A Multi-Agent Approach for Generating Ontologies and Composing Services into Executable Workflows

John McDowall
George Mason University
Department of Computer Science
4400 University Drive
Fairfax, VA 22030, USA
jmcdowal@gmu.edu

Larry Kerschberg
George Mason University
Department of Computer Science
4400 University Drive
Fairfax, VA 22030, USA
kersch@gmu.edu

ABSTRACT

This paper proposes rethinking how ontologies are used to compose web services into business processes. Unlike hand-crafted ontologies, we describe using a multi-agent system (MAS) to automatically generate semantic mappings from service interfaces. Comparing synonyms and contextual clues, we infer meanings of input and output parameters with no explicit semantics (as in a Web Services Description Language document). We further describe how this semantic mapping can be used to derive executable processes by comparing the derived ontologies of each service interface and mapping each service's outputs to inputs of every other service and finding the paths through the resulting graph.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Online Information Services – *Web-based services*; I.2.11 [**Computing Methodologies**]: Distributed Artificial Intelligence – Multiagent systems

General Terms

Algorithms, Experimentation, Theory.

Keywords

Multi-agent system, web service, ontology, semantic workflow.

1. INTRODUCTION

Web Services are promoted as the next evolution of software reuse—build it once, invoke it from any platform using almost any programming language. The promise of reuse is predicated on the widespread availability of web services that can be rapidly composed and recomposed into new workflows as business needs evolve. Instead of developing new systems or applications, developers will adapt to changing business processes by adding new services that users could invoke as needed. In addition to speeding up development cycles and simplifying the introduction of new capabilities to existing systems, widespread web service deployment will allow end users to add new functionality at run time.

EDBT 2010, March 22-26, 2010, Lausanne, Switzerland.

In its simplest form, the idea is for developers to expose capabilities as web services and register these services in publicly available registries using protocols such as Universal Description, Discovery, and Integration (UDDI). Users will search the registries for services that perform some desired task, and then compose those services into a workflow that meets their business needs.

Currently, most successful web services deployments are focused either on improving the developer's ability to write and update applications quickly, or on giving business partners access to data without allowing direct access to the data store. Sabre Holdings, one of the world's largest travel companies, has had great success deploying services to simplify its own application development and the ability of third parties to connect to their data systems [12]. However, empowering users to modify their workflows as business needs changed was not a consideration. Similarly, ING deployed a service-oriented architecture that simplifies the task of updating and integrating applications, but it too is focused on improving the lot of developers, not empowering end users [6].

The vision of run-time discovery and invocation of web services is powerful and attractive, but it has not come to fruition because realizing that vision is more complex than it sounds. In this paper, we describe an alternative method to enable dynamic service composition. We do this by presenting:

- An example scenario that illustrates the issues complicating dynamic workflow composition;
- A description of the technology shortfalls that are preventing dynamic workflow composition;
- A new approach to enable dynamic composition; and
- Contributions of this approach.

2. EXAMPLE SCENARIO

We present a simple but illustrative example drawn from two publicly available Web Services. The first service, XigniteGlobalQuotes [36], accepts a stock symbol as input and returns the opening, closing, and other price points of the stock together with the currency the prices are valued in. The second service, ForeignExchangeRate [37], returns an exchange rate based on "from" and "to" currencies submitted. If a user wants to check the value of a stock on the London exchange and convert the price from pence to US dollars, it should be a simple matter to parse the output from the first service to extract the currency type, submit that to the second service, and use the returned rate to do the calculation.

In practice, using the output from the first service as input to the second service requires significantly more work than expected. One obvious difference is syntactic: the first service returns a value called "Currency," while the second accepts a value called "CommaSeparatedListOfCurrenciesFrom." Both are of type string, but nothing in either interface indicates whether these elements use the same value set or XML schema declarations. A developer must read the documentation for each service to know whether the services can be composed in the desired way or if some intermediate step is required to convert the currency identifier supplied by the first service into the currency identifier accepted by the second service. As a result, the desired workflow cannot be composed by an end user when needed; it must be designed and composed by a developer in advance of the need. Previous work focuses on using an ontology to define the terms used as inputs and outputs together with their relationships to each other. While ontologies are a help, they have limitations—if the services were originally developed using different ontologies, many of the same manual integration steps will be necessary. Even if both services use a common ontology, it takes a knowledgeable modeler to develop and maintain that ontology.

There is an additional aspect to this scenario that we consider. The proliferation of new tools and data sources available to an end user can result in information overload. This is increasingly true in the field of intelligence analysis, where new data sources and services are being fielded more rapidly than most analysts can learn how to use them. As a result, end users revert to the tools they know best, regardless of whether a new capability might meet some of their analytical needs. By analyzing new services as they become available and determining what new workflows they make available, we may be able to assist end users by giving them the ability to ask new questions, such as "I have information X [i.e., some service input], what can I ask about it [i.e., what service outputs are available from X]?" Another potential use is giving the user the capability to define some needed information and then "backtracking" through the available services to find the possible inputs that will yield the desired information; this would be the equivalent to asking "I need information X, what information is required to get it?". Instead of analysts spending time learning how to chain these capabilities together, we can automate that discovery and composition and free them to concentrate on the information.

3. TECHNOLOGY SHORTFALLS

Many techniques for dynamic composition of web services have been proposed; an excellent overview can be found in [2]. The main reason dynamic composition has not been deployed in operational systems is that several elements must fall into place in a mutually supporting manner, and these elements have not yet materialized.

Foremost among the missing pieces is the semantic interoperability that is a prerequisite to integrating any two components in a system. We define semantic interoperability as the degree to which two interfaces use common meanings for input and output elements irrespective of the names of those elements. In the example cited above, the two services are not *syntactically* interoperable because each uses a different name to denote the currency element in its interface. If each uses the International Standards Organization (ISO) standard 4217 currency codes, they are *semantically* interoperable[1]. Conversely, if one service uses the ISO 4217 alphabetic codes and the other uses a text description that does not conform to the ISO designation, then the services are not semantically interoperable.

Whether two services are semantically interoperable is not readily apparent from interface definitions such as Web Services Description Language (WSDL), but must be interpreted from the context or specified in an ontology. This ontology may be stored externally and referenced by a Universal Resource Identifier (URI) or it may be incorporated directly into the WSDL. Without a common understanding of the meanings of terms, it is left to the developer to determine what the service does and what input and output data it uses. Service providers have little incentive to use common semantics; ensuring commonality consumes limited resources with no guaranteed benefit. Because developers have to do the same level of research to use web services that they do when using traditional libraries, the UDDI registries that were to enable dynamic service discovery and invocation have withered [23]; the public UDDI registries unveiled by large companies such as Microsoft and IBM a few years ago have all been closed. Their place has been taken by web repositories aimed at human users such as www.seekda.com. Such registries offer a way for developers to find and reuse services, but do not support dynamic discovery and invocation of services; these sites are generally design-time resources.

There are other missing pieces to the puzzle:

- Creating application workflows at run time,
- Handling errors within composed workflows, and
- Adapting to heterogeneous security models.

First, as discussed in [10], the ability to quickly and easily create or modify application workflows at run time is not supported by current workflow definition languages. These languages are complex and not readily accessible to end users. Second, error-handling within composed workflows requires manual analysis and intervention. In order to ensure that a workflow either runs to completion or fails in a graceful manner, potential error conditions must be anticipated and appropriate actions planned. Such error handling must ensure that consumers are not penalized by receiving incorrect data or being charged for services not rendered. Third, the use of different security models by different services prevents ready interoperability because services cannot share authentication, authorization, and billing information for users who invoke the services. Each of these shortcomings of the current web services landscape is discussed below

3.1 Semantic Interoperability

The single largest barrier to run-time service discovery and invocation, as well as the composition of services into workflows, is the lack of semantic interoperability of services from different providers. This issue is not confined to users attempting to find

[1] This is true even if one uses the ISO alphabetic codes and the other uses the ISO numeric codes, as the ISO standard provides an unambiguous mapping of one to the other

services through semantic discovery; every developer who hopes to use a service must spend time learning the semantics of that service and how those semantics differ from other components of the system. Toward that end, there has been a great deal of research into different methods for enabling semantic interoperability [1,26,31,41]. All of these approaches have a common thread: the use of an ontology to capture the semantics of a particular field of endeavor and the mapping of service descriptions to that ontology, thereby minimizing the ambiguity inherent in human language.

One solution would be for service providers, within a given community of interest, to agree upon a common ontology or a common architectural framework, thereby defining a common vocabulary and relationships among terms. This option is not generally attractive to service providers; the negotiations required to agree on a common ontology are time-consuming, and must be largely complete before interfaces can be finalized. Agreement on a common architectural framework might be easier, but would still require all parties to ensure the framework was compatible with their implementation environment. The resulting additions to development timelines consume valuable resources for an uncertain benefit. Even if there were guaranteed benefits from cooperation, the success of the negotiations would not be guaranteed; the resolution of such inter-organizational issues is never as straightforward as the resolution of technical problems.

Semantic Web Services (SWS) are one proposed solution to this issue. SWS, as described in [27], include semantic markup as part of the web services description; the ontology is combined with the syntactic markup found in a WSDL document into a single interface specification. Several mechanisms have been proposed over the years, beginning with the DARPA Agent Markup Language-Services (DAML-S), which evolved into the Ontology Web Language-Services (OWL-S) and other proposals such as WSDL-Semantic (WSDL-S), also known as Semantically Annotated WSDL (SAWSDL) [24,40]. Some of these techniques have been combined with some success [22], but none to date has solved the problem. And while these efforts have each shown some promise, they all have the same limitations as other ontology-driven efforts: semantic heterogeneity. If one concept is described by two different ontologies, each using different aspects of the concept, the two descriptions are not readily interoperable.

Some interesting work has been done to address the use of multiple ontologies and the need to match ontologies to enable interoperability. The work described in [28,30,29] explores a method for comparing ontologies and quantifying their similarities with the goal of identifying semantic matches; the OWLS-MX matchmaker [20] has shown promising results in matching service interfaces based on semantics and syntax. Still, this is a process that depends on the manual definition of each ontology. Ultimately, all semantic interoperability schemes suffer from the same limitations: ontologies must be manually generated, and some human must perform the semantic matching of services that do not share a common semantic markup.

3.2 Workflow Development

Within the context of this discussion, we use the term “workflow development” to refer to a multistep process wherein two or more Web Services are composed in such a manner that the output of one is used as input to the next, continuing until the desired task is completed. A workflow can be classified as one of two types: orchestration or choreography. Orchestration describes an executable process that combines several services into a single logical unit that is controlled by a single party to a transaction. An orchestration may be deployed and invoked as a discrete unit. By contrast, choreography is a more involved process where participating services exchange messages in order to accomplish some task. This makes choreography well-suited to agent-based applications, where individual agents can represent the constituent services and negotiate the workflow composition and execution by passing messages among themselves, similar to mechanisms described in [19,17,41]. For the sake of simplicity, and because the concepts described here apply to both orchestration and choreography, we use the term workflow to encompasses both types of process composition

The difficulty of composing web services into complex workflows is another major barrier to the advent of rapidly reconfigurable applications. The current service workflow definition languages are very expressive but also very complex. Among these languages, the Business Process Execution Language for Web Services (BPEL4WS) is the most common and well-known. Other proposed approaches, such as the Web Services Choreography Description Language (WS-CDL) [3], are likewise too complex to be readily accessible to the lay user. Even if all available services were semantically interoperable, the complexity of workflow definition languages would present a significant barrier to dynamic composition.

One suggested approach for improving the usability of workflow definition languages is to develop a symbolic encoding that would simplify manipulation of the workflow components. Mapping Unified Modeling Language (UML) concepts to BPEL4WS is one approach. In [38], Skogan et al use XSLT to transform UML to BPEL4WS. This has the advantage of using a symbolic language (UML) that is familiar to a large number of developers and business analysts. However, there are no established semantics for BPEL4WS. For this reason any standard mapping of UML to BPEL4WS remains an open question as discussed in [25]. Such a mapping would help to unlock the power of BPEL4WS and make it accessible to users. But even with a way to symbolically manipulate BPEL4WS workflows, the problem would still not be solved because of the inherent complexities of BPEL.

An alternative approach is described in [39], where UML activity models are used to represent a generic workflow that is then matched to concrete services and refined to fulfill the workflow purpose while meeting the user’s preferences (e.g., booking travel while not exceeding a given cost threshold). This approach does not translate the UML into BPEL4WS or another formal process language, but its constraint-based approach does provide a strong formal process to verify that the composed process meets the user’s needs. It should be noted that this process relies on the developer to ensure the services are semantically interoperable.

Another option for making workflow development more accessible is to find a way to simplify matchmaking between services—determining that the output message of one service can be used as the basis for an input message to another service. Any such match need not be exact. Taking the example at the beginning of this paper, the output of the XigniteGlobalQuotes service is a superset of the inputs required by the ForeignExchangeRate service; we can compose them into a workflow even though the match is not exact. (Note that this also assumes the services are semantically interoperable.) Once such simple connections can be discovered, inferring longer, more complex workflows can be viewed in terms of graph theory (as explained below).

If a data set that serves as an input or output message for a web service is thought of as a node, and any service that uses one node as an input and another as an output is thought of as an edge, then the set of available services forms a directed graph that can be analyzed to find possible paths from any known data to any desired data. (The work in [21] describes a similar approach where the nodes are services and edges represent data.) It should be noted that any such graph must be acyclical, as cycles in the graph may produce unpredictable results. A more detailed discussion of this concept is presented in Section 4.2.1.

Obviously, the problem is more complex than just discovering paths. Not all workflows are focused on retrieving data; many are focused on performing some real-world function such as purchasing goods. Therefore, matching input and output messages is only the start; knowing the operations performed by each service and the real-world effects of those services (such as charging a credit card) is also important. Like the input and output elements, the operations must also be semantically interoperable to some extent. A newly discovered service with an operation "GetCurrentRates" may perform the same function as the ForeignExchangeRate service operation "GetLatestRates," or it may return information about prices of electricity (depending on the meaning of "current").

One other issue is important to workflows with real-world effects: the need to maintain the state of the transactions that are part of a workflow whose components are themselves stateless. For example, consider a workflow that searches for a camera meeting particular criteria, selects a vendor based on some combination of features and price, and then purchases the camera. First the workflow will need to search available cameras to find which ones meet the specified criteria. Next, the workflow will have to check prices for each of the cameras. The workflow will then need to make a selection and execute the purchase. Obviously, maintaining the state of the process is essential to its successful completion. This includes tracking which cameras meet the specified criteria, which vendors carry the desired cameras, and available prices from each vendor. In addition, the ultimate selection and the status of the billing transaction must be maintained to complete the workflow successfully. Such a workflow requires either a state-aware data structure that maintains the state of the workflow or some external support framework to maintain the state of the process.

3.3 Error Handling

A critical enabler of any workflow system is error handling; unless errors are recognized and dealt with appropriately, the ultimate success or failure of any composed workflow may be ambiguous. Determining how to react to errors in a composed workflow is perhaps the most complex task of the workflow development process. One of the strengths of BPEL4WS is its robust error-handling mechanism. Process designers can define error conditions and specify the actions to be taken if an error occurs. While powerful, this is also part of what makes BPEL4WS a very complex language. To enable the dynamic composition of workflows, we must also simplify error handling.

There are different types of service errors. Consider a free service that takes a location as input and returns a weather forecast. If this service fails, the only adverse effect is that the user gets no forecast. In the case of the camera purchase workflow discussed above, failure can be more disruptive: if the purchase service fails after the credit card billing is submitted but before the order to actually ship the camera is sent, the user's card will be charged without the camera being sent. In any composed workflow, predicting, detecting, and handling these error conditions correctly is the difference between a reliable workflow that helps a user accomplish some task and a composition whose behavior is indeterminate.

The seemingly simple act of detecting errors is a non-trivial undertaking. Some errors are simple to detect: the service returns an error message. Other errors may be more subtle. If a service operates asynchronously the user sends the input message and waits for the output message. How long should the consumer wait for a reply before deciding the service has failed? Some services return small amounts of data and can be expected to return within seconds; others may return larger amounts of data or be located far enough away that network traffic issues can cause the service to take longer to respond in some circumstances than others.

Detecting errors is only part of the problem. What to do with those errors is another matter entirely. Some workflows may be more fault-tolerant than others. If workflows are derived from a directed graph as discussed earlier, then it may be possible to route around the error if other paths from the start point to the desired end point are available. If no alternate route is available, partial results may be acceptable to the user. For example, if a workflow accepts a location and time as input, retrieves a weather forecast for that time and place, then plots it to a map and returns the map to the user, the raw weather forecast may be an acceptable response if the mapping service fails.

Another important aspect of dealing with errors is rolling back a partially completed transaction. In the case of the camera purchase described above, detecting that an error has occurred between credit card billing and shipment of the camera does little good unless the credit card transaction can be canceled. Relational databases implemented the two-phase commit protocol for this very reason: completing only half of a financial transaction is unacceptable. While BPEL4WS also has mechanisms for dealing with this issue [7], these potential errors must be identified by the process designer, and that designer must specify the action to be taken in the event a fault is encountered. A mapping of service errors to a common semantic vocabulary would provide a ready way for inferring the severity of an error and the appropriate response.

3.4 Security

The remaining component for addressing workflow composition is security. Authentication, authorization, data integrity, non-repudiation, and more must all be adequately addressed to ensure the integrity of composed workflows. For many simple workflows (such as our original example), the involved services can likely be invoked anonymously. But for workflows involving financial transactions or sensitive data, security considerations are paramount and are a major reason the reconfigurable workflow vision remains unfulfilled.

Harmonizing security models is no easy task. Different services may use drastically different security models. Some services allow anonymous use; others require a simple username / password pair, and still others may require a digital certificate. These mechanisms are not inherently interoperable, and bridging their differences is far from simple. Even within a single organization such as the US Department of Defense, there is no agreement on how best to manage service security. There are standardization efforts underway, such as the OASIS WS-Security specification, but none has achieved the critical mass required to become a de facto standard.

Security is another example of where semantic interoperability is also important. As with input and output elements, just because two services' security characteristics are syntactically interoperable does not mean those services are semantically interoperable. The frameworks described in [16,13] provide a method of interoperability where the security objectives and specific properties of the system and individual services are formally described, and the requirements of the services comprising a workflow can be negotiated to come to a mutually agreeable solution that allows the workflow to proceed. While promising, this concept suffers from the same adoption issues that plague semantic markup of service interfaces: until there is a critical mass of services implementing it, there is little incentive for service providers to adopt this method; and until service providers begin to adopt this method, the critical mass will not be reached.

Another approach is to establish networks of trust among service providers and identity providers. But establishing mutual trust among disparate organizations remains a work in progress; many organizations with very sensitive data are leery of trusting identity providers over whom they have no leverage. However, industry efforts such as the OpenID foundation are attempting to assemble a critical mass of major industry players to create a de facto identity management standard. The OpenID foundation has met with limited success to date, but their efforts are still in their infancy.

Other efforts, such as the Shibboleth project (shibboleth.internet2.edu), seek to create webs of trust among organizations that can elect to trust each other as identity providers or service providers. These are binary relationships, but the resulting webs of trust can grow to include multiple identity providers who trust and are trusted by multiple service providers. To be sure, this is not the be-all and end-all of security, but getting a handle on identity management is the first prerequisite to enabling interoperable security among services from many different providers.

4. TECHNICAL APPROACH

Much good work has been done to lay the groundwork for dynamically composing services into complex workflows, but all previous approaches share one or two major drawbacks. One is the use of a purpose-built framework for developing, deploying, and managing services. Frameworks that take this approach provide the necessary interoperability at the expense of flexibility. All the services to be integrated must use the common framework to enable workflow composition. Services that were created using a different framework (or no framework at all) are not readily integrated into the framework without significant rework.

The other drawback is the use of manually created metadata to annotate service interfaces. This metadata, in the form of ontologies or other markup, is time-consuming to produce and maintain. Because it is oriented toward a specific set of services used in a specific business domain, such hand-crafted metadata also tends to be brittle; changes to service interfaces require changes to all the relevant metadata. At some point the collection of services and metadata will reach a critical mass where maintenance of existing artifacts consumes more effort than the integration of additional capabilities.

4.1 Rethinking Ontologies

Part of the problem is the ontologies themselves. Traditional ontologies are carefully crafted representations of a field of knowledge, capturing classes of things, their characteristics, and the relationships among the classes. These are excellent mechanisms for capturing the complexity and nuance of an entire field of knowledge, but for describing the limited number of inputs or outputs in a typical web service, they are overkill. For example, an ontology describing weather will contain classes for cloud types, cloud layers, weather events, temperature types (ambient, dew point, current, forecast) and other factors important to a thorough understanding of the weather domain. By contrast, a weather web service can be expected to contain elements for current temperature, forecast maximum and minimum temperatures, probability of precipitation, and other factors pertinent to the forecast for a given location. Matching the few elements in a service interface to the many elements in a fully-formed ontology requires a large and growing number of comparisons as the ontology expands.

Instead of large, complex ontologies that attempt to describe an entire field of knowledge, it would be much more efficient to create smaller ontologies containing only those elements needed to describe the input and output messages of services that are of interest to a given community of users or developers. Smaller ontologies are easier to build and maintain, and require many fewer comparisons when evaluating whether a given service's inputs or outputs fit into the ontology. Smaller, simpler ontologies may have another useful characteristic: it may be possible to automatically generate them from the services themselves.

4.1.1 Calculating Semantic Similarity

Every service interface includes input and output messages made up of one or more parameters. The names and arrangement of these parameters contain an implicit ontology embedded within their structure; they reflect the developer's understanding of the organization of the data the service operates on. Current web service compositions are built upon these ontologies: developers use the service interface and any available documentation to determine what inputs and outputs each service uses and how those parameters relate to the inputs and outputs of other services.

Developers use the names of the service parameters, together with contextual clues, to infer what data types the parameters refer to. A formal version of such a process is described in [34]. While programming such a complex process into a single program would be extremely difficult, it should be possible to develop an analogous process using an agent based system where each agent performs a simple task. Because each agent performs only a single well-defined task, the programming of each agent is relatively simple. The interaction of many such simple agents can yield very complex behaviors.

Each input or output parameter of a web service is described by several characteristics: the name, the datatype, the name of the operation, and other factors as described in [5]. In addition to the characteristics that are specific to each parameter, there is additional contextual information: namespaces referenced by the WSDL, the names of operations associated with the parameter, and other factors. Each of these can be the subject of a specific type of agent that is specialized to evaluate that factor.

To infer if the inputs and outputs are semantically compatible, the attributes of each service's parameters can be compared to each other and their similarity evaluated to generate a compatibility score. Each factor is evaluated and scored on a scale in the range of 0..1, where 0 is no similarity and 1 is an exact match. These scores are then weighted differently to reflect the likely relevance of that factor to the semantic type of the parameter. An example of some factors to consider and possible weights is shown in Table 1 These factors are those available in a typical WSDL; others could be added. The weights shown are purely speculative and are shown for illustrative purposes; they are based on the likely influence that a given factor has on the semantic similarity of the service parameters. If both A and B reference the same namespaces, we can reasonably infer that any two parameters with the same name are more likely to have the same meaning than any two parameters with the same name whose services reference different namespaces.

Table 1 - Possible Comparison Weights

WSDL Element	Possible Weight
Namespaces	0.4
Operation Name	0.1
ComplexType Name	0.2
Parameter Name	0.2
DataType	0.1
minOccurs	0
maxOccurs	0
Total	**1.0**

Assigning a specialized agent to evaluate each of these factors allows us to program that agent in a way that optimizes its performance for that narrow task. A supervisory agent can then collect the evaluation scores generated by each agent and combine their weighted scores to generate an overall similarity ranking. Decentralizing the evaluation of individual factors in this way has the added benefit of increasing the number of factors that are evaluated by deploying additional agents that perform additional evaluations and report their results to the supervisor.

The evaluation of each parameter individually gives an indication of the likelihood that any one parameter is semantically compatible with any other parameter. The combination of these evaluations allows us to infer whether the output of one service can serve as the input to another. Inferring these semantics is no easy task, but the work described in [28,29,30,35] indicates that it may be possible.

4.1.2 An Illustrative Example

Consider two services, A and B. The output of A can be used as the input to B if the set of inputs of B is the same as, or a subset of, the outputs of A ($A_{out} \supseteq B_{in}$) and each input of B corresponds to an output of A of the same semantic type. (As will be shown, syntactic matching is not necessary but would be helpful.) For this example, let A be a service that accepts a latitude and longitude as input and returns the postal code, county, and state in which the position is located; let B be a service that accepts a postal code as input and returns a weather forecast for the designated area. The relevant parameters are summarized in Table 2.

To compare these services, agents are deployed that perform several checks to evaluate the compatibility of these service inputs and outputs. One agent does a straight comparison of the namespaces and evaluates their overlap. Another agent checks each parameter name against other parameter names through such means as a direct syntactic comparison, synonyms derived from WordNet, the application of stemming algorithms, and other techniques that may yield an indication that two parameters may be related.

Table 2 - Service Parameters

Service A Output				
Namespaces	<none>			
Operation	GetPostalCode			
ComplexType	PostalRegion			
Parameters	**name**	**type**	**minOcc**	**maxOcc**
	postalCode	integer	1	1
	county	string	0	1
	state	string	0	1
Service B Input				
Namepsaces	<none>			
Operation	GetWeatherForZip			
ComplexType	*<none>*			
Parameters	**name**	**type**	**minOcc**	**maxOcc**
	zipCode	integer	1	1

Still other agents perform similar evaluations on the remaining aspects of the interface, such as parameter data types, ComplexType names, operation names, etc. Applying these comparisons to each parameter from Service A results in a semantic similarity score for every parameter in Service B; those parameters with similarity scores exceeding a set threshold are determined to be semantically compatible. In this example, Service B has only one input parameter (zipCode), and it is evaluated as sufficiently similar to one of Service A's output parameters (postalCode) that we determine they are semantically compatible. Because all of Service B's inputs have a compatible match in the output of Service A, we know that Service A and Service be can be combined into a simple workflow.

4.1.3 Dealing with Unknown Terms

Particularly in the early stages of the analysis described here, we can expect that there will be service parameters that cannot be matched to any known terms. This may be because the parameters are new, and therefore have no corresponding parameters among the terms that have already been identified and matched. These are new additions that must be added to the set of terms to be compared as more services are analyzed; it is fair to expect that as more services are added the new term will eventually be matched with parameters that are part of the additional services.

In other cases, the lack of a match could be due to some aspects of the service interface that prevent the agents from finding any suitable candidate matches to known parameters. In these cases, the new parameter should be presented to a user for confirmation as a new semantic type or for manual selection of an appropriate match. This adjudication has the added effect of giving the system an ability to learn even though the individual agents may not be designed to learn from their experience.

4.1.4 Generating a Service Ontology

As more comparisons of the type described above are performed, we expect that certain common characteristics of compatible parameters will emerge. These common characteristics can be gathered to develop "master terms" that can serve as initial points of comparison for new service parameters. Relations among these master terms can be inferred from the frequency with which they appear together and any similarities among ComplexTypes containing them. By capturing these master terms and their relationships we can generate a minimal ontology that contains only those terms and relationships that are relevant to the set of services under consideration. As additional services become available, the ontology can grow to accommodate the new parameters and relationships.

Any ontology developed in this manner will undoubtedly have some "orphan terms" that are parameters for a service but have no compatible terms or established relationships to other terms. Some of these orphans will become a part of the ontology as more services are added and the set of parameters grows; others may remain orphans indefinitely.

Any ontology developed in this manner will not necessarily be an ontology in the classic sense of a structure that captures all the nuances of the subjects and relationships in a field of knowledge, but it will be as detailed as needed for the purpose of comparing service interfaces in order to generate possible workflows. Despite the initial limitations of the content of this ontology, it should still be possible to capture that ontology in a formal representation such as OWL or RDF. In addition, it should be easy to update such an ontology as more terms are learned or by allowing a human ontologist to manually modify it.

4.2 Defining Workflows

Determining that the outputs of one service can serve as the inputs of another service is a single step that will allow us to build only the simplest workflows. To build more complex, multi-service workflows we will need to find all the possible compositions among the set of available services.

Consider four services denoted A, B, C, and D. As explained above, any two services can be composed into a simple workflow if the outputs of one can be used as the inputs of the other (e.g., $A_{out} \supseteq B_{in}, C_{out} \supseteq D_{in}$). Longer workflows can be composed by applying the same principles across the set of available services. For example, A, B, and D can be composed into a workflow if the outputs of A can be used as the inputs of B and the outputs of B can be used as the inputs of D (i.e., $A_{out} \supseteq B_{in} \wedge B_{out} \supseteq D_{in}$). If the outputs of C can be used as the inputs of D, then we have the following possible workflows in this set of services: A to B, A to D, B to D, and C to D.

As the set of available services expands, this web of potential workflows will continue to expand, and maintaining the list of possible workflows will become more difficult unless we are able to leverage some other method for deriving workflows. Pairwise comparisons of a large number of services would be prohibitive. To avoid this, we plan to make use of the "master terms" discussed in Section 4.1.4. By comparing the inputs and outputs of any service to only the master terms, we can reduce the number of comparisons. Every match of a parameter to a master term equates to a match of that parameter to all the other parameters that match that term. These matches reduce the number of message comparisons that have to be made to only those that have already established a potential match thorough the match to a master term.

4.2.1 Workflows as Directed Graphs

The input and output messages of a service are in effect simple data objects. If each of these data objects is thought of as a node in a graph, each service can be seen as an edge in that graph. The result of this is a directed graph that captures the inputs and outputs of each service and their relationships to each other. Any path through this graph represents a potential workflow from one data object (service input) to another data object (service output). (The work described in [21] uses a similar concept, but uses nodes to represent services and edges to represent data.)

In the simplest case, if the results of each matching of a service's input and output parameters to another service's input and output parameters are captured as a node in the graph, the set of possible paths through that graph equals the set of potential workflows that can be composed from the available services. The application of any algorithm that calculates the paths among all nodes will yield the set of potential workflows. Some of these workflows will doubtless be trivial, and others may not have any particular business use. But regardless, all the potential workflow compositions that can be created from the available services will be calculated. Using services A, B, C, and D discussed above, an example of this is depicted graphically in Figure 1.

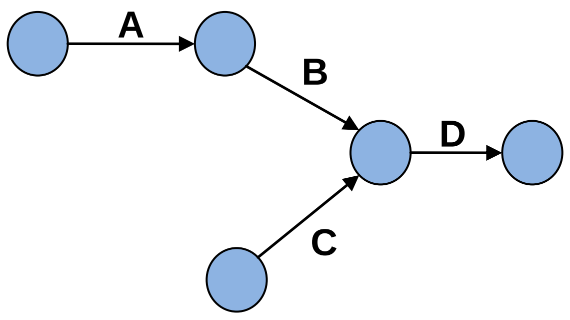

Figure 1 - Workflow as a Directed Graph

There are complexities that the simple case discussed above must address to be of any utility. To begin with, the paths through the graph that represent workflows must avoid cycles; any cycle in a given workflow would result in a workflow that has no end. Likewise, identifying duplicate paths between any two nodes will be necessary for two reasons: first, to ensure we select the optimal composition for any given business process (i.e., initial and final data set); second, to identify alternate paths to completing any given business process. These alternate paths are an important component in identifying ways to detour around services that are not functioning properly or whose costs are more than the user is willing to incur.

Once these workflows are identified, they can be captured for later use without the need to re-calculate them all at run time. Capturing them as simple BPEL4WS processes would also allow the possibility of exporting them for use within other systems that have access to the same set of services.

4.2.2 Path Management

The potential number of paths that will result for a large number of interoperable services may become very large, making path calculation a very resource-intensive undertaking. Calculating these paths for immediate presentation to the user each time a new service is introduced is probably not feasible for a large number of services. This is especially true because we can expect that a large number of services will not only introduce new paths to the graph but will introduce new nodes as well, requiring the existing paths to be re-calculated each time a new service is added.

We expect different sets of paths may be of interest to different users. Some users may know what information they need (e.g., a weather forecast) but not the prerequisite information required to get it (e.g., a city name, a postal code, etc.). Other users may have the exact opposite problem, knowing what information they have and wanting to learn what additional information they can retrieve related to it. Still other users may know what they have and what they want but not know if it is possible to get from one to the other with the available services.

One of the advantages of the agent-based approach we are proposing is that the different path calculations can be conducted concurrently. While some agents are analyzing a new service and finding potential matches, other agents can update the paths available for different uses as new matches are found. And because the agents can be readily interchanged at run time, we will be able to experiment with different analysis methods for analyzing the service connections to find available paths through the graph.

4.2.3 Additional Considerations

The simple case described above allows us to calculate only simple single-threaded workflows. There are more complex workflows that require a more sophisticated analysis before they can be made available to the user. Consider the services depicted in Figure 2. In contrast to the nodes shown in Figure 1, this graph has a node (denoted by a hexagon) that requires the output of two services to make up the input of a third service.

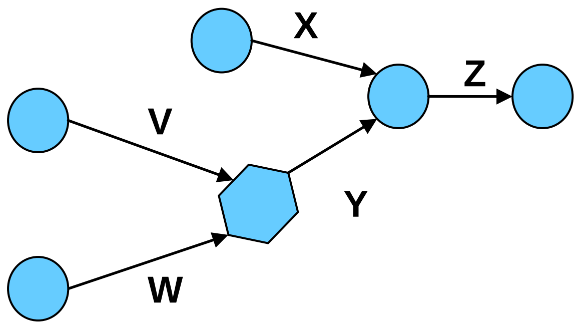

Figure 2 - More Complex Workflow

In this example, the input to service Z is composed of the outputs of services X and Y (i.e., $X_{out} \cup Y_{out} \supseteq Z_{in}$). When calculating possible workflows that can be created from our set of available services, we require the outputs of both services W and V in order to form the input required by service Y. If the user has the data required as input to service X or the inputs required by both W and V, then a usable path to Z can be calculated; if the user has only the inputs required by V or the inputs required by W, then neither service Y nor service Z cannot be part of any viable workflow. This sort of conditional path calculation is not accounted for in standard algorithms that calculate all-points paths within a directed graph. Because of dependencies such as this within the graph, more sophisticated algorithms will be necessary if more complex workflows are to be calculated from the set of available services.

4.3 Error Handling

Most WSDL descriptions contain no direct indications of the ways in which the service they describe might fail, so inferring possible failure modes within any workflow composition developed in the method described here will not be a simple matter. The most obvious type of failure is a service that returns no response. This can be accounted for with simple timers that consider a service to have failed if no response has been received within a pre-determined time limit. As there are no specific quality of service metrics defined within the WSDL specification, deriving a time limit might be best handled as a trial-and-error process. Supervisory agents monitoring workflow execution could keep track of execution times for individual services and develop performance profiles that could then be used as a guide for determining when a service was past its expected completion time.

In addition to timing out, services may return error codes when they encounter a problem. Just as the semantics of inputs and outputs are learned, the semantics of errors could also be learned. By capturing input and output values for services whose parameters have been learned, agents could learn to recognize “good” output from errors.

While recognizing an error does not guarantee that the error will be correctly handled, it is a necessary prerequisite to handling the error correctly. Once an error is encountered, supervisory agents can attempt to route the workflow around the error if an alternate path exists, or if no alternate path is available the error could be returned to the user for manual resolution.

4.4 Security
Dynamically harmonizing security models is not an easy task, and nothing described here can, by itself, solve that problem. However, if the semantics of the input and output parameters can be learned, it follows that a similar process can be applied to any available security information to be found in the WSDL. In some cases, this information may already be captured in the inputs in the form of a username and password; other cases may be much more complex. While a more thorough investigation of security harmonization is necessary, the process described here could, at a minimum, infer whether any two services used a similar security model and could be expected to inter-operate with relative ease. This is not a complete solution, but it does offer a minimum level of matching that could be used in generating workflows at run time. (Note that this all assumes the available services are not part of a common security framework; use of such a framework would make this step superfluous.)

5. IMPLEMENTATION
The approach discussed above is a largely theoretical, but some initial, admittedly very limited testing of the ideas described shows enough promise to merit further research. To gauge the viability of this approach, we conducted a limited evaluation of service interfaces to evaluate whether an automated matching approach might prove useful. Lacking a readily available service registry, we elected to select a sample of services from a publically available service search engine and test those against the example described in Section 4.1.2.

5.1 Initial Steps
The implementation to date has been confined to initial steps to gauge the feasibility of the approach discussed and illuminate the difficulties that we can expect to encounter if we proceed further.

Our initial implementation is built on the JADE (Java Agent Development) framework [4], an open-source framework initially developed by Telecom Italia. Our tests use a small number of agents to parse the service WSDLs and do some simple analysis.

5.1.1 Superintendent Agent
The Superintendent Agent is the initial entry point for any service. It receives the URL for a WSDL, retrieves the WSDL, then parses the WSDL into its constituent operations and their input and output messages and their parameters. These operations, messages, and parameters are all loaded into a relational database for further processing. Each operation is then passed to a Foreman Agent for further processing.

5.1.2 Foreman Agent
Each Foreman Agent receives an operation and retrieves its input and output messages. Each message is parsed for its input and output parameters, and each parameter is passed to a Crew Chief Agent for more detailed analysis.

5.1.3 Crew Chief Agent
Crew Chief Agents are where the real work of analysis is done at this point. For now, the Crew Chief analyzes each parameter and compares it to other parameters. We plan to create a series of specialized agents that will each perform some specialized analysis of each parameter and report results to the Crew Chief, which will combine the results and generate a similarity ranking. For example, one agent might take a parameter and retrieve synonyms from WordNet for comparison to know parameters, while another might compare the name of the parameter's message to known message names.

5.2 Feasibility Analysis
Attempting a purely syntactic analysis using the search engine available from seekda.com, we conducted two searches; one for services containing "weather" and another containing either "latitude" or "zip." Knowing that weather forecasts are generated for specific geographical areas, our hope was that some weather services would accept some location identifier (such as a zip code or postal code) and some location services would return a location identifier usable by one or more of the weather services. For each search, we selected the first 40 results to use as our test sample. For each result, we retrieved the WSDL and parsed it into its available operations, the input and output messages for each operation, and the individual parameters for each message. After eliminating those results where the WSDL was unavailable, we were left with 68 unique location operations with a total of 134 output parameters and 30 unique weather operations with a total of 187 input parameters.

5.2.1 Results
Attempting a purely syntactic matching of location output parameters to weather input parameters resulted in zero matches even though several of the location and weather services were hosted by the same provider. Extending the analysis to a simple parsing of parameter names where they are broken up based on "camel case" and then matching parts of parameter names (e.g., "ZipCity" matching "city" and "CityName") resulted in 20% of location output parameters matching inputs to weather operations and 27% of weather inputs having matches to location operations.

These initial results use all available operations from the services selected. If we assume that an operational use of such a system would use a set of services that are generally bounded to a specific domain of interest, some of the operations analyzed here would doubtless not be included in the pool of available operations. For example, the operation "GetDistanceToWater" may not be of interest. Likewise, some operations, such as "GetWeatherByZipCode" were in both samples. Filtering out duplicates and those services that are only peripherally related to retrieving locations and weather information, our results improved somewhat. Of the remaining 43 location operations, 30% of the 84 output parameters had a match to a weather operation input; of the 21 remaining weather operations 32% of the 160 input parameters matched an available location output.

5.2.2 Limitations
While we can expect that there are a number of false positives among these results, those can only be identified through manual analysis or composing the services and attempting to execute the resulting composition. For example, the input "city" may denote an identifying code and not the city name.

Similarly, several outputs use a custom data type such as "ZipCity" or "AddressPlus." While it is likely these results contain the information necessary for generate an input for another service (e.g., “AddressPlus” may contain a city name), many of these potential matches are not captured by our analysis and will require a more sophisticated message parser to enable us to automate the required analysis.

5.3 Next Steps

The results discussed here are admittedly very limited; they make no substantial attempt at inferring the semantics of the parameter names and completely ignore namespaces, operation and message names, and data types. Despite these limitations, we are very encouraged by these results and feel they indicate that the proposed approach is promising and merits further research.

6. RELATED WORK

The issues discussed earlier surrounding the discovery and use of web services have combined to make static binding the norm. But there has been promising work toward simplifying process composition and improving semantic interoperability. One of the most thorough efforts in this area is the METEOR-S Web Services Automation Framework (MWSAF) [33], which developed a framework that supports the semantic markup and subsequent composition of web services into complex workflows. This framework is aimed primarily at business-to-business applications, where domain experts can craft reusable workflows that are readily reconfigured as the needs of the business evolve over time. However, this is still a system that relies on the use of experts to develop and reconfigure workflows as needed; it does not put that power into the hands of end users.

Another framework is the Web Service Execution Environment (WSMX) [10], which enables the development and execution of web services based on the Web Services Modeling Ontology. However, while WSMX provides an interesting mechanism for marking up web services and performing matchmaking, it does not fully address the issues of composing complex workflows from web services discussed here.

Still another promising avenue is the Knowledge-based Dynamic Semantic Web Services (KDSWS) Framework described in [14], which developed a framework for managing the lifecycle of semantic web services. The KDSWS framework includes specifications for models and languages that aim to solve the problems of semantic harmonization and service composition. When the Knowledge/Data Model and Language (KDM/KDL) are used to describe the information domain of interest, the Knowledge-based Dynamic Services/Process Model and Language (KDSPM/L) can be used to rapidly compose semantic web services into executable processes. But even this framework still relies on developers marking up their services with the appropriate semantic information.

One effort with some similarities to our approach is the Woogle web service search engine described in [9], but there are notable differences. Rather than developing a search capability that will let users find and develop applications with services, we are focusing on developing a capability that find services and build the compositions for immediate use by the user. Also, while Woogle uses a clustering algorithm to find similar services based on the assumption that input and output parameters are generally grouped with similar parameters, we make no such assumption and instead rely on matching individual parameters and then building up groups of those parameters to match service operations. This has the added benefit of enabling us to find instances where the outputs of two operations can be combined to form the input of a third as discussed in Section 4.2.3. However, the clustering used by Woogle may be very useful in generating ontologies based on service interfaces as discussed in Section 4.1.4

7. CONTRIBUTIONS

Our contributions fall into three general categories: ontologies, metadata generation and workflow composition.

7.1 Ontologies

Previous work on service metadata has focused almost entirely on associating an externally-generated ontology with a service interface. Much of this work has concentrated on inserting OWL or RDF ontologies into the WSDL or updating the WSDL specification to include an ontology [20,24,26,32,40]. Some of these approaches store the ontology outside the WSDL document, but they still rely on a classic, hand-crafted ontology.

In contrast, our work completely rethinks the concept of the ontology. Whereas a traditional ontology is a detailed model of the knowledge of a specific field, we propose a radically simplified ontology that captures only those elements that compose the service interfaces under consideration. Such a minimalist ontology is not suitable for capturing and describing an entire field of endeavor, but it is sufficient to the desired purpose of matching service inputs and outputs.

7.2 Metadata Generation

While some there has been some work on automated metadata tagging [8,11,15], this work focused on matching metadata to service descriptions. Unlike previous work we are attempting to generate the metadata from scratch based on the information available in a standard WSDL description. Furthermore, our approach uses a multi-agent system that has two advantages over other systems. First, a system composed of many simple agents can be readily modified by adding new types of agents to the system and factoring their inputs into the overall evaluation. Second, a system composed of many agents holds the possibility of exhibiting the emergent behaviors often found in such systems, similar to that described in [18].

7.3 Workflow Composition

Most workflow composition systems concentrate on simplifying the task of developers by making process definition languages such as BPEL4WS simpler to use as in [38], or automated matching to achieve a specific goal as in [21]. These approaches both take the approach of finding specific services to meet a user’s expressed requirements. Our approach takes the set of available services and finds all the possible workflows available within the set. This may not yield a specific business process the user wanted, but that process could not be built from the available services even if the process were defined manually. Our approach has the advantage of finding workflows that may meet the user’s needs but would not have been obvious without an exhaustive comparison of all possible service combinations. By automating these comparisons we significantly reduce the developer’s workload.

8. AREAS FOR FURTHER RESEARCH

There are other issues to be resolved, including ensuring that state information is correctly maintained where necessary for a given workflow. But the most important aspect of this idea is the possibility that we can relieve the developer of the need to generate and maintain the semantic metadata that is necessary to enable workflow composition. As the generation and maintenance of this metadata is time-consuming and expensive, the ability to automate that generation would overcome a significant obstacle to making dynamically reconfigurable workflows a reality.

Additionally, "learning ontologies" that adapt their content and structure as new information becomes available is an interesting possibility that merits further study. Just as a person's perception of the world changes as they learn new information comes, our representations of our knowledge should change. Ontologies that learn and adapt would add to our ability to create systems that learn and adapt as new services become available.

9. REFERENCES

1. Akkiraju, R., Farrell, J., Miller, J., Nagarajan, M., Sheth, A., and Verma, K. *Web Service Semantics - WSDL-S*. Thomas J. Watson Research Center, Yorktown Heights, NY, 2006.
2. Alamri, A., Eid, M., and El Saddik, A. Classification of the state-of-the-art dynamic web services composition techniques. *International Journal of Web and Grid Services 2*, 2 (2006), 148-166.
3. Barros, A., Dumas, M., and Oaks, P. A Critical Overview of the Web Service Choreography Description Language (WS-CDL). *BPTrends Newsletter 3*, 2005.
4. Bellifemine, F., Poggi, A., and Rimassa, G. JADE: a FIPA2000 compliant agent development environment. *Proceedings of the fifth international conference on Autonomous agents*, ACM (2001), 216-217.
5. Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. Web Services Description Language (WSDL) 1.1. 2001. http://www.w3.org/TR/wsdl.
6. Coenen, A. SOA Agility in Practice. 2008.
7. Curbera, F., Khalaf, R., Mukhi, N., Tai, S., and Weerawarana, S. The next step in Web services. *Commun. ACM 46*, 10 (2003), 29-34.
8. Dill, S., Eiron, N., Gibson, D., et al. SemTag and seeker: bootstrapping the semantic web via automated semantic annotation. *Proceedings of the 12th international conference on World Wide Web*, ACM (2003), 178-186.
9. Dong, X., Halevy, A., Madhavan, J., Nemes, E., and Zhang, J. Similarity search for web services. *Proceedings of the Thirtieth international conference on Very large data bases - Volume 30*, VLDB Endowment (2004), 372-383.
10. Haller, A., Cimpian, E., Mocan, A., Oren, E., and Bussler, C. WSMX - a semantic service-oriented architecture. *Web Services, 2005. ICWS 2005. Proceedings. 2005 IEEE International Conference on*, (2005), 321-328 vol.1.
11. Heß, A. and Kushmerick, N. Machine Learning for Annotating Semantic Web Services. Association for the Advancement of Artificial Intelligence (2004).
12. Hoffman, T. Case study: Sabre's Web services journey. *Java World*, 2007. http://www.javaworld.com/javaworld/jw-01-2007/jw-0108-sabre.html.
13. Howard, R. and Kerschberg, L. Using Facets of Security within a Knowledge-based Framework to Broker and Manage Semantic Web Services. (2004).
14. Howard, R. and Kerschberg, L. A knowledge-based framework for dynamic semantic Web services brokering and management. *International Workshop on Web Semantics - WebS 2004*, (2004), 174-178.
15. Howard, R. and Kerschberg, L. A Framework for Dynamic Semantic Web Services Management. *International Journal of Cooperative Information Systems, Special Issue on Service Oriented Modeling 13*, 4 (2004), 441-485.
16. Jun Han, Kowalczyk, R., and Khan, K. Security-Oriented Service Composition and Evolution. *Software Engineering Conference, 2006. APSEC 2006. 13th Asia Pacific*, (2006), 71-78.
17. Kerschberg, L. and Jeong, H. Just-in-Time Knowledge Management. In *WM 2005, K.-D. Althoff, et al., Editors*. Springer, Berlin Heidelberg, 2005, 1-18.
18. Kerschberg, L., Jeong, H., and Kim, W. Emergent Semantics in Knowledge Sifter: An Evolutionary Search Agent based on Semantic Web Services. In *Journal of Data Semantics VI*. Springer, Heidelberg, 2006, 187-209.
19. Kerschberg, L., Jeong, H., Song, Y.U., and Kim, W. A Case-Based Framework for Collaborative Semantic Search in Knowledge Sifter. *Proceedings of the 7th international conference on Case-Based Reasoning: Case-Based Reasoning Research and Development*, Springer-Verlag (2007), 16-30.
20. Klusch, M., Fries, B., and Sycara, K. Automated semantic web service discovery with OWLS-MX. *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, ACM (2006), 915-922.
21. Kona, S., Bansal, A., and Gupta, G. Automatic Composition of SemanticWeb Services. *Web Services, 2007. ICWS 2007. IEEE International Conference on*, (2007), 150-158.
22. Kourtesis, D. and Paraskakis, I. Combining SAWSDL, OWL-DL and UDDI for Semantically Enhanced Web Service Discovery. In *The Semantic Web: Research and Applications*. 2008, 614-628.
23. Krill, P. Microsoft, IBM, SAP discontinue UDDI registry effort. *InfoWorld*, 2005. http://www.infoworld.com/d/architecture/microsoft-ibm-sap-discontinue-uddi-registry-effort-777.
24. Larvet, P., Christophe, B., and Pastor, A. Semantization of Legacy Web Services: From WSDL to SAWSDL. *Internet and Web Applications and Services, 2008. ICIW '08. Third International Conference on*, (2008), 130-135.
25. Mandell, D. and McIlraith, S. Adapting BPEL4WS for the Semantic Web: The Bottom-Up Approach to Web Service Interoperation. In *The SemanticWeb - ISWC 2003*. 2003, 227-241.
26. Martin, D., Paolucci, M., McIlraith, S., et al. Bringing Semantics to Web Services: The OWL-S Approach. In *Semantic Web Services and Web Process Composition*. 2005, 26-42.
27. McIlraith, S., Son, T., and Honglei Zeng. Semantic Web services. *Intelligent Systems, IEEE 16*, 2 (2001), 46-53.
28. Muthaiyah, S., Barbulescu, M., and Kerschberg, L. A Hybrid Similarity Matching Algorithm for Mapping and Upgrading Ontologies via a Multi-Agent System. (2008).
29. Muthaiyah, S., Barbulescu, M., and Kerschberg, L. An Improved Matching Algorithm for Developing a Consistent

Knowledge Model across Enterprises Using SRS and SWRL. *Hawaii International Conference on System Sciences*, IEEE Computer Society (2009), 1-9.
30. Muthaiyah, S. and Kerschberg, L. A Hybrid Ontology Mediation Approach for the Semantic Web. *International Journal of E-Business Research 4*, 4 (2008), 79-91.
31. Paolucci, M., Srinivasan, N., Sycara, K., and Nishimura, T. Towards a Semantic Choreography of Web Services: From WSDL to DAML-S. *Proceedings of the International Conference on Web Services (ICWS 2003)*, (2003), 22-26.
32. Paolucci, M., Wagner, M., and Martin, D. Grounding OWL-S in SAWSDL. In *Service-Oriented Computing – ICSOC 2007*. 2009, 416-421.
33. Patil, A.A., Oundhakar, S.A., Sheth, A.P., and Verma, K. Meteor-s web service annotation framework. *Proceedings of the 13th international conference on World Wide Web*, ACM (2004), 553-562.
34. Ringelstein, C., Franz, T., and Staab, S. The Process of Semantic Annotation of Web Services. In *Semantic Web Services: Theory, Tools and Appplications*. Informatin Science Reference, Hershey, PA, 2007, 350.
35. Rodriguez, M.A., Bollen, J., and Sompel, H.V.D. Automatic metadata generation using associative networks. *ACM Trans. Inf. Syst. 27*, 2 (2009), 1-20.
36. seekda. com, seekda. com, and seekda. com. XigniteGlobalQuotes - Web Service Details @ seekda.com. http://seekda.com/providers/xignite.com/XigniteGlobalQuotes.
37. seekda. com, seekda. com, and seekda. com. ForeignExchangeRates - Web Service Details @ seekda.com. http://seekda.com/providers/strikeiron.com/ForeignExchangeRates.
38. Skogan, D., Groenmo, R., and Solheim, I. Web service composition in UML. *Enterprise Distributed Object Computing Conference, 2004. EDOC 2004. Proceedings. Eighth IEEE International*, (2004), 47-57.
39. Thiagarajan, R., Stumptner, M., and Mayer, W. Semantic Web Service Composition by Consistency-Based Model Refinement. *Asia-Pacific Service Computing Conference, The 2nd IEEE*, (2007), 336-343.
40. Verma, K. and Sheth, A. Semantically Annotating a Web Service. *Internet Computing, IEEE 11*, 2 (2007), 83-85.
41. Yoon, J. and Kerschberg, L. A Functional Approach to XML-Based Dynamic Negotiation in E-Business. In *The Functional Approach to Data Management, P.M.D. Gray, et al., Editors*. Springer, Heidelberg, 2003, 393-416.