Composition-driven Filtering and Selection of Semantic Web Services

Evren Sirin and **Bijan Parsia** and **James Hendler**
{evren@cs.umd.edu, bparsia@isr.umd.edu, hendler@cs.umd.edu}
MINDSWAP Research Group,
University of Maryland,
College Park MD 20742, USA

Abstract

Creating novel functionality by means of the composition of existing web services is essential for a variety of applications. One of the main problems for automated web service composition is the gap between the concepts people use and the data computers interpret. We present an interactive approach that overcomes this barrier using Semantic Web technologies. Our approach uses contextual information to find the matching services at each step of the composition. The matches are filtered using ontological reasoning on the semantic descriptions of the services. We have developed a prototype based on these ideas and tested our system by generating OWL-S descriptions for some of the common services available on the Web, including translation, dictionary, and mapping services. Using our prototype, we demonstrate that our approach is also applicable to other network environments where devices provide their capabilities with semantic descriptions and their functionalities as web services.

Introduction

Web services are designed to provide interoperability between diverse applications. The platform and language independent interfaces of web services allow the easy integration of heterogeneous systems. Universal Description, Discovery, and Integration (UDDI) (UDDI 2000), Web Services Description Language (WSDL), (Christensen *et al.* 2001) and SOAP (W3C 2001) define standards for service discovery, description, and messaging protocols respectively.

Service Oriented Architectures tend to be component oriented with loose coupling as a systematic design emphasis. Services should not only be loosely coupled with their implementation, but they should be able to be coupled together with a minimum of difficulty so that *combinations* of services can be separated from their particular realization. Given such combinations — called service *compositions* — a service consumer can mix and match component services at will depending on service availability, quality, price, and other factor. While realizing service compositions on particular concrete services is an important task, *generating* such compositions to achieve new functionality is equally important. It is hoped that web services will be so flexible that service composition is closer to the specification of functionality than it is to programming.

An emerging industry initiatives to standardize some aspects of web service composition is the Business Process Execution Language for Web Services (BPEL4WS) effort (Curbera *et al.* 2001). BPEL4WS focuses on representing compositions in which the flow of processes and the bindings between services are known *a priori*. A more challenging problem is to compose services dynamically, that is, on demand (Benatallah *et al.* 2002). In particular, when a functionality that cannot be directly realized by existing services is required, existing services may be combined together to fulfill the request. (Masuoka, Parsia, & Labrou 2003)

The dynamic composition of services requires understanding the *capabilities* of those services (i.e., *what* they can do) and the *compatibility* of those services. A successful, executable composition combines a set of compatible capabilities in the correct way to achieve the overall goal of the composition. Full automation of composition is still the object of ongoing, highly speculative research—with little short term hope of serious victory. However, there is reason to believe that partial automation of composition, with a human controller as the most significant decision mechanism, is an achievable and useful goal. One difficulty is the gap between the concepts people use and the data formats computers manipulate. We can bridge this gap using Semantic Web technologies.

The Semantic Web (Berners-Lee, Hendler, & Lassila 2001) is an extension of the current Web in which information is given well-defined meaning, better enabling computers and people to work in cooperation. Structured information is provided by marking up content in a reasonably expressive markup language with a well-defined semantics. The Web Ontology Language (OWL) (Dean *et al.*) is a forthcoming W3C specification for such a language that will supersede the earlier DARPA Agent Markup Language (DAML+OIL) (Horrocks *et al.* 2001). OWL is an extension to the Resource Description Framework (RDF) (Brickley & Guha), enabling the creation of ontologies for arbitrary domains and the instantiation of these ontologies in the description of resources. The OWL-services language (OWL-S) [1] (OWL Services Coalition 2004) is a set of on-

Copyright © 2004, American Association for Artificial Intelligence (www.aaai.org). All rights reserved.

[1]The previous version of OWL-S was called DAML-S and was

tologies supporting the rich description of web services for the Semantic Web. Our work uses OWL and OWL-S to facilitate the user- and context-driven, dynamic composition of web services.

Interactive Composition Approach

We present a goal-directed approach for the composition of services where the composition is gradually generated with a forward or backward chaining of services. At each step, a new service is added to the composition and further possibilities are filtered based on the current context and user decisions. Here we show how this approach can be utilized to make the necessary travel arrangements for a trip. The first step is to book a means of transportation. You start by finding the services that let you make reservations for transportation. Then you need to filter these services because not all of the services are relevant to your current task—e.g. ones that does not provide transportation to your destination or ones that have no availability at the desired dates should not need to be considered. Filtering may be further used to help determine the service that best fits for your personal preferences, such as accepting a certain credit card or serving particular destinations with non-stop flights. After this step is resolved, you can continue the composition process by finding compatible services. Perhaps you have a clear idea of what further tasks you'd like to accomplish with this composition or perhaps just seeing the available, compatible services will suggest further goals. Just as with business or consumer services, we expect propinquity to be a key factor in determining desirable compositions, particularly when the "extra" services aren't strict requirements of the current task.

We have developed a service composition prototype that guides the user in creating a workflow of services in a step by step style as described above. Selection of services is done in the context of a composition step. When a service is put into the composition, the information about input, output, preconditions, and effects (IOPE) of this service is used to automatically filter out the services whose outputs are not compatible with the current selection. We support further, user-driven filtering of the compatible services based on other features of the services described against generally available OWL ontologies.

Service composition in our system is based on semantic annotations of services. As an example of how semantic descriptions aids the composition process, consider a simple scenario with two web services, an on-line language translator and a dictionary service—where the first one translates text between several language pairs and the second one returns the meanings of English words. If a user needs a *FrenchDictionary* service, neither of these can satisfy the requirement. However, together they can satisfy it: the input can be translated from French to English, fed through the English Dictionary, and then translated back to French. The dynamic composition of such services is difficult using just their WSDL descriptions, since each description would designate strings as input and output, rather than the necessary concept for combining them. In other words, some of these input strings must be the names of languages, others must be the strings representing user inputs and the translator's outputs. To describe the specific concepts like *language* or *French*, we can use ontologies published on the Semantic Web.

Service composition can also be used in linking concepts to services provided in other network environments. One example is a sensor network environment which includes two types of services: basic sensor services and sensor processing services. Each sensor is related to one web service which returns the sensor data as the output. Sensor processing services combine the data coming from different sensors in some way and produce a new output. A sensor ontology describes sensor capabilities – sensitivity, range, and so on – as well as other significant attributes like name, location, etc. These attributes, taken together, tell whether the sensor's service is relevant for, say, generating a fusion of data from a variety of services positioned in a certain way relative to each other. The fused data itself might be passed to feature extracting or pattern recognition services, with the ultimate results being used to identify particular objects in the environment. In this setting, we need to describe the services that are available for combining sensors and the attributes of the sensors that are relevant to those services. More importantly, the user needs a flexible mechanism for filtering sensor services and combining only those that can realistically be fused (e.g., those covering the same physical location).

Creating Semantic Service Descriptions

OWL-S partitions the semantic description of a web service into three components: the service profile, process model, and grounding. The *ServiceProfile* describes what the service does by specifying the input and output types, preconditions and effects. The *ProcessModel* describes how the service works; each service is either an *AtomicProcess* that is executed directly or a *CompositeProcess* that is a combination of subprocesses (i.e., a composition). The *Grounding* contains the details of how an agent can access a service by specifying a communications protocol, parameters to be used in the protocol, and the serialization techniques to be employed for the communication. The similarities between OWL-S and other technologies may be briefly expressed as follows. The ServiceProfile is analogous to yellow-page-like advertisements in UDDI, the ProcessModel is similar to the business process model in BPEL4WS, and the Grounding is a mapping from OWL-S to WSDL. The main contribution of OWL-S is the ability to support richer descriptions of the services and the real world entities they affect in such a way as to support greater automation of the discovery and composition of services.

OWL-S service descriptions are made to link to other ontologies that describe particular service types and their features. For example, an ontology can be written in OWL that is specialized for the description of sensors. This ontology contains a top level class *Sensor* to define the sensor concept. *Sensor* has subclasses such as *AcousticSensor* and *InfraRedSensor*. In the semantics of OWL, subclasses inherit

based on DAML+OIL

Making Travel Arrangements
1. Book transportation
1.1. Find transportation services
1.2. Filter out the services which has no availability at the desired dates
1.3. Select a service that accepts your credit card, offers a good price, etc.
2. Make hotel reservation (feed date of arrival information from previous service to this one)
...
3. Record expenses in your financial organizer (compute total of expenses from previous steps)

Table 1: A step by step composition of a service that will make the travel arrangements for a trip

the properties of their superclasses and may extend these attributes with additional ones. Since OWL-S service descriptions are nothing more than OWL documents, we can use all the domain modeling features of OWL to directly structure our service descriptions, as well as freely using concepts from other ontologies. For example, for our prototype, we developed a hierarchy of ServiceProfile types. This class tree, rooted in the ServiceProfile class, paralleled and made use of our sensor ontology in the obvious ways; e.g., sensors provide *Sensor* services which have *SensorServiceProfiles* and acoustic sensors provide *AcousticSensor* services which have *AcousticSensorProfiles*. We specialize ServiceProfiles rather than Services themselves because our primary interest is service selection and matchmaking which, in our system, is done using ServiceProfiles. In particular, we use the ServiceProfile hierarchy to help the user filter irrelevant services. If, at a certain composition step, all the system knows is that some sort of sensor service can be selected, the human user can determine the more precise requirements by examining the range of available sensor types. The sensor ServiceProfiles also have specific sets of non-IOPE parameter related attributes associated with them, defined using the extensible "service parameter" mechanism in OWL-S. We can use these attributes to define even more specific named classes of ServiceProfiles such as "NearbyAcousticSensors", or simply allow the user to define complex class expressions expressing their requirements on the fly.

In OWL-S service descriptions, information on how to execute the services is specified in the *Grounding*. In most Groundings, execution information consists of pointers into WSDL descriptions, which typically contain sufficient information to directly invoke the described service. There is an increasing number of WSDL-described web services available on the Web, both from independent developers and large companies (e.g., Amazon and Google). Annotating these web services with OWL-S provides a good opportunity for us to access a lot of semantically described, executable services.

Some aspects of deriving OWL-S descriptions from WSDL descriptions can be partially automated. For each *operation* a WSDL document describes, the document will provide a description of the input and output messages and their substructure for that operation. Normally we take a WSDL operation to correspond to an OWL-S AtomicPro-

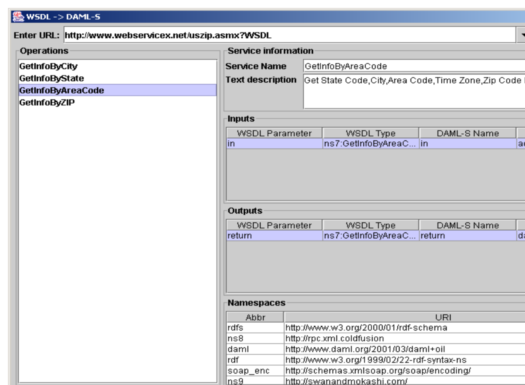

Figure 1: A tool to automate translation from WSDL descriptions to OWL-S

cess, with the parameters of that process corresponding to various message parts. In nearly all WSDL documents, the content of message parts are described by XML Schema datatypes, quite often complex types (that is, types which describe elements with possible attribute or subelement structure). Since parameter type compatibility is a critical part of our composition method, it is very important that the service description supplies sufficiently expressive types. There are two issues that arise when trying to incorporate XML Schema datatypes in OWL-S service descriptions. OWL itself permits only a constrained range of XML Schema datatypes; for those it does permit, it provides constructs and reasoning services which aren't nearly as intersting as those it provides for OWL classes.

In particular, there is currently only provision in OWL for defining properties whose range is one of a subset of the simple XML Schema datatypes (such as integers, strings, etc.), but there is no provision for using complex types. This is not due to any logical difficulty with integrating complex types with OWL, but rather due to the fact that the way OWL-DL references datatypes is by URIs, and there is no canonical way in XML Schema to determine a URI for a complex datatype. So only those datatypes with predefined URIs can be used in OWL documents. We expect this situation will eventually be resolved by the XML Schema working group. Until then, one cannot even expect OWL reasoners with ex-

cellent datatype to process complex datatypes interoperably.

Even when this problem is resolved, the fact remains that for many purposes it would be preferable to have the parameter types of OWL-S services be OWL classes, as it would allow for more flexible matching and more natural OWL-based descriptions. Since we are already augmenting the information in a WSDL description, it seems reasonable to do so with the types as well. Thus, we treat the WSDL supplied types as descriptions of the "wire format" of the service parameters, that is, the serialization of the values actually used by our process. We extended the OWL-S Grounding to all for the inclusion of marshaling and unmarshaling functions which our OWL-S executor can use to coerce XML Schema values to OWL individuals and back.[2] These functions are, by default, encoded as XSLT stylesheets. For example, an unmarshaling function is written as an XSLT transformation from XML fragments matching the specific XML Schema type to an RDF graph serialized in the RDF/XML exchange syntax. That graph encodes the relevant assertions about the individual which is the actual input to the service. Marshalling functions are implemented as the inverse transformation. Using published XSLT obviates the need for the OWL-S executor to be extended with specific type coercion functions — it just needs a generic XSLT processor, perhaps running as a remote service. The downside is that due to the extremely free syntax of RDF/XML (especially, the plurality of equivalent forms), it is difficult to write XSLT that can handle all the legal serializations of a given RDF graph, and the resulting stylesheet is difficult to understand and maintain.

Clearly, writing such transformation functions by hand is not feasible. Marshalling and unmarshaling functions already can be a source of subtle bugs as they require a deep understanding of both source and target formalism, a good understanding of the services both on the WSDL side (i.e., of the operational semantics of the service) and on the OWL-S side (i.e., of how the descriptions affect the various of OWL-S related inferences). Adding essentially irrelevant and idiosyncratic details of a specific linear syntax for RDF compounds the problem. Unfortunately, current standard solutions tend to compromise interoperability. In our system, since we control all our execution engines (in fact, we reuse a single implementation), we can require a specific profile of RDF/XML that avoids confusing or redundant constructs. Clearly if other engines do not generate that profile, then our XSLT transformations can fail. Also it is uclear that, even with a suitably designed profile, the necessary XPath queries will be sufficiently obvious and transparent to the programmer. Finally, while feeding the XSLT processor some XML allows for great flexibility, both in choice of implementation of processor and of the specific instance of some processor, it is unlikely that the internal representation of the individual will be, say, W3C DOM trees, so there is the constant need for additional data conversion.

All three issues would be dealt with by the incorporation of an RDF and OWL sensitive query language (such as RDQL or Versa) into the XSLT, or perhaps XQuery, standards.[3] Even if generic XSLT or XQuery processors generally failed to include such extension, it would provide a standard and appealing target for OWL-S engines to implement; and, even if the query languages weren't ideal, they would have both less of a conceptual gap and less of an implementation gap than XPath queries.

An appealing alternative to either technique is to use a higher level mapping language, perhaps along the lines of MDL (Worden 2002) as proposed in (Peer 2002). If the mappings could be compiled to XSLT or other transformation languages, there would be an enormous gain in portability, and by eschewing the general expressive power of programming languages like XSLT, there might be a significant gain in transparency and analyzability. Unfortunately, the design of such a language covering the entire expressivity of OWL is a formidable task.

Implementation

We have built a system to provide support for our interactive composition approach using semantic service descriptions. Filtering and selection of services is achieved by using matchmaking algorithms similar to those implemented in the DAML-S Matchmaker (Paolucci *et al.* 2002). In that system matchmaking uses ServiceProfiles to describe service requests as well as the services advertised. A service provider publishes a DAML-S (or, presumably in a successor, updated Matchmaker, OWL-S) description to a common service repository. When someone needs to locate a service to perform a specific task, a ServiceProfile for the desired service is created. Request profiles are matched by the service registry to advertised profiles using OWL-DL subsumption as the core inference service. In particular, the DAML-S Matchmaker computes subsumption relations between each individual IOPE's of the request and advertisement ServiceProfile. If the classes of the corresponding parameters are equivalent, there is an exact and thus best match. If there is no subsumption relation, then there is no match. Given a classification of the types describing the IOPEs, the Matchmaker assigns a rating depending on the number of intervening named classes between the request and advertisement parameters. Finally, the ratings for all of the IOPEs are combined to produce an overall rating of the match. (Gonzalez-Castillo, Trastour, & Bartolini 2002) and (Li & Horrocks 2003) extended this algorithm to consider the subsumption relation between the request and advertisement profiles considered as whole concepts. But like the DAML-S Matchmaker, they then map the subsumption relation into goodness of match, specifically:

- **Exact** If advertisement A and request R are equivalent concepts, it is called an Exact match

[2]These extensions, with further development by the OWL-S coalition, were subsequently included in OWL-S. These extensions and their implementation were done in collaboration with Fujitsu Labs of America, College Park, with extensive feedback from Ryusuke Masuoka.

[3]The 4Suite XSLT parser already integrates the Versa RDF query language as an XSLT extension (4su).

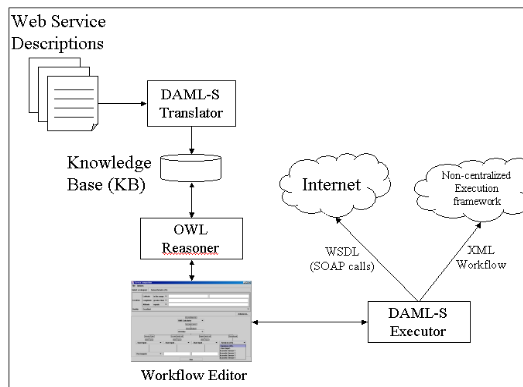

Figure 2: Overview of prototype system

- **PlugIn** If request R is sub-concept of advertisement A, it is called a PlugIn match
- **Subsume** If request R is super-concept of advertisement A, it is called a Subsume match
- **Fail** Otherwise, there is no match

Our system uses the same basic typology of subsumption based matches, but in some contexts we match based on the subsumption of the entire profiles, and in other contexts we use subsumption only to directly match individual parameters. (See the section entitled "Matching on IOPEs" for a discussion of the latter, and the section entitled "Filtering on Service Parameters" for the former.)

The general system architecture is shown in Figure 2. The system has two separate components. An inference engine is responsible for storing service advertisements and processing match requests. The inference engine is an OWL-DL reasoner named Pellet (Pellet 2003). The other component of the system is the composer where the workflow of service composition is generated. The composer communicates with the inference engine to discover possible matches and present them to the user. It also lets users to invoke the completed composition on specific inputs.

The composer lets the user create a workflow of services by presenting the possible choices at each step. The user is first presented with all the available services registered to the system. This first step is totally unguided. Each subsequent step of the composition makes use of two sorts of matching, on IOPEs (which is fully automated) and on other service parameters. Forms for entering constraints on the service parameters are generated from the ontologies defining those parameters. In any step, the final selection of the specific service is done by the user.

Matching on IOPEs

At each step of the composition, a list shows the IOPE compatible services that can be added to the composition.[4] When a service is selected from the list, the composer presents as options those services whose output could be fed to the current service as an input. Suppose the selected service accepts an input of type *Address* which is defined in a certain ontology where the concept hierarchy is shown in Figure 3. We would like to find the services which have an output that is compatible with this type. An output of a service would be considered compatible if it was of type *Address* or another concept which is subsumed by *Address*, i.e. *USAddress*. When the output of a service is subsumed by the input, the output type can be viewed as a specialized version of the input type and these services can still be chained together.[5] However, a service whose output is *Location* could not be composed with this service since *Address* concept will most likely have additional properties and restrictions on the existing properties of *Location*.

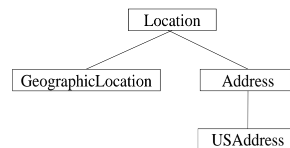

Figure 3: A simple hierarchy of location related concepts

Clearly, only Exact and PlugIn matches between the parameters of ServiceProfiles would yield useful results at this step. For service selection, we need match on individual parameters types instead of whole profiles, as we consider all type compatible services to be reasonable "next steps" of a composition. One interesting extension would be to consider certain service parameters against global constraints as part of service compatibility. For example, suppose before starting the composition process, the user enters an overall price limit on the composition. At any step, the system sums the values of all cost service parameters of the currently composed services, and uses the difference between that sum and the set limit to filter potential next steps.

The ordering of the result displayed in the list is based on the degree of the match. The Exact matches are more likely to be preferred in the composition and these services are displayed at the top of the list. The PlugIn matches are presented after the Exact matches and PlugIn matches are ordered according to the distance between the two types in the ontology tree.

Filtering on Service Parameters

The number of services displayed in the list as possible matches can be extremely large. For example, a power grid or telephone network might have many thousands of sensors each providing several services. This will make it infeasible for someone to scroll through a list and choose one of the services simply by name. Furthermore, even if the number of services is low, the service names themselves may not be contain enough information to let a user know what they do.

[4]It should be noted that currently we only match on IO, because the specification of preconditions and effects is still an open OWL-S issue.

[5]One reviewer worried that an Address consuming service might not be able to handle USAddresses. But *by definition* every USAddress is an Address, so a service that advertises itself as consuming Addresses better handle USAddresses as well.

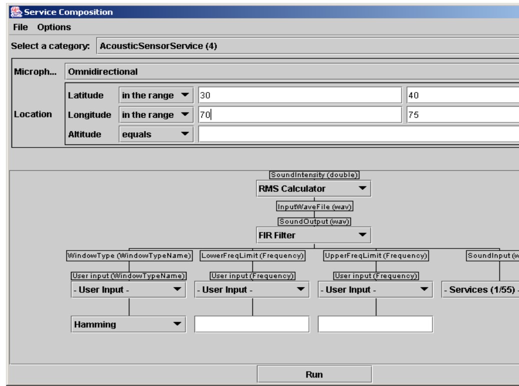

Figure 4: Filtering is used to see only omnidirectional acoustic sensors that are located at a latitude between 30-40 and a longitude between 70-75. It is seen that only one of 55 services satisfy these constraints

When the name of the service does not help to distinguish the services, we turn to the other service parameters, such as location, to help determine the most relevant service for the current task. Thus, a sensor description, linked to a particular service, can be queried as to the sensor's location, type, deployment date, sensitivity, etc.

The ServiceProfile hierarchies defines a classification which is used at the first level of filtering. By selecting a profile category from the list, user limits the shown available choices whose ServiceProfile matches with the selection. We examine the definitions of the various ServiceProfiles to build various user input forms for specifying further constraints on the desirable services.

Consider a example in the sensor network where we want to select a specific sensor service. With no other restriction, the system will present every available sensor service. This is better than presenting all the services, but the remaining choices can still be overwhelming. If the user chooses to filter the results to the services with *AcousticSensorServiceProfiles*, that decreases the number of matches significantly. The composer then queries the inference engine about the possible service parameters of the selected service type. Based on the answer returned from the engine, the composer creates a GUI panel in which the user can enter constraints for the properties of the services as shown in Figure 4.

The user's constraints are combined in a service request profile. The service request is sent to the inference engine and the result of this new query is applied to the previous result set. The services that do not satisfy the current constraints are removed from consideration. The matchmaking for this step can use Relaxed matches as well as Exact and PlugIn matches. Using Relaxed matches will probably increase the choices presented allowing the user to make a more flexible selection. Relaxed matches are permissible because we already know that the set of services the user is considering are compatible in this context.

Generating Composed Services

Each composition generated by the user using the existing prototype can itself be realized as a OWL-S CompositeProcess, thus allowing it also to be advertised, discovered, and composed with other services. In the composer, we generate exactly such a CompositeProcess description and also create the corresponding ServiceProfile with user added non-functional properties. Such a description is immediately available to the system as a named service which can be filtered and composed in the normal way. In this way, the user can quickly build up a set of complex compositions in a piece-meal fashion as the tasks at hand demand.

Execution of Composed Services

The current implementation of the system executes the composition by invoking each individual service and passing the data between the services according to the flow constructed by the user. This method is primarily dictated by the OWL-S and WSDL specifications (of the time) which both describe the web services as an interaction of either a request/response or as a notification messaging between two parties. As a consequence of this design, the client program serves as the central control authority that handles all the RPC calls to invoke individual services. Future iterations of both specifications are likely to include more complicated arrangements of parties.

Improving IOPE Matching with Ontology Translation Services

With both IOPE matching and service parameter filtering there is a strong need for a suitable set of service descriptions of sufficient and compatible detail to support, for IOPE matching, the appropriate subsumptions and, for service parameter filtering, intelligible form based queries. It is straightforward to elaborate the service parameter filter forms by extending the definitions of the concepts used to describe those parameters. We expect that such extension will be done using standard ontology editing tools.

We have already discussed improving IOPE matching by converting the IO type descriptions from XML Schema datatypes to OWL classes. In that process, the choice of target OWL class is critical to generating matchmaking hits. The Semantic Web is likely to have a large number of somewhat overlapping ontologies, that is, ontologies which have fairly similar, but distinct concepts. If service description authors choose different, but relevantly equivalent, classes to unmarshall their XML Schema datatypes to, the system will fail to match intuitively compatible services. Ideally, some sort of concept or ontology mapping would make these relevant equivalences transparent to the system. Aside from the normal OWL-DL constructs for equating classes, we have the concept of a *TranslatorServiceProfile*, that is, of services whose entire job is to take the description of an OWL individual against one ontology, and produce the relevantly equivalent set of assertions against another.

However, there is an important sense in which these services are unimportant to the composition process. Rather,

they are *only* important insofar as they promote the composition of other services which actually move the user closer to her goal. They are not suggestive of interesting further steps, thus are merely a burden on the user. To eliminate this, we do not actually present the translation services to the user, but rather created "fused" services on the fly. A fused service is a chain of translation services terminating in a non-translation service. The fused service is presented as a type compatible non-translation service, thus increasing the number of substational options at any particular step.

Related Work

There are several different industry efforts to create support web service composition, with the Business Process Execution Language for Web Services (BPEL4WS) (Curbera *et al.* 2001) being perhaps the most prominent. BPEL4WS supersedes IBM's Web Services Flow Language (WSFL) and Microsoft's XLANG. BPEL4WS provides a language for the formal specification of business processes and business interaction protocols. It extends the interaction model of WSDL to define a process that provides and consumes multiple Web Service interfaces. Such a process can be thought of as composing a set of web services from other web services. While compositions built with our composer presumably could be "compiled" to BEPL4WS workflows, it is important to recognize that BEPL4WS is more like an extended WSDL than an entirely distinct layer. In our prototype, we did not support use of the more complex OWL-S control constructs (which are difficult to present in a intuitive way), or even of preconditions and effects, which provide an automation friendly approach to state management. Thus, the motivation to move from the XML Schema type system to OWL remains even if the control flow is ultimately described by BPEL4WS

The DAML-S Matchmaker (Paolucci *et al.* 2002) is a system to augment current UDDI architecture with semantic service descriptions. The Matchmaker aims to improve the discovery process by allowing location of services based on their capabilities in support the composition task. The basic idea used in the Matchmaker is making use of the subsumption relation between the classes to find flexible matchings beyond the capabilities of UDDI. (Gonzalez-Castillo, Trastour, & Bartolini 2002) and (Li & Horrocks 2003) extends the matchmaking algorithms to exploit more features of subsumption relations. (Li & Horrocks 2003) also reports some problems related to the design of OWL-S Profile specification. Our work utilizes slightly evolved versions of methods presented in these works directly in a composition editor. In a sense, our work is about effectively using matchmaking *while* composing services, to help guide the composition process.

Peer (Peer 2002) addresses the problem of mapping XML Schema types to OWL classes. The author suggests to use Meaning Definition Language (MDL) to map structural data types to their semantics. MDL is an XML based language which allows to explicitly define how the structures of an XML data type conveys meanings that referenced is contained in ontologies expressed by languages like OWL.

Conclusion and Future Work

In this work, we have shown how to use semantic descriptions to aid in the composition of web services. We presented a goal-oriented, interactive composition approach where matchmaking algorithms are utilized to help users filter and select the services while building the composition. Indeed, it is the filtering and selection of services that helps the user drive the composition process. We have implemented these ideas in the prototype system and shown that it can compose the actual web services deployed on the Internet as well as providing filtering capabilities where a large number of similar services may be available. Our prototype is the first system to directly combine the OWL-S semantic service descriptions with actual invocations of the WSDL descriptions allowing us to execute the composed services on the Web.

The service composition in the current framework requires a human in the loop. A human who has the domain knowledge for the task should guide the overall composition process where the composer aids the operator by providing the relevant choices at each step. The incorporation of planning technology in the inference engine would result in further automation of the system. The ability to access the machine understandable data via the Semantic Web is expected to make it easier to integrate a planner into the system.

One way to improve the suggestions provided for a match is to enhance the system with the capability to learn from past user activity. The previous service compositions that were created by the operators would give an idea about the general tasks a user wants to accomplish. Using this information, the composer could reorder the available choices in the list presented or simply present a composition similar to the previous ones.

The accuracy of the matches found by the inference engine depend on how detailed the ontologies are. Richer ontologies with more specific descriptions for sensors and their non-functional properties will help the engine to find sqbetter answers to the queries. As ontologies become widely used on the Semantic Web, we expect to find an increasing number of cross references between related concepts in different ontologies (and OWL supports such cross-referencing directly) and thus the impact of semantic information will become more apparent.

Acknowledgements

This work is supported in part by the Army Research Laboratory, the Defense Advanced Research Projects Agency, Fujitsu Laboratory of America College Park, Lockheed Martin Advanced Technology Laboratories, the National Science Foundation, the National Institute of Standards and Technology, and NTT Corp.

References

4Suite - XML and RDF Toolkit. http://www.4suite.org/.

Benatallah, B.; Dumas, M.; Fauvet, M.-C.; and Rabhi, F. 2002. Towards Patterns of Web Services Composition. In Gorlatch, S., and Rabhi, F., eds., *Patterns and Skeletons for Parallel and Distributed Computing*. UK: Springer Verlag.

Berners-Lee, T.; Hendler, J.; and Lassila, O. 2001. The Semantic Web. *Scientific American* 284(5):34–43.

Brickley, D., and Guha, R. Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation submitted 22 February 1999 http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/. (current May 2002).

Christensen, E.; Curbera, F.; Meredith, G.; and Weerawarana, S. 2001. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

Curbera, F.; Goland, Y.; Klein, J.; Leymann, F.; Roller, D.; Thatte, S.; and Weerawarana, S. 2001. Business Process Execution Language for Web Services, Version 1.0. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.

Dean, M.; Connolly, D.; van Harmelen, F.; Hendler, J.; Horrocks, I.; McGuinness, D. L.; Patel-Schneider, P. F.; and Stein, L. A. Web Ontology Language (OWL) Reference Version 1.0. W3C Working Draft 12 November 2002 http://www.w3.org/TR/2002/WD-owl-ref-20021112/.

Gonzalez-Castillo, J.; Trastour, D.; and Bartolini, C. 2002. Description Logics for Matchmaking of Services. In *Workshop on Applications of Description Logics ADL 2001*.

Horrocks, I.; van Harmelen, F.; Patel-Schneider, P.; Berners-Lee, T.; Brickley, D.; Connoly, D.; Dean, M.; Decker, S.; Fensel, D.; Hayes, P.; Heflin, J.; Hendler, J.; Lassila, O.; McGuinness, D.; and Stein, L. A. 2001. DAML+OIL. http://www.daml.org/2001/03/daml+oil-index.html.

Li, L., and Horrocks, I. 2003. A Software Framework For Matchmaking Based on Semantic Web Technology. In *Proc. of the Twelfth International World Wide Web Conference (WWW 2003)*.

Masuoka, R.; Parsia, B.; and Labrou, Y. 2003. Task computing - the semantic web meets pervasive computing -. In *Proceedings of 2nd International Semantic Web Conference (ISWC2003)*.

OWL Services Coalition. 2004. OWL-S: Semantic Markup for Web Services. OWL-S White Paper http://www.daml.org/services/owl-s/1.0/owl-s.pdf.

Paolucci, M.; Kawamura, T.; Payne, T. R.; and Sycara, K. 2002. Semantic Matching of Web Services Capabilities. In *The First International Semantic Web Conference*.

Peer, J. 2002. Bringing Together Semantic Web and Web Services. In *International Semantic Web Conference 2002 (ISWC'02)*.

Pellet. 2003. Pellet - OWL DL Reasoner. http://www.mindswap.org/2003/pellet.

UDDI. 2000. The UDDI technical white paper. http://www.uddi.org/.

W3C. 2001. SOAP 1.2 - W3C Recommendation 24 June 2003. http://www.w3.org/TR/soap12-part0/.

Worden, R. 2002. Meaning Definition Language v2.06. http://www.charteris.com/XMLToolkit/MDL.asp.