Automatic Creation of Web Services from Extraction Ontologies

Cui Tao, Yihong Ding, and Deryle Lonsdale

Brigham Young University, Provo, Utah 84602, U.S.A.
{ctao, ding}@cs.byu.edu, {lonz}@byu.edu

Abstract. The Semantic Web promises to provide timely, targeted access to user-specified information online. Though standardized services exist for performing this work, specifying these services is too complex for most people. Annotating these services is also problematic. A similar situation exists for traditional information extraction, where ontologies are increasingly used to specify information used by various extraction methods. The approach we introduce in this paper involves converting such ontologies into executable Java code. These APIs act individually or compositionally as services for Semantic Web extraction.

1 Introduction

One goal of the Semantic Web is to enable personalized, automatic, targeted access to information that can be useful to a user. This might include finding out the availability and price of a book, or reserving and ticketing travel itineraries. Currently tools to provide such services are in their infancy, but much human intervention is needed to hard-code and hand-specify their functionality.

In order to increase the use of services with the Semantic Web, we need a more automatic way of making them machine-interpretable, and we need to annotate them with a more systematic description of their respective semantic domains, conceptual coverage, and executable functions. This goal is perhaps optimally realized when a Web service is annotated by an ontology. An ontology enables domain experts to declare standardized, sharable machine-processable knowledge. In a traditional Web setting we have found ontologies useful for information extraction applications. In this paper we show how ontologies can serve two useful purposes in the creation of Semantic Web services.

First, we map an extraction ontology to a set of Java APIs, through which we can automatically create atomic information extraction Web services. Each of these atomic Web services instantiates a lexical ontology concept; domains with extensive vocabularies could spawn a considerable number of derived services. Since each atomic service directly derives from formal definitions in an ontology, we can use this information to sidestep the traditional requirement for hand-annotation of the service, a difficult process. Second, given these Java-instantiated services and their corresponding ontological properties, we can automatically compose complex Web services to respond to users' requests.

Our goal is thus to develop a method of automatically creating Web services based on extraction ontologies. In this paper we begin by introducing extraction ontologies. In the next section we sketch the process of compiling these ontologies into executable Java classes. Next we discuss web services and their creation via composition of these classes. Finally, we mention future work and applications.

2 Extraction Ontologies

An extraction ontology is a conceptual-model instance that serves as a wrapper for a narrow domain of interest such as car advertisements. We use OSM [3] as the semantic data model for an extraction ontology; we also augment OSM to allow regular expressions as descriptors for constants and context keywords. The conceptual-model instance includes objects, relationships, constraints over these objects and relationships, descriptions of strings for lexical objects, and keywords denoting the presence of objects and relationships among objects. When we apply an extraction ontology to a source document, the ontology identifies instantiated objects and relationships and associates them with named object sets and relationship sets in the ontology. This wrapping of recognized strings makes them machine-interpretable in terms of the schema inherent in the conceptual-model instance.

In essence, an extraction ontology is semantically equivalent to a Semantic Web ontology written in OWL[1]. We are not using OWL directly because our OSM extraction ontology contains formal specifications for data extraction patterns beyond standard OWL. To be compatible with the Semantic Web standards, we have developed an OWL-OSM converter that transforms the two ontological representations; however, this ontology conversion research is beyond the scope of this paper.

An extraction ontology consists of two components: (1) an object/relationship-model instance that describes sets of objects, sets of relationships among objects, and constraints over object and relationship sets, and (2) for each object set, a data frame that defines the relevant extraction patterns. Figure 1 shows part of our car-ads extraction ontology, including object/relationship model declarations (Lines 1-8) and sample data frames (Lines 9-18).

An object set in an extraction ontology represents a set of objects which may either be lexical or non-lexical. Data frames with declarations for constants that can potentially populate the object set represent lexical object sets, and data frames without constant declarations represent non-lexical object sets. *Car*, for example, is a non-lexical object set. *Year* (Line 9) and *Mileage* (Line 14) are lexical object sets whose character representations have a maximum length of 4 characters and 8 characters respectively. We describe the constant lexical objects and the keywords for an object set by regular expressions using Perl-like syntax. We denote a relationship set by a name that includes its object-set names (e.g. *Car has Year* in Line 2 and *PhoneNr is for Car* in Line 8). The *min:max*

[1] Web Ontology Language (OWL), http://www.w3.org/2004/OWL/

```
1. Car [-> object];
2. Car [0:1] has Year [1:*];
3. Car [0:1] has Make [1:*];
4. Car [0:1] has Model|Trim [1:*];
5. Car [0:1] has Mileage [1:*];
6. Car [0:*] has Feature [1:*];
7. Car [0:1] has Price [1:*];
8. PhoneNr [1:*] is for Car [0:1];
9. Year matches [4]
10.         constant {extract "\d{2}";
11.                   context "\b'[4-9]\d\b";
12.                   substitute "^" -> "19"; },
13.                   ...
14. Mileage matches [8]
15.         ...
16.         keyword "\bmiles\b", "\bmi\.", "\bmi\b",
17.                 "\bmileage\b", "\bodometer\b";
18. ...
```

Fig. 1. Car-Ads Extraction Ontology (Partial)

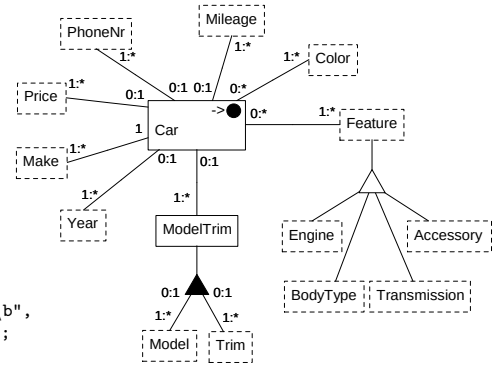

Fig. 2. The Graphical View of the Car Ontology

pairs, such has *1:**, *0:**, or *0:1*, in the relationship-set name are participation constraints. *Min* designates the minimum number of times an object in the object set can participate in the relationship set and *max* designates the maximum number of times an object can participate, with * designating an unspecified maximum number of times.

Figure 2 shows the equivalent graphical view of the ontology in Figure 1. A dashed box represents a lexical object set and a solid box represent a non-lexical object set. Lines between object sets represent the relationship sets among them and the digit pairs on both ends of the lines represent participation constraints. A black triangle represents an aggregation which constitutes a part/subpart relationship. In Figure 2, for the $Model|Trim$ aggregation, we implicitly have the binary relationship sets *Model* is part of $Model|Trim$ and *Trim* is part of $Model|Trim$. OSM uses a clear triangle to denote a generalization/specialization and connects a generalization at an apex of the triangle and to a specialization at the opposite base. In Figure 2, *Feature* is the generalization of *Engine*, *BodyType*, *Accessory*, and *Transmission*. In this research, we convert the knowledge represented in OSM to the Java programming language. The next section discusses this translation process.

3 From Extraction Ontologies to Java

We have developed an ontology compiler that translates from an ontology language (in this paper, we use OSM-L, which is a language for the OSM model) to Java. This compiler takes an extraction ontology as input, and outputs Java APIs automatically. It also helps ontology writers to find syntax errors which are usually hard to find by visual inspection. Other research such as [7] and [2] have also tried to automatically translate an ontology such as OWL or RDF to

Java APIs. These approaches, however, only focus on mapping—not compiling—certain desirable properties from the ontologies to corresponding executable Java code. In addition, they do not provide translating ontologies to Web services.

In the rest of this section we discuss how we use Java classes and methods to describe the knowledge represented in an ontology. In particular, we present our use of Java to describe the five major components of an extraction ontology: object sets, data frames, relationship sets, aggregation, and generalization/specialization. The first two map to atomic Web services (Section 4.1), and the rest to complex Web services (Section 4.2).

Object Set An object set describes information about a concept in a source ontology. In our system, an interface is generated automatically for each concept in the source ontology. We choose to use interfaces because Java only supports multiple inheritance through interfaces. An interface, however, can only provide static variables and abstract methods. We also design the compiler to generate an implemented class for each interface. For the car ontology in Figure 2, for example, there are 15 concepts. The ontology compiler generates 15 interfaces as well as 15 implemented classes. All of these interfaces use the same template as Figure 3 (a) shows[2]. Each interface declares five static data fields. The *name* variable stores the concept's name. The *type* variable specifies whether the concept is a primary concept. The *length* variable corresponds to the matching length in the ontology. The *dataFrame* vector stores all the extraction rules defined by the ontology and the *relationSet* stores all the relationships between this concept and any other concepts. There are also three static methods: *recognize*, *listRelation*, and *checkSuperClass*. The first is to recognize this concept from a source document depending on the extraction rules defined in the data frame, which we will discuss in Data Frame section. The second method is to store all the relationships that this concept participates in. All such relationships are stored in a vector called relationSet, which we will discuss in the Relationship Set section. The third one is to check all the generalization concepts of a concept, which we will discuss in the Generalization/Specialization section.

Data Frame There are two kinds of object sets: lexical objects and non-lexical ones. A non-lexical object set describes an abstract concept that does not have any value to extract, just like an interface cannot be instantiated. For a non-lexical object, its *dataFrame* is *null*, and its *recognize* method only has an empty body in the implemented class. For lexical objects, on the other hand, we want to save the information from their data frames and implement the *recognize* methods.

[2] The interface/class for each concept has its own name (the concept name for the interface name and the concept name concatenating an "impl" for the class name). We use *Concept* here to illustrate the template. We decided not to generate an overall concept class with each individual concept as an instance of this overall class. Because we want to provide an individual Web service for each concept, it is more convenient to make each concept an interface, which allows packing each concept separately. The basic Web service can use one concept interface and the implemented class without touching any other interface/class.

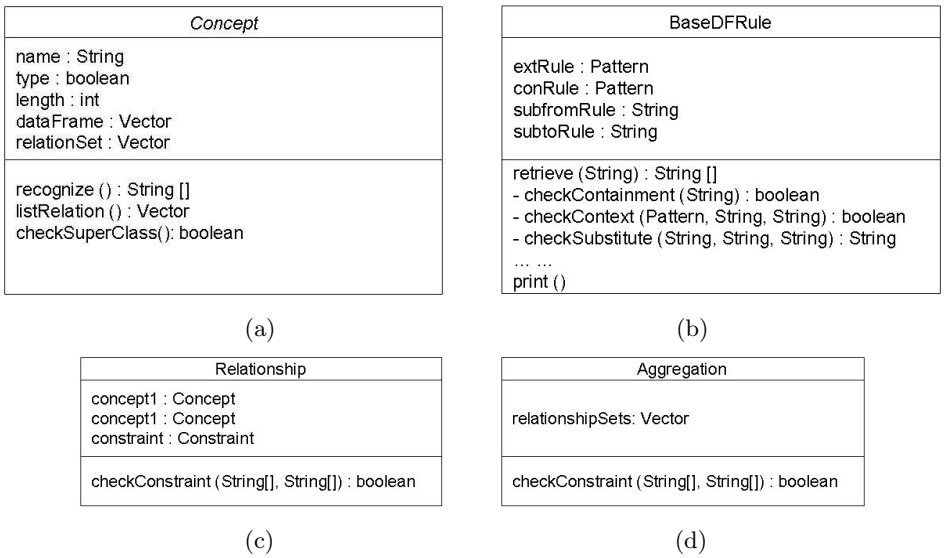

Fig. 3. The UML Diagrams for the Generated Java Classes

A data frame contains a set of extraction rules. Each extraction rule is an instance of the BaseDFRule class. Figure 3(b) shows the UML diagram of the BaseDFRule class. We use four data fields to store the four parts of an extraction rule: extraction pattern, context pattern, substituteFrom pattern, and substitueTo pattern. This class also contains methods that extract information according to the extraction rule. The method *retrieve* takes a string as input, finds all the substrings that match the extraction rule, and then returns the extracted results in an array. There is a set of private methods that help this process. How to extract the information via an extraction rule is beyond the scope of this paper; please refer to [3] for detailed information. In each concept implementation class, the vector *dataFrame* stores all the extraction rule instances contained in the data frame of this concept. The *recognize* method processes one extraction rule a time by calling its *retrieve* method, and recognizes all the strings that match the concept.

Relationship Set The ontology compiler generates a *Relationship* class to represent relationship sets in a source ontology. Figure 3(c) shows the UML diagram of the *Relationship* class. Each relationship set among two concepts is an instance of the *Relationship* class. The *Relationship* class has three data fields: the two concepts involved and the participation constraint among these two concepts. One method, *checkConstraint*, is implemented to check the participation constraints. This method takes the extraction results of the two concepts, and checks whether the constraints hold. For non-lexical object sets, we assume that the participation constraints hold automatically. Consider the relationship set *Car[0:1] has Year[1:*]* as an example. Since *Car* is a non-lexical object set, the code does not need to check the constraint *[1:*]*. The *Year* concept, on the other

hand, is a lexical object set, so we need to check if the constraint *[0:1]* holds. If the system retrieves more than one Year value from a source record, then the method will return false and the system will return a failure condition.

A concept might be involved in many relationship sets. The *listRelation* method in its corresponding concept implementation class stores all the Relationship instances that this concept directly involved in the *relationSet* vector.

Aggregation An aggregation has one or more is-subpart-of relationship sets. We use an Aggregation class to represent aggregation relationship sets. Each aggregation in an ontology is an instance of the *Aggregation* class. Figure 3(d) shows the UML diagram of the *Aggregation* class. The *relationshipSets* vector stores all the binary relationship sets (as in instances of the *Relationship* class). The *relationshipSets* Vector for the $Model|Trim$ aggregation, for example, stores two *Relationship* instances: *Model [1:*] is-subpart-of* $Model|Trim$ *[0:1]* and *Trim [1:*] is-subpart-of* $Model|Trim$ *[0:1]*. The method *checkContainment* checks whether the aggregation condition holds by checking whether each extracted value of the sub-concept is a substring of the super-concept. If not, the system will throw an error to the user.

Generalization/Specialization A generalization/specialization specifies an *is-a* relationship, identical to inheritance in Java. A specialization concept should inherit all the relationship sets its generalization concepts have. The interface of each specialization concept extends all its generalization concepts. Since Java supports multiple inheritance over interfaces, one child interface can extend more than one parent interface. In our car ontology, for example, the interfaces Engine, BodyType, Accessory, and Transmission extend the interface Feature. In an implementation class, the *listRelation* method also calls the parent class's *listRelation*, so that the parent's relation sets can be added to the child *relationSet* vector. The *checkSuperClass* is implemented to find all the super classes (generalization concepts) of a class (concept). The parent class list can be obtained though the getInterface method in the Class class in the Java standard class library.

4 Web Service Creation

Ontologies enable machine communication, and the Semantic Web is a typical environment that requires machine communication. Web services, on the other hand, are typical ways of performing machine communication. In the environment of the Semantic Web, Web services describe conventions for mapping knowledge easily and conveniently into and out of Web application programs. Hence we deem it desirable not only to generate Java APIs based on declarative domain knowledge in extraction ontologies, but also to directly generate machine-interpretable Web services using these generated Java APIs.

4.1 Atomic Web Services for Data Recognition

A Web service is a remote service that can not only satisfy a request from a client but can also be accessed via a standard specification interface. There

```
<definitions name='edu.byu.deg.examples.PriceService'
   ...
   xmlns:cps='http://www.deg.byu.edu/wsdl/edu.byu.deg.examples.PriceService/'
   ... >
   <message name='PriceRecognizer0SoapIn'>
      <part name='inputDoc' type='xsd:string' />
   </message>
   <message name='PriceRecognizer0SoapOut'>
      <part name='inputDoc' type='xsd:string' />
   </message>
   <portType name='edu.byu.deg.examples.PriceService'>
      <operation name='PriceRecognizer' parameterOrder='inputDoc'>
         <input name='PriceRecognizer0SoapIn' message='cps:PriceRecognizer0SoapIn'/>
         <output name='PriceRecognizer0SoapOut' message='cps:PriceRecognizer0SoapOut'/>
      </operation>
   </portType>
   ...
</definitions>
```

Fig. 4. Part of PriceService.wsdl.

is no need for a service requester to know the details about internal service implementation. Instead, a service requester only needs to know the input and output of an operation, and the location provided by the service provider, both of which are described in the Web service description language (WSDL)[3]. The central problem of creating Web services is, therefore, to create a WSDL file that specifies a desired Web service.

In general, we can create an atomic data recognition Web service for each concept in an ontology. As described above, there is a *recognize* method in the generated Java interface/class of each concept. We therefore build a Web service based on this method for each concept. These generated services are *atomic* because each of them recognizes data instances with respect to one and only one ontology concept. In essence, none of them should be further decomposable. Moreover, users can compose these atomic Web services to be more complex Web services.

Figure 4 shows an example of an atomic Web service. This service is automatically generated from a Java program that is built through the ontology compilation process presented in the last section. The WSDL file is created for the Price recognition Java class. The service describes a remotely executable method named "PriceRecognizer" that takes an argument named "inputDoc." Both the return datatype and the datatype of the input argument are "xsd:string". With this information, users can directly invoke the service to retrieve price values within the "inputDoc". When invoking the service, users do not need to know either the implementation details or even the specific programming language used for the implementation.

Several available tools can accomplish such transformations by directly generating WSDL files from Java classes. For instance, our example in Figure 4 is generated by the *java2wsdl* command-line tool contained in the GLUE plat-

[3] Web Services Description Language (WSDL), http://www.w3.org/TR/wsdl/

form[4]. GLUE also provides facilities that allow users to publish generated Web services that may be useful to other people.

4.2 Complex Web Services Through Composition

Complex services are composed with multiple simple atomic services. Web services represent loosely coupled interactions which are well suited to integrate disparate software domains and bridge incompatible technologies. It is therefore favorable to compose atomic Web services to accomplish complex operations, which is known as the process of Web service composition.

An essential requirement of automated service composition is that machine agents can interpret both the functionalities and the meanings of the operands in a basic Web service. Through a WSDL file, machines can execute a service correctly. The same WSDL descriptions, however, do not provide any explanation of the intent of a service or what the meanings of inputs and outputs are. Hence machines do not know what a Web service targets. This problem of lack of semantic explanation for Web service functionality has already been identified as a major problem for Web service research [1].

To solve this problem, we usually require Web services to be annotated before machines can automatically compose them. We use ontologies to denote the inputs, outputs, and parameters of a service. This solution is known as Web service annotation (such as [1], [5], [6], [8], and etc.). This type of service annotation operations, however, requires additional processing after services are generated. Not surprisingly, service annotation is not trivial.

Our method of automatically generating atomic Web services from extraction ontologies provides a resolution to this service interpretation problem. Because our generated services are based on ontologies from the onset, each of its generated features has its original formal definition in the starting extraction ontology. For this reason, the after-generation Web service annotation is no longer needed.

The simplest composition of atomic data recognition services involves retrieving a binary relation between two concepts. In an extraction ontology, we do not define relationship recognizer methods. Hence our relationship identification is based on a necessary but incomplete checking of a discovered relation. Our method determines a discovered binary relationship to be a defined relationship in an ontology when the following three conditions are satisfied simultaneously: (1) domain checking: the application domain matches a defined ontology; (2) concept-pair checking: the two concepts in the discovered binary relationship matches two concepts in the defined ontology; and (3) participation-constraint checking: the participation constraint of the discovered relation matches the participation constraint of the target relationship in the defined ontology. Although in theory this relationship-checking method is not complete enough to determine a relationship set, we find that it works very well in practice, especially when the scope of an application domain is narrow [3]. Because our research focuses

[4] The GLUE platform, http://www.themindelectric.com

on service composition involving the generated atomic ones, this narrow-domain assumption always holds.

We defer the domain-checking aspect to future discussions. In essence, we can view the domain checking problem as a standard ontology matching problem. Our previous experience in solving the ontology matching problem [4] allows us to simply assume prior knowledge about the ontology the user seeks.

With these discussions, we can reduce our generation problem for complex services to automatic composition of a service that captures two concepts and their participation constraints based on a known ontology. For example, assume we want to produce a complex service "car has price" as the ontology in Figure 2 shows. According to our relation-checking method, the dynamically created car-price complex service is composed of three basic services: the car recognition service, the price recognition service, and a check constraint service as Figure 3(c) shows.

More complex would be a recognition service for the relation "price for make." In the original ontology as Figure 2 shows, there are no explicitly declared links between the two concepts *Price* and *Make*. So the service composition method needs to perform a relation discovery process to retrieve an implicit relation between the two concepts that can be derived by declarative specifications of ontology relationships. In our example, we know both "car has price" and "car has make." Through a standard ontology inference process, we can derive a relation of "price for make" as well as its participation constraints. After the implicit relation is generated, our system can process it through the same complex service composition procedure as before.

As we discussed earlier, two typical relationships in extraction ontologies are aggregation and generalization/specialization. For an aggregation complex service request, in addition to the normal service composition procedure, we add a checkConstraint service as Figure 3(d) shows. For generalization/specialization, we add a checkSuperClass service as Figure 3(a) shows.

Although these examples are simple, we can perform the same composition process recursively. That is, each constructed service can be an unit, and we can perform a binary composition of any two constructed units. Therefore, this recursive service composition process can eventually produce very complex services based on users' requests. Even better, since in each iteration the new composed service has itself mapped to machine-interpretable formal semantics, the new composed service will be inherently machine-interpretable no matter how complicated it is. There is no need for additional service annotation processes in these automatically generated Web services.

5 Conclusion and Future Work

In this paper we have sketched a two-step process for creating Semantic Web services. The first, which is fully implemented, involves compiling extraction ontologies to Java code that represents atomic executable Web services. The second stage involves composing these Web services together recursively with

information derivable from the original ontologies. This additionally provides an automatic means for annotating these services in a standardized and formal manner.

Several directions remain to be pursued. First, we have yet to fully explore the second-stage composition process, particularly with respect to exploitation of the full range of ontological relationships possible and the resulting complexity and consistency of the derived services. The relative benefits and challenges to this approach versus less automated approaches is also unclear at this point.

Second, we have yet to develop a rigorous testing methodology for assessing how well the system can assure appropriate coverage of services, accuracy of results given user queries, and quantifiable reduction in annotation efforts over more traditional efforts. This would involve comparing the results with those obtained via other proposed ontology-based service frameworks.

Third, we intend to explore development of a comprehensive Semantic Web services environment that allows users to specify a query of interest, match it with pre-existing extraction ontologies of appropriate domain and coverage and thereby select which pre-existing services are the most appropriate. In the absence of pre-existing services, the tool would allow the user to select the ontologies that are the most relevant, supervise (if desired) their mapping to atomic services, and direct (again, if desired) their composition into more complex services which can then be executed to satisfy the user's request.

References

1. M. L. Brodie. Illuminating the dark side of Web services. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, pages 1046–1049, Berlin, Germany, September 2003.
2. A. Eberhart. Automatic generation of Java/SQL based inference engines from RDF schema and RuleML. In *Proceedings of the First International Semantic Web Conference on The Semantic Web (ISWC'02)*, pages 102–116, London, UK, June 2002.
3. D.W. Embley, D.M. Campbell, Y.S. Jiang, S.W. Liddle, D.W. Lonsdale, Y.-K. Ng, and R.D. Smith. Conceptual-model-based data extraction from multiple-record Web pages. *Data & Knowledge Engineering*, 31(3):227–251, November 1999.
4. D.W. Embley, L. Xu, and Y. Ding. Automatic direct and indirect schema mapping: Experiences and lessons learned. *SIGMOD Record*, 33(4):14 – 19, December 2004.
5. D. Fensel and C. Bussler. The Web service modeling framework WSMF. *Electronic Commerce: Research and Applications*, 1:113–137, 2002.
6. Andreas Heß, E. Johnston, and N. Kushmerick. ASSAM: A tool for semi-automatically annotating Semantic Web services. In *Proceedings of the 3rd International Semantic Web Conference (ISWC'04)*, Hiroshima, Japan, November 2004.
7. A. Kalyanpur, D. Pastor, S. Battle, and J. Padget. Automatic mapping of OWL ontologies into Java. In *Proceedings of Software Engeering and Knowledge Engeering (SEKE'04)*, Banff, Canada, June 2004.
8. A. Patila, S. Oundhakar, and K. Verna. METEOR-S Web service annotation framework. In *Proceedings International WWW Conference*, pages 553–562, New York, NY, May 2004.