Filtering and Selecting Semantic Web Services with Interactive Composition Techniques

Evren Sirin, Bijan Parsia, and James Hendler, *University of Maryland*

Building complex services by hand is notoriously difficult. At every step of a composition, users face a plethora of choices. Our assisted composition approach uses the richness of Semantic Web Service descriptions and information from the compositional context to filter matching services and help select appropriate services.

To demonstrate the utility of Semantic Web Service descriptions for service composition, we've developed a goal-oriented, interactive composition approach that uses matchmaking algorithms to help users filter and select services while building the composition. Indeed, it is the filtering and selection of services that helps the user drive the composition process. We've implemented these ideas in a prototype system that can compose the Web Services deployed on the Internet and provide filtering capabilities where a large number of similar services might be available.

As this article describes, our prototype is the first system to directly combine the OWL-S semantic service descriptions with actual invocations of the WSDL descriptions, letting us execute the composed services on the Web.

The Semantic Web meets Web Services

Web Services provide interoperability between diverse applications, using platform- and language-independent interfaces for easily integrating heterogeneous systems. Universal Description, Discovery, and Integration (UDDI, www.uddi.org), WSDL,[1] and SOAP define standards for service discovery, description, and messaging protocols, respectively.[2]

Service-oriented architectures tend to be component oriented, emphasizing loose coupling as a systematic design approach. Not only should services be loosely coupled with their particular implementation or deployment, but inter-service dependencies should be minimized to allow easy combination of services into larger systems. Given such combinations—called *service compositions*—a service consumer can mix and match component services at will, depending on such factors as service availability, quality, and price. Although realizing service compositions on particular concrete services is an important task, generating such compositions to achieve new functionality is equally important. Ultimately, Web Services should be so flexible that service composition is closer to the specification of functionality than to programming.

An emerging industry initiative to standardize some aspects of Web Service composition is the Business Process Execution Language for Web Services effort.[3] BPEL4WS focuses on representing compositions in which the flow of processes and the bindings between services are known beforehand. More challenging is the problem of composing services dynamically—on demand.[4] In particular, when service customers need a functionality that existing services can't individually realize, existing services can be combined to fulfill the request.[5]

The dynamic composition of services requires that we understand their capabilities as well as their compatibility. A successful, executable composition correctly combines a set of compatible components to achieve the composition's overall goal. Full automation of composition is still the object of ongoing, highly speculative research—with little short-term hope of serious victory. However, partial automation of com-

Related Web Service Composition

Several industry efforts are under way to create support for Web Service composition, with the Business Process Execution Language for Web Services (BPEL4WS) as perhaps the most prominent.[1] BPEL4WS supersedes IBM's WSFL and Microsoft's XLANG. BPEL4WS provides a language for scripting business processes and business interaction protocols. It extends WSDL's interaction model to define a process that provides and consumes multiple Web Service interfaces. We can think of such a process as composing a set of Web Services from other Web Services.

Compositions built with our composer presumably could be "compiled" to BPEL4WS workflows, although as you add more complex OWL-S control constructs, the mapping becomes more hairy. Also, BPEL4WS doesn't support any sort of preconditions and effects, nor is it particularly declarative. So while BPEL4WS workflows might be useful as a grounding for our OWL-S compositions (much in the way WSDL operations ground AtomicProcesses), it is unlikely that any tighter integration would be reasonable or useful.

The DAML-S Matchmaker system augments current UDDI architecture with semantic service descriptions.[2] The Matchmaker aims to improve the discovery process by allowing location of services based on their capabilities in support of the composition task. The basic idea behind the Matchmaker is to use the subsumption relation between the classes to find flexible matchings beyond the capabilities of UDDI. Several research groups extend the matchmaking algorithms to exploit more features of subsumption relations.[3,4] Lei Li and Ian Horrocks also report some problems related to the design of the OWL-S Profile specification. Our work uses 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.

Jihie Kim, Yolanda Gil, and Marc Spraragen present an interactive workflow composition tool named CAT (Composition Analysis Tool). This tool combines knowledge-based representations of components, together with planning techniques that can track the relations and constraints among components.[5] Although their tool is not using Web standards such as OWL, OWL-S, or WSDL, it is possible to extend the tool to support these languages.

References

1. F. Curbera et al., "Business Process Execution Language for Web Services, v1.0," S. Thatte, ed., IBM, July 2001; www-106.ibm.com/developerworks/webservices/library/ws-bpel.
2. M. Paolucci et al., "Semantic Matching of Web Services Capabilities," *The Semantic Web–ISWC 2002: 1st Int'l Semantic Web Conf.*, LNCS 2342, Springer-Verlag, 2002, pp. 333–347.
3. J. Gonzalez-Castillo, D. Trastour, and C. Bartolini, "Description Logics for Matchmaking of Services," *Proc. Workshop Applications of Description Logics*, CEUR, 2002.
4. L. Li and I. Horrocks, "A Software Framework for Matchmaking Based on Semantic Web Technology," *Proc. 12th Int'l World Wide Web Conf.*, World Wide Web Consortium, 2003, p. 48.
5. J. Kim, Y. Gil, and M. Spraragen "A Knowledge-Based Approach to Interactive Workflow Composition," *Proc. Int'l Conf. Automated Planning and Scheduling, Workshop Planning and Scheduling for Web and Grid Services*, AAAI Press, 2004.

position, with a human controller as the most significant decision mechanism, seems 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 "Related Web Service Composition" sidebar discusses related approaches.)

The Semantic Web extends the current Web by giving information well-defined meaning, better enabling computers and people to collaborate.[6] Users provide structured information by marking up content in a reasonably expressive markup language with a well-defined semantics. OWL[7] is a W3C recommendation for such a language (superseding the earlier DAML+OIL[8]). OWL is an extension to the RDF, which lets us create ontologies for arbitrary domains and instantiate these ontologies to describe resources.[9] OWL-S (the previous version of OWL-S was called DAML-S and was based on DAML+OIL) is a set of OWL ontologies 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

Our goal-oriented approach for service composition gradually generates the composition with a forward or backward chaining of services. At each step, our system adds a new service to the composition and filters further possibilities based on the current context and user decisions. Let's see how you can use our approach to make necessary travel arrangements.

As Figure 1 shows, the first step is to book a means of transportation. Start by finding the services that let you make reservations for transportation. Then filter these services because not all are relevant to your current task: some might not provide transportation to your destination, and others might have no availability at the desired dates. Filtering might further help determine the service that best fits your personal preferences, such as ones that accept a certain credit card or serve particular destinations with nonstop flights.

After resolving this step, 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 simply seeing the available, compatible services will suggest further goals. Just as with business or consumer services, propinquity is a key factor in determining desirable compositions, particularly when the "extra" services

```
Making Travel Arrangements
   1. Book transportation
      1.1  Find transportation services
      1.2  Filter out the services that have 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 the total expenses
      from previous steps)
```

Figure 1. A step-by-step composition of a service that will make travel arrangements for a trip.

aren't strict requirements of the current task.

We've developed a service composition prototype that guides users in creating a workflow of services step by step as just described. Users select services in the context of a composition step. When a service goes into the composition, this service's information about input, output, preconditions, and effects (IOPE) serves to automatically filter the services whose outputs are incompatible with the current selection. We support further, user-driven filtering of the compatible services based on other service features described against generally available OWL ontologies.

Service composition in our system relies on semantic annotations of services. As an example of how semantic descriptions aid the composition process, consider a simple scenario with two Web Services—an online language translator and a dictionary service—where the first translates text between several language pairs and the second returns the meanings of English words. If a user needs a FrenchDictionary service, neither 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 because 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 the strings representing user inputs and the translator's outputs. To describe the specific concepts such as language or French, we can use ontologies published on the Semantic Web.

Service composition can also serve in linking concepts to services provided in other network environments. For example, take a sensor network environment that includes two types of services: basic sensor services and sensor-processing services. Each sensor relates to one Web Service that 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, such as name or location. Taken together, these attributes indicate whether the sensor's service is relevant for, say, generating a fusion of data from various services positioned in a certain way relative to each other.

The fused data itself might pass to feature-extracting or pattern-recognition services, with the ultimate results serving to identify particular objects in the environment. In this setting, we need to describe the services that are available for combining sensors and the sensor attributes 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—those covering the same physical location.

Creating semantic service descriptions

OWL-S partitions a Web Service's description into three components: 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 executes directly or a CompositeProcess that combines subprocesses—a composition. The Grounding contains the details of how an agent can access a service by specifying a communications protocol, parameters to use in the protocol, and serialization techniques to be employed for the communication.

Annotating these Web Services with OWL-S gives us a good opportunity to access several semantically described, executable services.

OWL-S resembles other technologies several ways:

- The ServiceProfile is analogous to yellow-page-like advertisements in UDDI.
- The ProcessModel is similar to the business process model in BPEL4WS.
- The Grounding is a mapping from OWL-S to WSDL.

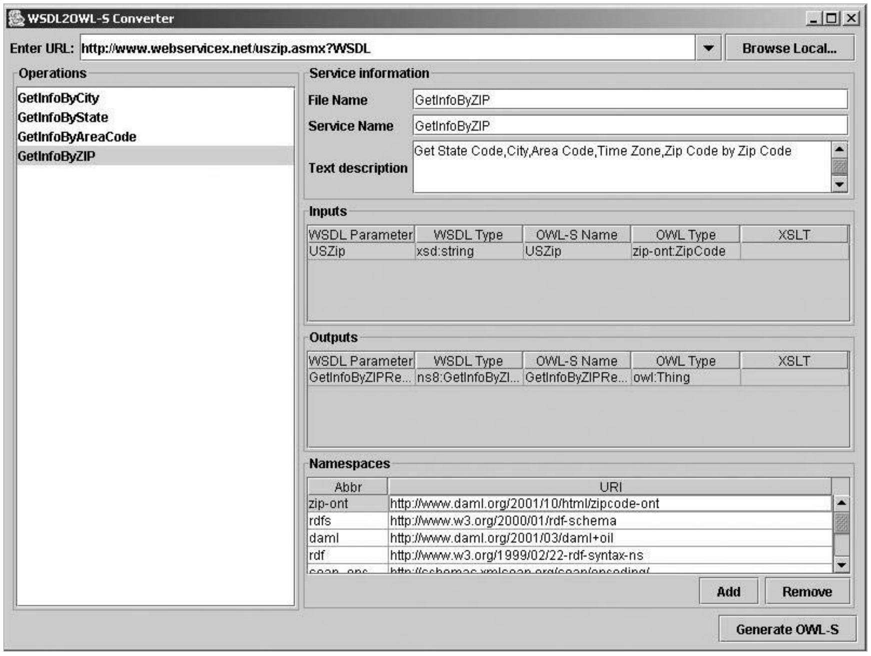
OWL-S's main contribution is its ability to support richer service descriptions and the real-world entities they affect sufficient for greater automation of the discovery and composition of services.

OWL-S service descriptions link to other ontologies that describe particular service types and their features. For example, suppose you've written an ontology in OWL for describing sensors. This ontology contains a top-level class Sensor to define the sensor concept. Sensor has subclasses such as AcousticSensor and InfraRedSensor. In OWL's semantics, subclasses inherit the properties of their superclasses and can extend these attributes with additional ones. Because OWL-S service descriptions are nothing more than OWL documents, we can use all of OWL's domain modeling features 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, uses our sensor ontology in the obvious ways—for example, sensors provide Sensor services, which have SensorServiceProfiles and acoustic sensors provide AcousticSensor services having 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, the system knows only 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 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 let users define complex class expressions to indicate their requirements on the fly.

In OWL-S service descriptions, information for executing 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. Increasing numbers of WSDL-described Web Services are available on the Web, both from independent developers and large companies, such as Amazon and Google. Annotating these Web Services with OWL-S gives us a good opportunity to access several semantically described, executable services.



Enter URL: http://www.webservicex.net/uszip.asmx?WSDL

Operations

GetInfoByCity
GetInfoByState
GetInfoByAreaCode
GetInfoByZIP

Service information

File Name: GetInfoByZIP
Service Name: GetInfoByZIP
Text description: Get State Code,City,Area Code,Time Zone,Zip Code by Zip Code

Inputs

WSDL Parameter	WSDL Type	OWL-S Name	OWL Type	XSLT
USZip	xsd:string	USZip	zip-ont:ZipCode	

Outputs

WSDL Parameter	WSDL Type	OWL-S Name	OWL Type	XSLT
GetInfoByZIPRe...	ns8:GetInfoByZI...	GetInfoByZIPRe...	owl:Thing	

Namespaces

Abbr	URI
zip-ont	http://www.daml.org/2001/10/html/zipcode-ont
rdfs	http://www.w3.org/2000/01/rdf-schema
daml	http://www.daml.org/2001/03/daml+oil
rdf	http://www.w3.org/1999/02/22-rdf-syntax-ns

Figure 2. A tool to automate translation from WSDL descriptions to OWL-S.

Some aspects of deriving OWL-S descriptions from WSDL descriptions can be partially automated (see Figure 2). For each operation a WSDL document describes, the document will describe the input and output messages and their substructure for that operation. Normally, a WSDL operation corresponds to an OWL-S AtomicProcess, with that process's parameters corresponding to various message parts. In nearly all WSDL documents, the content of message parts are described by XML Schema data types, quite often complex types—types that describe elements with possible attribute or subelement structure.

Because parameter type compatibility is critical to our composition method, the service description must supply sufficiently expressive types. Two issues arise when trying to incorporate XML Schema data types in OWL-S service descriptions:

- OWL itself permits only a constrained range of XML Schema data types.
- For those it does permit, it provides constructs and reasoning services that aren't nearly as interesting as those it provides for OWL classes.

In particular, OWL currently only provides for defining properties whose range is one of a subset of the simple XML Schema data types (such as integers or strings), but no provision exists for using complex types. This situation isn't due to any logical difficulty with integrating complex types with OWL, but rather because OWL references data types by URI. Because there is no canonical way in XML Schema to determine a URI for a complex data type, OWL documents are constrained to use only data types with predefined URIs. We expect that the XML Schema working group will eventually resolve this situation. Until then, we can't even expect OWL reasoners with excellent data type support to process complex data types interoperably.

Even when this problem resolves itself, it would be preferable for many purposes to have the parameter types of OWL-S services be OWL classes, as that would allow for more flexible matching and more natural OWL-based descriptions. Because we're 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 service parameters' "wire format"—that is, the serialization of the values our process actually uses.

We extended the OWL-S grounding to include marshaling and unmarshaling functions that our OWL-S executor can use to coerce XML Schema values to OWL individuals and back. (These extensions, with further development by the OWL-S Coalition, were subsequently included in OWL-S. We did these extensions and their implementation in collaboration with Fujitsu Labs of America, College Park, Maryland, with extensive feedback from Ryusuke Masuoka.)

These functions are, by default, encoded as Extensible Stylesheet Language Transformation 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 becomes the actual input to the service. Marshaling functions are implemented as the inverse transformation. Using published XSLT obviates the need to extend the OWL-S executor with specific type-coercion functions—it just needs a generic XSLT processor, perhaps running as a remote service. Unfortunately, given the extremely free syntax of RDF/XML—especially, the plurality of equivalent forms—it's difficult to write XSLT that can handle all the legal serializations of a given RDF graph. The resulting stylesheet is almost impossible to understand and maintain.

Clearly, hand-writing such transformation functions is not feasible. Marshaling and unmarshaling functions can already be a source of subtle bugs because they require a deep understanding of both source and target formalism, coupled with a good understanding of the services both on the WSDL side (of the operational semantics of the service) and on the OWL-S side (of how the descriptions affect the various 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, because 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 don't generate that profile, our XSLT transformations can fail. Also, it's unclear that, even with a suitably designed profile, the necessary XPath queries will be sufficiently obvious and transparent to the programmer. Finally, although feeding the XSLT processor some XML allows for great flexibility—in choice of both implementation of processor and the specific instance of some processor—it's unlikely that the internal representation of the individual will be, say, W3C document object model (DOM) trees. So,

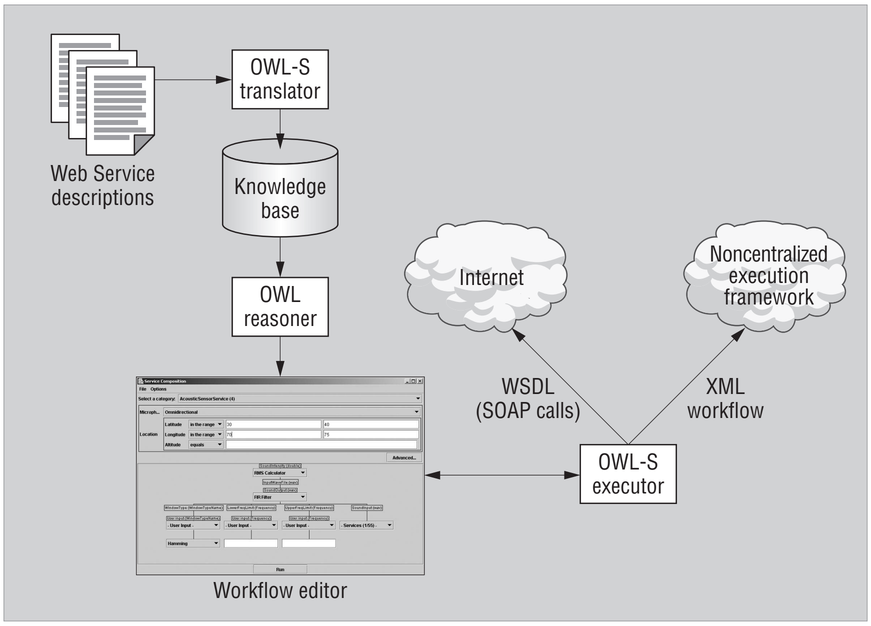


Figure 3. An overview of the prototype system.

the constant need exists for additional data conversion.

Incorporating an RDF- and OWL-sensitive query language (such as RDQL or Versa) into XSLT, or perhaps XQuery, standards would deal with all three issues. (The 4Suite XSLT parser already integrates the Versa RDF query language as an XSLT extension.) Even if generic XSLT or XQuery processors generally failed to include such extension, they would provide a standard and appealing target for OWL-S engines to implement. 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.

As an appealing alternative to either technique, we could use a higher-level mapping language, perhaps along the lines of Meaning Definition Language (MDL),[10] as Joachim Peer proposed.[11] If the mappings could be compiled to XSLT or other transformation languages, there would be an enormous gain in portability; by eschewing the general expressive power of programming languages such as XSLT, there might be a significant gain in transparency and analyzability. Unfortunately, designing such a language covering even a significant subset of the expressivity of OWL is a formidable task.

Implementation

Our system uses OWL-S service descriptions to support our interactive composition approach. It filters and selects services using matchmaking algorithms similar to those that the DAML-S Matchmaker implements.[12] The Matchmaker uses ServiceProfiles to describe both service requests and advertised services. 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, the system creates a ServiceProfile for the desired service. The service registry matches request profiles to advertised profiles using DAML+OIL subsumption as the core inference service. In particular, the DAML-S Matchmaker computes subsumption relations between individual IOPEs of the request and advertisement ServiceProfile. If the corresponding parameter classes are equivalent, there's an exact and thus best match. If there's no subsumption relation, then there's 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 IOPEs combine to produce an overall rating of the match.

Several researchers extended this algorithm to consider the subsumption relation between the request and advertisement profiles considered as whole concepts.[13,14] 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's called an Exact match.
- PlugIn. If request R is a subconcept of advertisement A, it's called a PlugIn match.
- Subsume. If request R is a superconcept of advertisement A, it's called a Subsume match.
- Fail. Otherwise, there's no match.

Our system uses the same basic typology of subsumption-based matches. In some contexts, we match on the basis of the subsumption of the entire profiles; in others, we use subsumption only to directly match individual parameters.

Figure 3 shows the general system architecture. The system has two separate components: an inference engine that stores service advertisements and processes match requests and the composer that generates the service composition workflow. The inference engine is an OWL-DL reasoner named Pellet (www.mindswap.org/2003/pellet). The composer communicates with the inference engine to discover possible matches and present them to users. It also lets users invoke the completed composition on specific inputs.

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

Matching on IOPEs

At each composition step, a list shows the IOPE-compatible services users can add to the composition. (Currently, we only match on IO because the specification of preconditions and effects is still an open OWL-S issue.) When the user selects a service from the list, the composer presents as options those services whose output could feed to the current service as an input. Suppose the selected service accepts an input of type Address, which is defined in a certain ontology with the concept hierarchy Figure 4 shows. We'd like to find the services that have an output compatible with this type. An output of a service would be compatible if it were

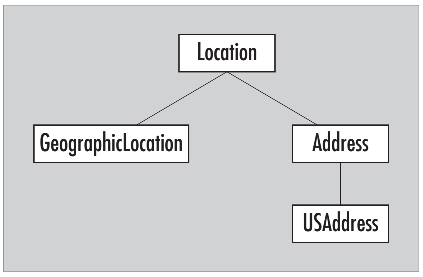
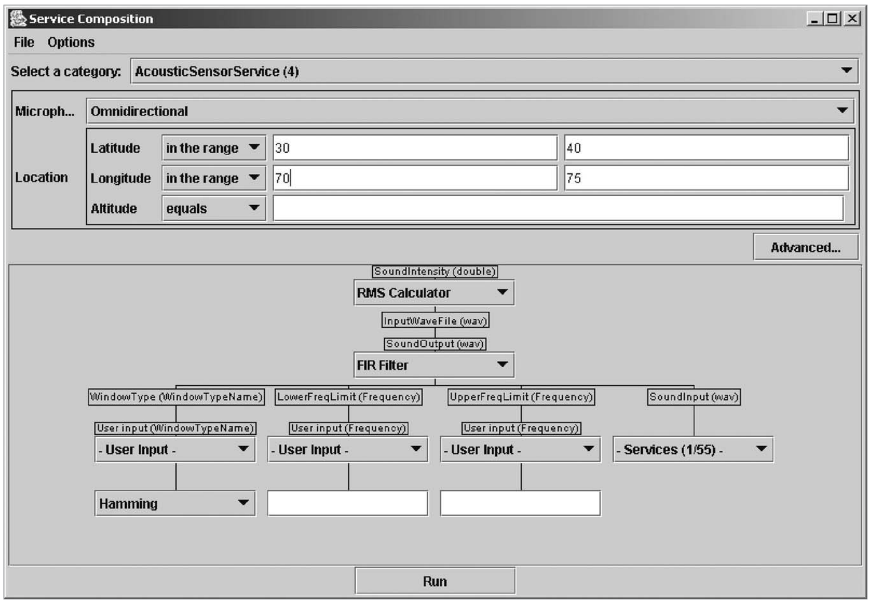


Figure 4. A simple hierarchy of location-related concepts.



Select a category: AcousticSensorService (4)

Microph... Omnidirectional

Location
Latitude in the range 30 40
Longitude in the range 70 75
Altitude equals

SoundIntensity (double)
RMS Calculator
InputWaveFile (wav)
SoundOutput (wav)
FIR Filter
WindowType (WindowTypeName) — User input (WindowTypeName) — - User Input - — Hamming
LowerFreqLimit (Frequency) — User input (Frequency) — - User Input -
UpperFreqLimit (Frequency) — User input (Frequency) — - User Input -
SoundInput (wav) — - Services (1/55) -

Figure 5. Filtering serves to present only omnidirectional acoustic sensors located at a specific location. The display shows that only one of 55 services satisfies these constraints.

of type Address or another concept that's subsumed by Address, such as USAddress.

When a service input subsumes the output, the output type is just a specialized version of the input type, so these services can still chain together. However, a service whose output is Location couldn't be composed with this service because the Address concept will most likely have additional properties and restrictions on the existing properties of Location.

Clearly, only Exact and Plug-In matches between the parameters of ServiceProfiles would yield useful results at this step. For service selection, we need a match on individual parameter types instead of whole profiles because 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 all cost service parameter values for the currently composed services, using the difference between that sum and the set limit to filter potential next steps.

Results are ordered in the list according to the degree of match. Exact matches are more likely to be preferred in the composition, so these services sit at the top of the list. The Plug-In matches come after the Exact matches, while Plug-In matches are ordered according to the distance between the two types in the ontology tree.

Filtering on service parameters

The number of services the list displays 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, making it infeasible for someone to scroll through a list and choose a service simply by name. Furthermore, even if the number of services is low, the service names themselves might not contain enough information to let a user make an informed choice. When the service's name 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, users can query a sensor description, linked to a particular service, as to the sensor's location, type, deployment date, sensitivity, and so forth.

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

Consider an 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 approach is better than presenting all the services, but not sufficiently better. The user can decrease the number of matches significantly, for example, by filtering the results to the services with AcousticSensorServiceProfiles. The composer then queries the inference engine about the possible service parameters of the selected service type. Based on the answer the engine returns, the composer creates a GUI panel in which the user can enter constraints for the service properties, as Figure 5 shows.

A service request profile combines the user's constraints. The service request goes to the inference engine, which applies the result of this new query to the previous result set. The engine removes from consideration services that don't satisfy the current constraints. The matchmaking for this step can use Relaxed matches as well as Exact and Plug-In matches. Using Relaxed matches will probably increase the choices presented, letting the user 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 the user generates using the existing prototype can itself be realized as an OWL-S CompositeProcess, so it can also 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 nonfunctional properties. Such a description is immediately available to the system as a named service, which can be filtered and composed normally. In this way, the user can quickly build up a set of complex compositions piecemeal as the tasks at hand demand.

Executing composed services

In its current implementation, the system executes the composition by invoking each

The Authors

Evren Sirin is a doctoral candidate at the University of Maryland. He is a member of Mindswap research group and works on Semantic Web Services and Web Service composition. He has also been developing various tools for the Semantic Web, including an API for OWL-S and a description-logic-based OWL reasoner called Pellet. Contact him at evren@cs.umd.edu.

Bijan Parsia is a research philosopher at the University of Maryland's MIND Lab. His current research focuses on Web-based ontologies, description-logic use and implementation, trust networks, and Semantic Web Services. Contact him at bparsia@isr.umd.edu.

James Hendler is a professor at the University of Maryland and the director of Mindswap research group. He has joint appointments in the Department of Computer Science and the Institute for Advanced Computer Studies. He was the recipient of a 1995 Fulbright Foundation Fellowship, is a member of the US Air Force Science Advisory Board, and is a fellow of the American Association for Artificial Intelligence. He is also the former chief scientist of the Information Systems Office at DARPA and is a member of the World Wide Web Consortium's Semantic Web Coordination Group. Contact him at hendler@cs.umd.edu.

individual service and passing the data between the services according to the flow the user constructed. 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 remote procedure calls to invoke individual services. Future iterations of both specifications will likely include more complicated arrangements of parties.

Improving IOPE matching with ontology translation services

With both IOPE matching and service parameter filtering, there's 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. Elaborating the service parameter filter forms by extending the definitions of the concepts used to describe those parameters is a straightforward process. We expect that such extension will occur using standard ontology editing tools.

We've already discussed improving IOPE matching by converting the IO type descriptions from XML Schema data types to OWL classes. In that process, the choice of target OWL class is critical to generating matchmaking hits. For any given domain, the Semantic Web is likely to have several somewhat overlapping ontologies—that is, ontologies with fairly similar, but distinct, concepts. If service description authors choose different, but relevantly equivalent, classes to unmarshal their XML Schema data types 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 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, then produce the relevantly equivalent set of assertions against another.

However, in one important sense these services are unimportant to composition. They're only important insofar as they promote the composition of other services that actually move the user closer to his or her goal. They're not suggestive of interesting further steps, thus are merely a burden on the user. To eliminate this obstacle, we don't actually present the translation services to the user, but rather have created *fused* services on the fly. A fused service is a chain of translation services terminating in a nontranslation service. The system presents the fused service as a type-compatible nontranslation service, thus increasing the number of substantial options at any particular step.

The service composition in the current framework requires a human in the loop. A human who has the domain knowledge for the task must guide the overall composition process while the composer determines the relevant choices at each step. Incorporating planning technology in the inference engine would further automate the system. The ability to access the machine-understandable data via the Semantic Web should make it easier to integrate a planner into the system.

One way to improve the suggestions provided for a match is to give the system the capability to learn from past user activity. The previous service compositions that operators created 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 nonfunctional properties will help the engine find better 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. ■

Acknowledgments

We thank Ryusuke Masuoka for his contribution to this work. This work is supported in part by the Army Research Laboratory, DARPA, Fujitsu Laboratory of America College Park, Lockheed Martin Advanced Technology Laboratories, the National Science Foundation, the National Institute of Standards and Technology, and the NTT Corporation.

References

1. E. Christensen et al., “Web Services Description Language (WSDL) 1.1,” World Wide Web Consortium (W3C) note, 2001; www.w3.org/TR/2001/NOTE-wsdl-20010315.
2. “SOAP 1.2,” World Wide Web Consortium (W3C) recommendation, N. Mitra, ed., W3C, June 2003; www.w3.org/TR/soap12-part0.
3. F. Curbera et al., “Business Process Execution Language for Web Services, v1.0,” S. Thatte, ed., IBM, July 2001; www-106.ibm.com/developerworks/webservices/library/ws-bpel.
4. B. Benatallah et al., “Towards Patterns of Web Services Composition,” *Patterns and Skeletons for Parallel and Distributed Computing*, S. Gorlatch and F. Rabhi, eds., Springer-Verlag, 2002, p. 265.
5. R. Masuoka, B. Parsia, and Y. Labrou, “Task Computing—The Semantic Web Meets Pervasive Computing,” *The Semantic Web–ISWC 2003: 2nd Int’l Semantic Web Conf.*, LNCS 2870, Springer-Verlag, 2003, pp. 866–881.
6. T. Berners-Lee, J. Hendler, and O. Lassila, “The Semantic Web,” *Scientific American,* vol. 284, no. 5, 2001, pp. 34–43.
7. M. Dean et al., “Web Ontology Language (OWL) Reference Version 1.0,” World Wide Web Consortium (W3C) note, Nov. 2002; www.w3.org/TR/2002/WD-owl-ref-20021112.
8. I. Horrocks et al., “DAML+OIL,” 2001; www.daml.org/2001/03/daml+oil-index.html.
9. D. Brickley and R. Guha, “Resource Description Framework (RDF) Model and Syntax Specification,” World Wide Web Consortium (WC3) recommendation, Feb. 1999; www.w3.org/TR/1999/REC-rdf-syntax-19990222.
10. R. Worden. “Meaning Definition Language, v2.06,” July 2002; www.charteris.com/XML-Toolkit/MDL.asp.
11. J. Peer, “Bringing Together Semantic Web and Web Services,” *The Semantic Web–ISWC 2003: 1st Int’l Semantic Web Conf.*, LNCS 2342, Springer-Verlag, 2002, p. 279.
12. M. Paolucci et al., “Semantic Matching of Web Services Capabilities,” *The Semantic Web–ISWC 2003: 1st Int’l Semantic Web Conf.*, LNCS 2342, Springer-Verlag, 2002, p. 333.
13. J. Gonzalez-Castillo, D. Trastour, and C. Bartolini, “Description Logics for Matchmaking of Services,” *Proc. Workshop Applications of Description Logics* (ADL 2001), CEUR, 2002.
14. L. Li and I. Horrocks, “A Software Framework for Matchmaking Based on Semantic Web Technology,” *Proc. 12th Int’l World Wide Web Conf.,* World Wide Web Consortium, 2003, p. 48.