Abstract. The addition of higher-level semantics to web services offers the means for systems based on these architectures to achieve greater flexibility and autonomy by allowing automatic service substitution and discovery. Constructing semantic descriptions of services, however, is problematic, as current proposals for semantic service languages are complex, requiring considerable user input in order to describe and bind semantics to underlying service implementations. Additionally, these descriptions are typically built separately from the underlying implementations of the services themselves, presenting problems of maintenance when underlying service descriptions must be changed. In this paper, we describe OWLSBuilder, a tool for authoring service semantics in OWL-S which exploits and complements the structure present in the source code of underlying Java web services. This approach allows us to automatically generate bindings between semantic entities and the underlying service descriptions as well as simplifying the process of maintaining the semantic descriptions themselves.

Keywords: semantic web services, tools, annotation, ontologies, OWL-S

1. Introduction

Semantic Web Services (SWS) offer a means to achieve meaningful automated service composition and substitution in heterogeneous service-oriented environments. Several languages (e.g., OWL-S [MPM+04], SAWSDL [W3C07a], WSMO [RKL+05], WSMO-Lite [RKL04] and SA-REST [SGL07]) can be used to describe (i) non-functional service properties (e.g., textual descriptions, service categorisations etc.) (ii) service semantics: Inputs, outputs, preconditions and results (IOPRs from here on) and (iii) bindings to the underlying service implementations. These languages are, by their nature complicated to write, requiring a working knowledge of an underlying ontology language, an understanding of the logical structure of the semantic description and (in many cases) a great deal of user-input in order to build the service grounding. This complexity may, in part be responsible for the current lack of functioning semantic web service descriptions (a study in [KZ08] completed in 2008 found only 65 publicly available semantic service descriptions in the wild).

A common approach to building and managing service descriptions is to treat the “description phase” of service creation as separate from the process of creating the underlying services themselves (e.g., Protégé-OWL [KFN04], OWLS Editor [SM04], WSMO Studio [DSM07]). While this approach is well suited to adding new semantic descriptions to existing services written by third parties, this separation poses problems when (i) creating semantic descriptions for new services, as service designers must switch from the metaphor they use to develop the services themselves (typically a programming language - see below) to a separate tool to produce descriptions, and (ii) maintaining descriptions, as changes must be manually tracked and corrected in the corresponding semantic description. While it may be possible to assist users by verifying descriptions, some parts of current languages (primarily XSLT groundings in OWL-S and SAWSDL) are difficult to accurately verify for correctness.
The prevalent approach (shown in Figure 1 for services derived from JAX-WS) for building web services in high-level languages such as Java, C# Visual Basic.net etc. is somewhat different. Rather than producing a service implementation and attaching a web-service description later, the description (in WSDL and XML Schema) is typically built directly from and linked closely to the program structure of the underlying implementation (or interface description): web service operation descriptions (WSDL) and underlying data formats (XML/XSD) are both derived semi-automatically from the method signatures and from the classes of data passed in and out of those methods. In the case where the desired web service or XML bindings differ from default values, in-language annotations (supported by Java 1.5, C# 2.0) on service methods and data classes are added to clarify and guide the construction process. Consequently, creating and maintaining web service descriptions is relatively simple as the transformation is driven directly by the program structure.

2. The OWL-S model for semantic services

While a number of proposed languages exist for adding semantics to services we focus on the OWL-S model in this paper. For the context of the paper we summarise OWL-S version 1.2 here, full documentation can be found in [OWL]. We target the OWL-S model as, out of the available semantic service models it has the most mature specification and has a range of supporting tools in a mature state.

OWL-S is an upper ontology for service descriptions written in the OWL web-ontology language [W3C07b]. An OWL-S service description consists of a number of facets as we now describe:

**Service profile** The service profile provides the information required to discover a web service, it is split in to non-functional aspects such as those which relate to the creator of the service and a description of its purpose, and functional aspects such as the required inputs, outputs, preconditions and results of invoking the service.

**Service process model** The process model describes an operational model of a service. This may be either a single atomic process, or an orchestration of two or more service operations composed by workflow connectives which must be executed in their entirety to achieve the outcomes of the service. Each of the atomic, component service processes within this workflow (or the single process in the case of a service which is itself atomic) also include their inputs, outputs, preconditions and results.

**Service grounding** The service grounding links the semantics of the service to its operational implementation. For a typical web service using WSDL, the grounding consists of a set of transformations (written in XSLT) which, given suitable service inputs expressed as a serialised RDF graph, produce suitable XML input which may be used to invoke the underlying SOAP service, and correspondingly translate the XML output of the SOAP service back into RDF triples. Groundings are extensible to different underlying service implementations, for instance a grounding exists for directly binding to Java method invocations, allowing local services to be called without translating inputs and outputs via XML.

OWL-S distinguishes between atomic and composite services. An atomic service contains a single atomic process description which is exposed through its process model and grounded to a given implementation through the grounding model. In a composite service, the semantics exposed in the service profile represent the overall semantics of executing the underlying, compound service workflow. As we are primarily concerned with the process of annotating a single atomic java service method we focus on atomic service semantics in this paper.

The fundamental functional properties of services are expressed in the profile and process model in terms of their Inputs, Outputs, Preconditions and Results (IOPRs), we describe these in more detail here.

**Input and output parameters** are parameters which are passed into, and returned by the service when it is invoked. These are named and each is associated with a corresponding OWL class denoting the type of the parameter. When invoking a semantic service, an execution environment must bind an RDF-described object to each of the corresponding input parameters of
the service. These are in turn grounded to the relevant underlying data structure for the for the service (XML fragments in the case of SOAP services) and passed to the concrete service implementation. After the execution of the underlying service is complete, the concrete outputs produced by the service are translated into RDF objects and bound to their respective output parameters where they may be interpreted as semantically annotated objects.

**Service preconditions** are conditions which must be met in order for the service to be executed successfully. Preconditions denote constraints over the inputs to the service. While OWL-S is designed to be agnostic to the underlying language used for these constraints the most prominent language is the Semantic Web Rule Language (SWRL), a rule language which uses Datalog-like syntax and DL reasoning. In this case preconditions are expressed as SWRL expressions (conjunctions of atoms over the data and object properties of the input objects and built in SWRL functions) which must be satisfied when the service is invoked.

**Service results** denote the logical outcomes of invoking a service. Results in OWL-S are conditional, and different result assertions may come into force depending on the inputs passed to the service and the outputs produced. Results are used to assert auxiliary information (as triples referring to existing or new service outputs) including the effects of executing the service. As with preconditions result conditions and effects are expressed using the SWRL language.

### 3. Modelling semantic services with annotations

The following section describes the input required from a service designer and the operation of the OWLSBuilder tool which facilitates the construction of semantic service descriptions. We assume that the designer is already familiar with the JAX-WS/JAXB model for constructing web services from Java and that they already have, or are constructing such a service. Figure 2 outlines the operations of the tool used for deriving semantic descriptions from JAX-WS annotated Java source (classes or interfaces). The process summarised as follows and described further below: (i) A domain ontology is selected, constructed manually or constructed automatically from annotated Java domain classes. (ii) Existing JAX-WS service classes are then extended with Java annotations describing OWL-S service properties (IOPRs) and non-functional service properties of the desired service. (iii) The annotated Java classes are compiled and the retained JAX-WS and semantic annotations are parsed to build a conventional JAX-WS model, and a semantic model of the service bound to that JAX-WS model. (v) Service profile and process descriptions (including IOPRs) are built from the semantic model. (iv,vi) Grounding descriptions are built for invoking the semantic service through java interface, and via SOAP. In the latter case XSLT transformations which lower RDF service input parameters to XML and lift XML output parameters to RDF are generated automatically based on the bindings specified in the semantic model.

A summary of the annotations supported by the tool are summarised in Table 1, annotations are broadly broken down into general annotations which may be applied to both data and service classes, data binding annotations and service binding annotations.

#### 3.1. Domain binding

Domain binding serves to connect the Java and JAX-WS model of the underlying web service with the semantic (RDF) model of the OWL-S service description. OWL Classes referred to within the IOPRs of OWL-S services are expressed in terms of one or more OWL domain ontologies. In order to ground the service to a concrete web service we must be able to infer the XML structure of a given component of a service call from an RDF graph description of the component and vice-versa. In the case of JAX-WS services, this XML structure is derived semi-automatically from the structure of the underlying Java beans which are referred to by the service using supporting annotations in JAX-B (Java XML Binding).

In the case where this domain ontology already exists, binding annotations are added to Java entity classes (ii) by the designer which link the types and Java properties of those entities to corresponding OWL classes and OWL properties.

In the case where no domain ontology exists, entity classes may be annotated first and used to generate a new ontology based on the annotations. This process annotations on bean classes and creates new OWL classes and properties which correspond to those of the underlying beans. It should be noted that due to the restricted expressiveness of the Java type system, ontologies generated in this fashion may only correspond to OWL-Lite semantics without data restrictions on classes. Ontologies generated in this fashion may be manually extended by the user as required.
The domain binding model used by the tool assumes a default correspondence between Java classes and OWL classes which may be extended and customised using binding annotations, and Java primitive types and XSD types (used in both XSD/WSDL and OWL/OWL-S).

Figure 3 shows how two annotated beans ANumber and Participant have been annotated to bind them to two classes in their respective ontologies. The ANumber class shows a simple Java bean class bound to the http://numbers.org/Numbers.owl#Integer class with the Java bean property value corresponding to the hasValue property in that ontology. The Participant class shows a more complex binding; in this case the properties of the OWL class in question do not directly correspond to the structure of the Java Bean. To overcome this, binding annotations may be specified at the class level (@OwlsBindings) which link sub-properties of a give Java object to local properties of it’s corresponding OWL class. In the example, the name property of the details object contained in the Participant class is bound to the hasName property of the top-level OWL class and the objID property of the details property is bound to the identifying URI (referred to by {id}) of the OWL class.

The framework also allows collection properties of Java beans to be bound to OWL properties. Unordered collections (java.util.Set) are bound as multi-valued properties on the relevant OWL property. This is the case the the preferences property of the Participant object above. Ordered collections (arrays and subclasses of java.util.List) are treated somewhat differently, in this case, the type of the target property must be defined as a subclass of the DAML ObjectList type 1. Where contained types are specified in arrays or as type parameters in generic collections, these are used to bind/validate the contained types of the respective OWL properties.

The tool and its internal representation of these bindings is designed to be flexible in the sense that, in addition the binding annotations shown in Figure 3, bindings may also be imported from other Java to RDF binding systems. At present bindings from Jastor [KPBP04] are supported, however, in principle any framework which provides a binding scheme which allows for serialisation of Java beans to RDF such as RDFReactor2, Kazuki3 and JenaBean4 could also be supported with additional plugins to the tool.

---

1. http://www.daml.org/services/owl-s/1.2/generic/ObjectList.owl
3.2. Service description

Once domain entities have been linked to the required domain ontology the service itself is annotated (ii in Figure 2). By default (when annotations are omitted) the tool uses a one-to-one correspondence between Java parameters and service inputs and Java return values and service outputs. Java methods may only return a single value however, in the case where this return value has a compound type, the designer may prefer to describe the semantic service as having multiple outputs derived from elements of the Java compound return type. Likewise a designer may wish to break compound input types down into multiple OWL-S input parameters.

Our tool supports these binding cases by assigning a context path to each input and output annotation. This path (which follows the dotted syntax used in binding targets described in Section 3.1) indicates the root in the Java bean graph of the relevant parameter from which the service parameter should be defined.

As with JAX-WS, default values are applied where annotations or annotation parameters are omitted, allowing services to be created quickly with minimal annotation. Most aspects of the OWL-S model are supported (except those relating to composite processes which are not required for atomic process descriptions).

3.3. Preconditions and results

Preconditions and results are logical expressions which, because of restrictions on Java Annotation syntax would be difficult to represent succinctly using only Java annotations. Instead we opt to use a simple logic-programming style syntax for encoding the SWRL conditions and expressions in preconditions and results as text. These expressions are themselves embedded within annotations and are parsed and verified separately from the main semantic annotations.

Precondition annotations are encoded using the @OwlsPreconditions annotation which takes one or more expressions, these expressions are expressed as conjunctions (separated by commas) of SWRL literals. Accepted literals include ontology property atoms (e.g. prefix:hasValue(#ind,20) where prefix:hasValue is a property in an ontology imported by the service), SWRL built-in functions which allow for ontological inference in the body of rules as well as mathematical, comparison, date, string and list functions. The full set of specified SWRL built-ins supported by the parser however it should be noted that some of these functions are not currently supported by reasoners (e.g. list functions).

Conditional results annotations are encoded similarly, with a SWRL conditional expression representing the (possibly empty) condition of the result applying and a SWRL expression denoting the effect (as assertions over the knowledge base) of the result’s application when the condition is true.

Expressions may refer to any inputs or outputs in their scope as variables (inputs for preconditions, both for results). OWL-S also supports process variables which are scoped across the preconditions and results of a service (sometimes called existential or local variables). These variables may be declared on a given process using the @OwlsLocal(name="v",type="Type") annotation. Similarly result variables may be declared within the context of a result (between the condition and effect of a result) using the @ResultVar annotation.

3.4. Intermediate semantic model and core service description

In order to construct a semantic description the tool builds an intermediate model of the service based on the annotations defined in the service source code. The tool itself operates on compiled service and domain classes (which include semantic annotations and JAX-WS annotations), these classes are loaded and examined using the Java reflection API. The first phase of the tool parses the semantic annotations contained in these classes and builds a programmatic model representing the service, non-functional properties, its implementing OWL-S process and its IOPRs. When handling annotations for service inputs and outputs the tool takes into account the underlying Java context (the structure of the underlying implementation method including its return value and its parameters). This context is then used in order to bind the semantic model of those parameters to their XML representation in the grounding phase (see below). Preconditions and results are parsed out of the annotations (described above) and then translated into an RDF model of the SWRL expressions they represent which is stored along with the service description. The second phase of the tool validates the intermediate model to ensure that relevant OWL types and properties exist and that variables used SWRL expressions follow the scoping rules mandated by the OWL-S specification, reporting errors to
the user where they are found. The resultant intermediate semantic model contains a hierarchical description of the service object, its profile and the atomic process description for the service. This model is translated into an RDF graph (via the OWL-S API) representation of the service which can be exported as a serialised service description.

3.5. Groundings

Finally, the tool produces groundings based on the mappings contained in the intermediate semantic model. Groundings in OWL-S are flexible and separate from the description of the core service semantics, allowing services to have different types of grounding depending on their underlying implementation (and the capabilities of the caller of the service). At present we target two grounding languages: the WSDL grounding model (used for SOAP web services) and the Java grounding model supported by the OWL-S API library. The latter grounding model allows services to be invoked directly as Java objects.

XSLT is a functional programming language which transforms one XML document into another via a series of internal templates which may be invoked either by matching a given part of the input document implicitly, or called explicitly within another template via a template call. For each service grounding a transformation is required for each input and output. Input groundings convert serialised RDF to XML elements which correspond to the SOAP message parts of the outgoing message of a call. An OWL-S execution framework such as OWL-S API then assembles these SOAP message parts into a soap envelope which is dispatched to the service as a request.

The tool produces input groundings for each of the semantic service inputs specified by annotations by inspecting the Java context of the given parameter annotation and any bindings applied in the service interface and then building a root parameter template for that parameter. The root template applies to the incoming RDF description as a whole and is the first template which invoked by the grounding for that parameter. This root template builds a top-level XML element which corresponds to the appropriate SOAP message part of the relevant service parameter and all sub-properties. Within this root template, when a sub-element which is compound is seen (e.g., a Java class with associated domain bindings) then a template call is inserted to the appropriate sub-element template for that type. These sub-element templates recursively build the XML representation of a given Java type from its corresponding RDF structure using the bindings described in Section 3.1. In the case where multiple options exist for serialising a given RDF type (e.g., a Class has sub-classes) then an conditional is used to dispatch the most specific template based on the current RDF context. As the grounding transformation is constructed, the transitive closure of all required sub-element templates is constructed by examining dependent properties of Java classes referred to by the service method parameter type in question.

The convention in JAXB is to serialise list types as sequences of child elements, however in RDF/OWL the convention is to use a recursive, singly linked lists. Where a list type is used, a special template is inserted which unrolls RDF-style lists into sequences of XML elements.

Output transformations work much the same way, but in reverse, in this case the transformation builds a serialised RDF document by parsing the XML of a relevant SOAP message part which corresponds to a given service output.

Grounding generation is extensible in OWLSBuilder as it is in OWL-S. To demonstrate this the tool includes support for generating groundings which allow Java components to be invoked directly as semantic services (the mechanism is included as part of the OWL-S API). In contrast to the WSDL grounding which requires an XSLT transformation, the Java grounding transforms RDF to and from Java objects on the fly using the included domain and service bindings. This approach negates the need to communicate over HTTP and may be useful for speed and/or for treating local components and remote services in a homogeneous manner.

4. Run-time support

Service descriptions can be generated off-line by running the OWLSBuilder tool on the compiled Java service and domain classes, in this case the generated .owl files must be stored and published manu-

5http://on.cs.unibas.ch/owls-api/index.html

6 As RDF is a graph-structured language, the XSLT transformations used for inputs require that the RDF given as input is serialised in a canonical way starting with the object representing the parameter value in question and expanding objects in properties at the point they are first seen. Where graphs are not serialised in this way then XML fragments will be omitted - this is a limitation of the OWL-S Specification rather than the tool.
Table 1. Summary of service annotation syntax.

<table>
<thead>
<tr>
<th>Annotation Syntax</th>
<th>Scope</th>
<th>Required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@URINamespace(prefix = &quot;ns1&quot;, value = &quot;<a href="http://a.uri.net">http://a.uri.net</a>&quot;)</td>
<td>class</td>
<td>no</td>
<td>Declares a namespace prefix which may be used in abbreviations elsewhere in annotations.</td>
</tr>
<tr>
<td>@OwlsImport(&quot;<a href="http://a.uri.net">http://a.uri.net</a>&quot;)</td>
<td>class</td>
<td>no</td>
<td>Imports a given ontology into the service ontology</td>
</tr>
<tr>
<td>@OwlsBinding(from = &quot;java.path&quot;, to = &quot;some.rdf.path&quot;)</td>
<td>class method</td>
<td>no</td>
<td>Externally binds the Java element (contained in this.getJava().getAttr()) to a corresponding RDF node.</td>
</tr>
<tr>
<td>@OwlBean(name = &quot;Integer&quot;, ontology = &quot;<a href="http://numbers.org/Numbers.owl">http://numbers.org/Numbers.owl</a>&quot;)</td>
<td>class</td>
<td>no</td>
<td>Indicates the OWL class which corresponds to this Java domain class</td>
</tr>
<tr>
<td>@OwlProp(name = &quot;hasValue&quot;, ontology = &quot;<a href="http://numbers.org/Numbers.owl">http://numbers.org/Numbers.owl</a>&quot;)</td>
<td>method</td>
<td>no</td>
<td>Indicates the OWL property which corresponds to this Java bean property. This must be applied to a Bean getter method (e.g. methods named getPropName()) with a corresponding getter method</td>
</tr>
<tr>
<td>@OwlsClass(defaultOntology = &quot;<a href="http://numbers.org/Numbers.owl">http://numbers.org/Numbers.owl</a>&quot;)</td>
<td>class yes</td>
<td></td>
<td>Declares that a service class contains OWL-S annotated methods.</td>
</tr>
<tr>
<td>@OwlsService(name = &quot;AddService&quot;, description = &quot;...&quot;)</td>
<td>method yes</td>
<td></td>
<td>Declares basic properties of the service derived from a given service method, including the service name.</td>
</tr>
<tr>
<td>@OwlsOutParam(name = &quot;rv&quot;, owlType = &quot;#Integer&quot;, bindings = [ @OwlsBinding(from = &quot;.&quot;, to = &quot;hasValue&quot;) ] )</td>
<td>method no</td>
<td></td>
<td>Declares an output service parameter, including its name, type and the relative location within the Java return type to base the OWL-S parameter on.</td>
</tr>
<tr>
<td>@OwlsInParam(name = &quot;x&quot;, owlType = &quot;#Integer&quot;, bindings = [ @OwlsBinding(from = &quot;.&quot;, to = &quot;hasValue&quot;) ] )</td>
<td>method, param. no</td>
<td></td>
<td>Declares an input service parameter in a similar way to outputs. Input parameters may be declared relative to a given Java parameter, and multiple service inputs may be derived from a given Java parameter.</td>
</tr>
<tr>
<td>@OwlsLocal(name = &quot;var&quot;, type = &quot;spre;Type&quot;)</td>
<td>method no</td>
<td></td>
<td>Declares a process local variable of type spre;Type which may be referred to and bound in a precondition rule, and referred to in a result condition or effect.</td>
</tr>
<tr>
<td>@OwlsPreconditions(num:hasValue(#x, #xval), &quot;swrlb:greaterThan(#xval, 0)&quot;)</td>
<td>method no</td>
<td></td>
<td>Declares a SWRL syntax precondition expression on a service method. The expression on the left restricts the value of the hasValue property of parameter #x to values greater than 0. The syntax of preconditions is discussed further in Section 3.3</td>
</tr>
<tr>
<td>@OwlsResult(condition = &lt;swrl&gt;, effect = &lt;swrl&gt;, vars = @OwlsResultVar( . . . ))</td>
<td>method no</td>
<td></td>
<td>Declares a conditional result of the current service, it’s effect and any locally scoped result variables.</td>
</tr>
</tbody>
</table>

ally by the designer. In many cases, however it may be preferable to publish the OWL-S descriptions simultaneously with the service itself corresponding to the mechanism by which JAX-WS services are typically published. This approach ensures that the published semantic description of a service corresponds exactly to that of the underlying implementation at all times. In order to facilitate this, the OWLSBuilder tool includes a plugin to the open source Apache CXF 7 web service framework. When deployed in an CXF service environment this plugin automatically introspects the implementing class of each JAX-WS service as it is deployed and, in the case that OWLSBuilder annotations are present, publishes the OWL-S description corresponding to the deployed service in the service framework. The CXF plugin may also be extended to interact with other service repositories, allowing descriptions to be published externally.

5. Example

In the following section we give an example of how an Java service interface is annotated. The service in question is a simple addition service which takes two Java objects of type ANumber (shown in Figure 3 and described in Section 3.1) adds the values and returns another object indicating the sum of the first two. For the sake of demonstrating the capabilities of the tool we assume that the service requires that both numbers

7http://cxf.apache.org/
must be positive and that we wish to know when the result is going to be odd or even.

Figure 4 shows the annotated service source code for the example service. The semantic service parameters are described using a numerical ontology (http://numbers.org/Numbers.owl) which contains two classes Integer with an integer data property hasValue and a subclass EvenNumber. The annotations in lines 2-3 bind a prefix #num to the domain ontology, and import the domain ontology into the service. The URI prefix declaration simplifies references in subsequent annotations and in SWRL expressions. The @OwlsClass annotation in line 4 declares the default URI used for the service ontology (where subsequent OWL-S service objects will be declared) and binds the URI of the numbers ontology to the default namespace, this namespace is used for types referred to in parameters. The @OwlsService annotation on line 6 declares that the add method should have an OWL-S service description and its basic properties. The service output (#rv) and inputs (#x and #y) are declared with the on lines 7, 18 and 20 respectively. The OWL type (#Integer) and bindings for #rv and #y are determined automatically by examining the domain annotations on the associated Java type. In order to demonstrate how a binding may be overridden in a service, for parameter #x the type and bindings are included explicitly in the @OwlsInParam annotation. Lines 8–14 declare the logical semantics of the service. For this service we have two local variables (#valX, #valY), these are bound to the value of the hasValue property of the two input parameters in a precondition which ensures that both of these values are greater than or equal to 0 using SWRL built-ins. Finally a the result of the service conditionally asserts that the output parameter #rv is a subclass of EvenNumber in the domain ontology if the sum (variable #sum) of #valX and #valY is divisible by 2.

6. Related Work

Existing approaches to constructing semantic web service descriptions can be broken down as follows: Ontology based editors: such as the Protégé OWL-S tab and the OWL-S editor provide pro-forma user interfaces for building descriptions based on the structure of the underlying ontology from which services are defined. While these simplify the process of constructing the core semantic descriptions of services, neither of these tools provide a means for a user to link semantic descriptions to their underlying implementations or verify that such descriptions are correct. A number of tools use the underlying WSDL descriptions to assist in constructing both the semantic description and groundings as is the case in [GBT05] and [DSMK07] which provide interactive drag-and-drop systems for associating ontological concepts with elements of existing web service descriptions in OWL-S and WSMO respectively. The SAWS: [SCGC08] and ASSAM [HK04] tools extend this model further by using machine learning to assist the user by predicting likely association between semantic concepts and the underlying service. Drag and drop interfaces are clearly beneficial in assisting the process of building
groundings, however the maintenance of the groundings they create is still problematic as the model for grounding a service is still maintained externally to the service description and a designer must manually update and correct grounding descriptions following a service change.

Our tool is not the first to propose deriving semantics from code-level annotations. Of particular note is METEOR-S [POSV04, RMVS04] which supports the embedding of semantic description properties within comments on service code. METEOR-S also supports the generation of OWL-S service descriptions, however it lacks support for the automatic generation of XSLT bindings from underlying bindings and instead relies on simple bindings for primitive types and user-generated, reusable XSLT transformations for compound types, while this provides assistance to a designer in some cases (in the sense that they only need to construct transformations for a given type once) the production of those transformations itself is still an inherently manual and potentially complicated process.

Finally, [ZDK09] presents an embedded annotation framework aimed at semantically describing the functionality of Java software components and while it also uses Java 1.5 annotations for describing the inputs, outputs and non-functional properties of Java classes, it lacks support for expressing logical properties of components (Preconditions and results) and generating the groundings required for externalised semantic languages like OWL-S.

7. Evaluation and future work

The tool has been evaluated in the context of the EU-funded ALIVE project where it has been used to build base-level (atomic) OWL-S services for two use cases in intelligent telecommunications routing and tourism advice respectively. In the trial, domain ontologies for the service domain were produced in OWL by academic partners which were then handed off to two Java developers responsible for creating the underlying web services used by the project. Neither of the developers had any prior experience in semantic web services. In both cases, with minimal assistance the developers were able to create functioning service descriptions which have since been integrated into the project as demonstration services. In total, 26 atomic service descriptions were produced over approximately 3 weeks (with the rate at which services were produced increasing as the developers’ experience increased). Feedback from the developers was generally positive however one developer did report some frustration with having to correct incorrectly typed names of OWL classes and properties used in annotations and SWRL conditions. These problems could be resolved by adding support for parsing and validating annotation semantics to a Java programming environment such as Eclipse.

The current version of the tool focuses on WSDL-described SOAP services with semantic annotations in OWL-S. A number of other semantic service proposals exist such as WSMO, WSMO-Lite, SA-REST and SAWSDL. It is not clear at time of writing which (if any) of these standards will gain widespread adoption. Of interest is whether the current OWLSBuilder architecture can be applied to any of these other semantic description frameworks. Of particular interest is SAWSDL which aims to provide a lightweight semantic annotation and binding framework as an extension to the WSDL 1.1 and 2.0 web service description model. SAWSDL is agnostic to the underlying service description model and does not prescribe a particular model for describing the semantics of services however the approach taken to linking semantic data to the underlying web service data (the groundings) is similar to OWL-S. In SAWSDL, the XML schema of a service interface is tagged with semantic model references, and lifting and lowering mappings which bind these to a semantic model are described using an external language such as XSLT or SPARQL. The semantic model references used in SAWSDL correspond to the bindings described in Section 3.1 of our tool and could be extracted in order to extend the WSDL model produced by JAX-WS with SAWSDL annotations. Likewise the liftings and lowerings in SAWSDL (for XSLT, bound to RDF/OWL at least) follow exactly the same pattern as OWL-S groundings, such that the XSLT groundings produced by our tool can be used verbatim in SAWSDL descriptions.

SOAP is not the only protocol available to web service designers and for a class of services the simpler RESTful service model (relying on conventional HTTP verbs and URI conventions instead of

---

8This approach was the de facto style for code-annotation before the widespread adoption of embedded annotations and achieves similar results.

9The descriptions of these services are published automatically here: http://147.83.200.118:8080/repo/web/list, however as the project is ongoing some services may not always be available.
a new protocol layer) is preferred by many developers and users. Several binding mechanisms exist for defining REST services in Java, including JAX-RS, a standards-based annotation framework which is similar to JAX-WS. SA-REST[SGL07] proposes a microformat-based semantic description language for RESTful services which conceptually follows a similar model to SAWSDL (with model references and lifting/loweringings). Descriptions are, however not embedded in service descriptions but given as markup within textual documents. While it is not clear how or if such documents should be produced from Java services, as with SAWSDL transformations could automatically be produced using existing parts of the OWLSBuilder tools.

The OWLSBuilder tool is open source and freely available\(^{10}\). OWLSBuilder currently provides annotation support for JAX-WS services and generates OWL-S 1.2 services.

**References**


\(^{10}\)http://wiki.bath.ac.uk/display/owlsbuilder


