strings in each category that are unique. A higher percentage means that the category’s descriptive strings are largely non-repeating, whereas a lower percentage means that the category is comprised of many repeating descriptors. The "# of Services" is the average number of services in a category for each method. Lastly, the "Service Similarity" is the average similarity score between services in a category for each method.

Table 2. Statistical breakdown of Manual vs. Automatic Categorization

<table>
<thead>
<tr>
<th>Categorization Type</th>
<th>Manual</th>
<th>Automatic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Categories</td>
<td>9</td>
<td>108</td>
</tr>
<tr>
<td>% Unique Descriptors</td>
<td>0.51</td>
<td>0.62</td>
</tr>
<tr>
<td>Number of Services</td>
<td>61</td>
<td>5</td>
</tr>
<tr>
<td>Service Similarity</td>
<td>0.39</td>
<td>0.77</td>
</tr>
</tbody>
</table>

The COP method created many more categories, but each category was comprised of more similar services as shown by the average Service Similarity, which is nearly double for Automatic. This is promising as the percentage of unique descriptors for automatic was only
slightly higher than manual. As a result, the COP method is useful for creating many tightly
defined categories. This may not be optimal, however, if the domain requires a small number of
broadly defined categories. To accommodate this, the COP algorithm was modified to enable the
creation of a specific number of predefined categories. The creation of exact categories was
tested by varying the number of categories from five to 100. The corresponding averages are
listed in Figure 9.

In measuring our categories, we have used the metrics of service similarity within a
category and the percentage of all the string descriptors that are unique. We feel these are two
measurements that describe a category's homogeneity or heterogeneity very well. A higher
average service similarity shows that a category is comprised of services that are similar.
Conversely, a higher percentage of unique parts means that a category contains many different
descriptive strings and it is therefore less strictly defined. It is expected that as the service
similarity increases, the percentage unique will decrease, and vice versa. Figure 9 shows this to
be untrue using the COP method. In each experimental test run, we limited the number of
categories to an exact integer and then measured the characteristics of those categories. It is
interesting to note that the rate at which the average service similarity increased was greater than
the rate of unique descriptive strings. This suggests perhaps that the repository is comprised of
services that are more similar than dissimilar. The COP approach created 5 categories in 43
seconds and created 100 categories in 14 seconds, which is reasonable for a bootstrapping
measurement. Modifications to the repository can be monitored in real-time and handled on a
service by service basis in under 1 second.

In the original experiment, the COP method dynamically created 108 categories, while
the manual categorization produced 9. The most interesting result from the second version of
this experiment is that when exactly 9 categories were requested, the COP method returned 9 categories with an average service similarity of 40.3% and a unique strings percentage of 50.1%. In the manual categorization of the services into 9 categories, the average service similarity was 38.7% and the average unique strings percentage was 51.3%. This suggests that the COP method actually performed better in the task of creating nine categories. Furthermore, the approach is much more practical than manual categorization for use within dynamic or agent-based systems. Using the COP method to categorize a repository either freely or into a certain number of categories enables an agent or system to dynamically become aware of the nature of a repository. It can further aid in the discovery of pertinent mashups, which is the third component of the SAMS architecture.

Figure 9. Service Similarity and Unique String Percentage metrics

4.4 Relation Identification Component

The third component of the SAMS architecture is the ability to identify potentially related
web services. This might be extended to allow for web service mashup recommendation, which is described in Section 5.3. This component seeks to identify links between two services by finding shared input or output data provisions. In this way, this component creates a graph of services connected by their similarity. This can be presented as a more complete operational picture around some attribute, object or characteristic. Figure 10 demonstrates how related services are identified using the similarity method to find similar or equal part names.

![Diagram explaining how complementary services are identified](image)

**Figure 10.** Diagram explaining how complementary services are identified

Using this definition for identifying connected services enables the problem to be seen in terms similar to that of other web service tasks: reducing a service to a set of descriptive strings. Any service is reduced to its descriptive parts and compared to other services to identify those services that share a descriptive string. The number of shared descriptors required to declare a relation can be varied to create connections between services that are more similar or less similar.
4.5 Implementation Details

This architecture was implemented using Java. The majority of the experimentation work was performed on a 1.6GHz single processor Pentium M 730 Averatec Laptop. The system contained roughly 25 classes spread among 4 packages. It consists of roughly 3500 lines of code.

5. Applications of the SAMS Approach: Evaluative Simulation and Case Studies

This section evaluates the SAMS approach and its components via the implementation of exemplary SOC applications. In order to aid in the management of web services, these tools are required to be fast and automated, adhering to the principles in Section 1. The different tools target specific tasks often performed when dealing with web services. Web service recommendation attempts to present a user with service offerings that fulfill some need of the user. In some cases, monitoring of the user identifies the domain or need. In other cases, a user actively submits a description of a desired service and the recommendation system identifies a service that fulfills that need. The second software application devised from the SAMS architecture is a communication protocol for intelligent agents that collaborate for web services management. This work specifies how agents might communicate when representing an entity that uses web services. Lastly, this section presents a web service mashup tool that extends the relation identification component in SAMS.
5.1 Web Service Recommendation

Web service recommendation attempts to present a user or intelligent agent with services that are relevant to the current working domain. Recommendation has been researched in different ways. Recommender systems initially were related to the e-commerce domain [9]. In later work, the SAGE project attempted to present web services on an individual basis by monitoring a user's activities [5]. The research in this work is more closely aligned with the latter approach. In this recommendation scheme, monitoring of a user's actions is simulated and used as the input for the similarity method. The similarity method extracts important strings from the textual input and creates a set of descriptive strings. Then, using the syntactical similarity method, TSM-L, comparisons are made between each candidate service and the set of descriptors. The service with the greatest similarity to the descriptors is recommended as a potential service for user consumption. Formally, the target set of descriptors is composed of strings related to some domain, and can be expressed as: $S_t = \{s_1, s_2, \ldots, s_n\}$. Recall from Section 2.1, that a service can be expressed as a set of descriptors, $S_d$. The recommendation system then identifies the intersection between two sets of descriptive strings: one set describing the operational domain, and one set describing a service. If the intersection is sufficiently large, the service is deemed to be "similar" and relevant to the operational domain.

In order to test this method for web service recommendation, five html files representing travel, currency conversions, online bookselling, finance and sports were saved from the Internet. These pages simulate the data that monitoring software might pull from a user's daily activities. The experiment tested whether the similarity method described above would recommend relevant web service operations given the content from the various files. In each case, the html file was transformed into a set of descriptive strings and compared against the repository of
services. Figure 11 depicts the web service recommendation scenario for a finance html webpage.

![Web Service Recommendation Scenario](image)

**Figure 11.** Example web service recommendation scenario

In the area of web service recommendation, it is likely that only a few services should be returned. This work does not specify the ideal number of web services returned, nor can it be said whether there ought to be a point below which no service can be deemed a recommendation. Rather than examine the entire list of recommended services, the top 3 services that showed the most similarity to the target set were examined manually to verify the practicality of this approach. The idea of this experiment is that regardless of how similar the services are to the target, a user is likely to want to see the MOST similar service to its operational domain. The results below list actual operation names returned for the individual files.

In Table 3, TSM-L showed the ability to identify services that are similar to a target set of strings that describes an operational domain. For instance, if the operational domain is finance, TSM-L recommended services that related to finance, such as market and exchange status, as well as a mortgage index service.
Table 3. Recommended services based on input file

<table>
<thead>
<tr>
<th>Type of File</th>
<th>Operation Name</th>
<th>Ranking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Itinerary generated from Travel website</td>
<td>GetStations</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>IsValidExchange</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>IsExchangeOpen</td>
<td>3</td>
</tr>
<tr>
<td>Currency conversions webpage</td>
<td>GetSearchTerms</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>NumberToDollars</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Search</td>
<td>3</td>
</tr>
<tr>
<td>Random book search from online bookseller</td>
<td>ListBooks</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>BooksInfo</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>WishlistSearchRequest</td>
<td>3</td>
</tr>
<tr>
<td>Finance homepage on Yahoo.com</td>
<td>IsValidExchange</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>GetCurrentMortgageIndex</td>
<td>2</td>
</tr>
<tr>
<td>Sports homepage on msn.com</td>
<td>GetSportNews</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>WorldCupFootball</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>GetBriefings</td>
<td>3</td>
</tr>
</tbody>
</table>

5.2 Intelligent Agent Collaboration

This section presents a protocol of agent communication for web services management. This work assumes that an intelligent software agent facilitates the service consumption within a network. Given many different networks that are connected (i.e. corporate networks across the Internet) and that each network may have an intelligent agent that governs its use of web services, a protocol that defines how agents share information about service execution, discovery and storage would foster greater efficiency and overall robustness. An example of this scenario might be a partnership between two businesses in which they agree to share information regarding their service consumption and organization. Similarly, a large business with multiple divisions that exhibit different service behavior, might wish to foster information exchange about services between the divisions. This work presents a first look at how agents would carry out such information exchanges.
This work occurs within a federation of intelligent agents that work together through communication to incorporate the web services of-best-fit into each of their business processes. This work focuses mostly on the agent “society”, or communication between agents, and uses this as the catalyst for all agent actions and functions. By allowing agents access to each other’s knowledge base, incorrect decisions are limited, and correct decisions are more quickly executed. Although the need for widely applied collaboration between agents of a business domain is visible, the obstacles to fostering such an atmosphere remain numerous. The first problem is that multiple businesses do not operate within the same firewalls or business architectures. Also, with no set language or communication protocol established, businesses would need to proactively set down rules for inter-communication. Although related work shows that agents have the capabilities of proactively discovering web services [5], similar but unconnected businesses will not often take the effort to establish formal lines of communication. Finally, cross-domain knowledge can be beneficial for multi-faceted businesses, such as travel agencies.

In order for an agent to share web service information, the agent must have certain abilities with regards to service similarity comparison and categorization. In this federation of agents, each agent uses TSM-L to compare and recommend services. Apart from querying Universal Description, Discovery and Integration (UDDI) [34] registries, the agents also may also offer and store services locally. The services that an agent uses both over the Internet and locally are categorized using a categorization technique called Categorization on Pairing (COP). An agent will keep track of the service similarity within each category for determining the homogeneity of a collection of services. With these internal capabilities, agents will send messages to other agents within the federation to share service information. Such shared
information would include, services recommended given a particular target set of descriptors, the number and size of local categories, the service similarity within categories, the percentage of recommendations that result in executions and the services that are recommended most frequently. In order for agents to share this information, there must be established messaging protocols, or formats, that agents adhere to when creating their correspondence. This work outlines four messaging protocols for service collaboration, each of which extends the Association for Computing Machinery (ACM) agent messaging protocols defined in the FIPA Communicative Act Library [15].

5.2.1 Defining Agent Behavior

Communication makes the tasks of web service discovery, management and recommendation a collective process, rather than an independent action. (1) Web Service Recommendation is the primary motivation for agent communication. As agents create a recommendation of services from their own local repository, other agents perform the same search on their own repository and return to the original agent a set of services that they would recommend, given a target set of descriptors. Using agents to do these actions is highly beneficial because it automates functions and increases scalability. Agents group the services in their local repository into categories of similar services. It is likely that over time, one agent’s local categories will resemble another agent’s, and an alignment of these categories makes recommendation easier so that agents can search within roughly similar categories across their separate domains. This is referred to as (2) Category Alignment. A third reason for communication is linked to the second task and is collaboration for category self-similarity, or (3) Category Analysis. After categories have been aligned, as in task (2), it is also beneficial to determine the uniqueness of the services in two equivalent categories so that the agent can
accurately judge the strength of relevancy matches. The last reason for communication is a (4) Call for Services. One agent sends this call out to multiple agents, or one agent in particular, asking for web services to populate its own local repository. Also, newly created agents use this communication to create their local repositories from scratch.

![Diagram of intelligent agent messaging protocols](Image)

**Figure 12.** Intelligent agent messaging protocols (1),(2),(4),(3) clockwise from top left

### 5.2.2 Evaluating Agent Behavior

The protocols stem from the ACL communication standard and attempt to provide a formal method for agent collaboration for service information sharing. In order to determine an estimated communication response time, we investigated related work containing performance measurements of tools associated with agent communication. Gigaspaces [16] is a collaboration framework typically used for interprocess (i.e. inter-agent) communication. Although [2] shows that Gigaspaces performs under 4 ms (irrespective of number of entries) when reading the shared
space, it does not address the interprocess messaging. Both Cortese et. al. [12] and Ahuja et.al. [1] agree that the interprocess messaging overhead is about 8-10ms per concurrent process (agents) considering a reasonable network infrastructure and hardware. The values given in Cortese et. al. is based on JADE, Ahuja et. al. based their measures on GigaSpaces and CORBA [11].

After extrapolating data size and number of round-trips per recommendation, the response time per recommendation is about 5 times the number of concurrent agents for less than 100 concurrent agents and 4 times the number of agents greater than 100 concurrent agents. However, if executing other protocols (i.e. the service sharing protocol, category analysis protocol, and category alignment protocols) for each recommendation then the response time is tripled. Figure 13 shows the estimated communication/messaging time as it relates to concurrent agents. Other measures show the performance impact of the other services when they are executed for each recommendation, 50% of the time, and 25% of the time.

![Figure 13. Estimated messaging time for recommendation scenario](image)

Figure 14 expresses the total estimated recommendation time which includes the
calculation the recommendation score. A tradeoff for system performance and maintaining current information is updating other services 25% of the time. Using this level of service, the recommendation response time is about 6 seconds for 250 collaborating agents. This result would suggest that the framework should optimize performance by limiting agent collaboration to the top 250 most relevant agents within the federation. This is believed to be a reasonable number to assure the fidelity of the recommendations.

![Figure 14. Total estimated recommendation response time](image)

### 5.3 Web Service Mashup

A mashup entails the simultaneous execution of two or more services that are related in the content they manipulate, but are of different functionality. A service mashup is the simultaneous execution of two or more services to create an integrated tool that provides a more complete description about some object or characteristic. For example, web services that provide mapping capability can be integrated with capabilities from the United Parcel Service (UPS) to understand the path of packages that get lost in the mail. This integration of web services outputs is the general idea behind service mashup. In this example, the outputs from the
two services are of a different nature, but they share the same location attribute. Much of the current work involves tools and techniques for integrating web services information and furthermore the visualization of that information. This work focuses on the discovery of candidate web services prior to the actual act of mashing them up (i.e. service mashup discovery) [8].

The technology for integrating service outputs (i.e. the fundamental requirement for service mashup) lies in the broader area of data integration. In this area, semantics and more specifically languages such as the Resource Definition Framework (RDF) and the Web Ontology Language for Services (OWL-S) play a significant role. However, an important consideration is that open services randomly offered over the Internet are, at least currently, not described in terms of semantics. And, even if they use semantics, they would not adhere to a common ontology. As such, this approach uses the syntactical, natural language approach TSM-L to predict when the underlying web services messages are related. As such, we use human naming tendencies and the characteristics of service developers to inform our matching algorithms.

Figure 15 shows the average predicted mashups per service for each of the 9 automatically created categories across three different metrics. In the first, the services within a single category were compared to each other. In the second, the services in one category were compared with the services from all other categories. The last data point is just a measure of the predicted mashups per service between the starting category and the other category with which it is most mashable. That is, the other category with which it shared the highest number of potential mashups.
Using both potential mashups within a category and throughout the repository can lead to differing mashups. In a repository of categorized services, the likelihood that two services in the same category will share a descriptor is much greater than that of two services in different categories. With this in mind, it is logical to search within categories for mashup candidates. However, an important concept for prediction is the notion that the services are distinct and provide different functionality. To account for this, an upper boundary for service similarity was imposed to limit how similar two services could be. This cutoff was determined to be 60% through experimentation and is a novel approach to identifying mashups that are composed of sufficiently different, but compatible, services. In Figure 16 the testing repository has been categorized into nine categories using the COP method described above. Notice that the number of predictions significantly decreases for each category at the 60% cutoff. This drop suggests that a service similarity above that point indicates that the services in the predicted mashup are more similar than different. By imposing the maximum service similarity, the mashups predicted...
are legitimate potential mashups between services that are of different functionality.

Figure 16. Mashup Predictions found via service similarity (left) and without outlier (right)

Despite the risk of mashing services that are too similar when discovering mashups within a category, there is a significant performance advantage to searching for potential mashups to a particular service within its own category. In fact, both the 60% cutoff and using the service's category when searching for predictions reduces the number of services to compare to the target set of descriptors greatly. This may be very practical when performance is a concern in a system. There are competing reasons for advocating mashups throughout the entire repository, as opposed to only those within a category. These include more "natural" or varied predictions, involving loosely connected services that share just one attribute such as zip code. Another might be the breadth of the mashup search in the entire repository, which would incorporate changes to the other categories in the repository. If performance is not a major concern, but entirety is, searching in the entire repository is more likely to be the most beneficial mashup approach. Ultimately, it may depend on a case by case basis, which would allow the user of the system to easily switch the mashup search space from within a category to the entire repository. It is the last approach that is taken by the SAMS method. Mashups in the SAMS
approach can be found within a service's category, which impose the 60% cutoff for service similarity, and also throughout the entire repository.

Figure 17 is an example mashup created using this system. Using the set of descriptive strings from one service, it was able to identify several other services with which it could mash. It should be noted that the predictions center around the attributes of location and identification.

This software tool is capable of examining a repository of services and identifying potential mashups among the services. An interesting application of this tool might be in a repository of categorized services. In such an environment, this prediction tool can provide cross-category mashups to see what types of services aggregate most frequently or most accurately.

These tools augment different tasks for web service management. Using a syntactical similarity approach that stems from the SOC domain itself, services were matched accurately for recommendation and mashup. Furthermore, it was demonstrated how these approaches might be incorporated into an intelligent community of agents for networked management of web
services.

6. Discussion and Future Work

The SAMS architecture was implemented using Java and included the architecture components mentioned in Section 4. Although many of the components were tested individually, the entire system can be executed at once, with limited input from the user other than the number of categories and mashup restrictions or specifications. The repository reading component of the SAMS architecture is the slowest component in terms of running time. For the repository used in this work consisting of 545 real web services, the reading component executed in under two hours. The important data from the WSDL files in the repository is converted into an xml storage format, enabling subsequent executions to load the repository in just seconds. Therefore, the only time the system needs to re-read the repository is when services are added or removed from the repository. When adding or removing services it is possible to augment the reading component by targeting the new or removed services. Categorization in the SAMS architecture runs in under five minutes for the 545 services in this work. This is reasonable since the categorization component is executed infrequently. The relation identification component runs comparably to categorization in about five minutes for the entire repository of 545 services.

As presented in Section 3.2, a very compelling application of the SAMS architecture would be to facilitate intelligent agent communication for service management. In addition to using the various components of the architecture locally on internal services, it benefits all parties involved to allow interaction and knowledge sharing between different entities that have established a working relationship. In such a community, the SAMS implementation would require real-time support for actions such as agents joining and leaving the network, or new
service offerings that have become available. The environment would be constantly changing, but it the belief of the author that the SAMS approach would operate successfully. However, further research is required to understand approaches to updating the system in real-time (i.e. online learning). Furthermore, additional research would focus on packaging and deployment of an implementation of the SAMS architecture. Although the various components can be executed at once or individually, a more complete application would allow run-time customization of different settings such as which ambiguous strings to ignore from the repository.

The syntactical approach of this work is one of its defining features and probably most controversial. In today's computing environment, semantic methods are limited by lack of organization and sufficient computing power. Because of this, syntactical methods are practical and accurate enough. However, as Web 2.0 concepts emerge and gain popularity, and computing power continues to increase, it is likely that semantic methods will dominate the data management approaches of the future. This work is not meant to replace or ignore the value of semantic methods, but rather, supply a working framework for data management as it applies to web services until semantic methods can become more applicable. Ultimately, it may be the case that a blend of semantic and syntactical methods is the most effective strategy for data management.

7. Conclusion

In this work, a web service management architecture was proposed to enable deeper orchestration of web services and their data provisions. This architecture was implemented and shown to be effective for the applications of web service recommendation, intelligent agent collaboration and web service mashup. The architecture is meant to be in-place and require little
to no adherence from outside entities. The architecture is able to be implemented locally, by any entity that desires to gain insight into the connections that exist between the services that are used. This architecture relies on a novel syntactical approach for service comparison and similarity. Lastly, it is extensible and scalable.

8. Acknowledgements

This work was made possible by research grants provided by the National Science Foundation. The grants are: 0548514, 0634302, and 0723990.

9. References


Syntactical Matching." ICWS 2007: 575-582


[28] Pu, K., Hristidis, V., and Koudas, N. "A Syntactic Rule Based Approach to Web Service Composition", Proceedings on the International Conference on Data Engineering (ICDE’06), Altanta GA, USA


[33] StrikeIron - (http://www.strikeiron.com/)

[34] UDDI Version 3.0.2 OASIS Group (http://www.uddi.org/pubs/uddi_v3.htm)


[36] "Web Services Description Language (WSDL) 1.1" W3C (http://www.w3.org/TR/WSDL)

[37] Web Service List - (http://www.webservicelist.com/)
