13 Adding Semantics to Web Service Descriptions

We have discussed OWL-S in Chapter 12. Because we are more interested in the idea of automatic service discovery, we have concentrated more on the profile description of a given Web service. In fact, two different paths have been proposed for Semantic Web services, even for the purpose of automatic service discovery.

One path is to create systematic and stand-alone semantic descriptions based on some universally agreed ontologies; any given service can be described using these upper ontologies, and these descriptions will cover different aspects of the given service. An automatic agent will have enough information for the discovery, invocation, composition, and monitoring of this service. Clearly, OWL-S is such an upper ontology.

The other path is relatively lightweight: semantics are added to Web services by inserting semantic annotations into the current Web service standards, such as Universal Description, Discovery and Integration (UDDI) or Web Services Description Language (WSDL) documents. The main advantage of this path is the reuse of the related standards, given that these standards have been accepted by application developers and also enjoy wide support from different vendors and products.

To see the whole picture, we will concentrate on this lightweight approach in this chapter. We will cover two proposed methods in this area: WSDL-S and semantically enhanced UDDI.

13.1 WSDL-S

13.1.1 WSDL-S Overview

WSDL-S (Web Service Description Language Semantics) was developed by IBM and the University of Georgia, and presented to W3C as a member submission in late 2005 [51]. It is a proposal for marking up Web service descriptions with semantics. One main advantage of WSDL-S is the reuse of WSDL. Building upon an existing Web service standard such as WSDL, which has been a W3C standard since 2001, will make the added semantic layer more practical and easier to be adopted by application developers.

WSDL-S also depends on domain-specific ontologies. Its semantic annotations are added to different parts of a WSDL document by using domain ontologies. Another advantage of WSDL-S is that it does not limit the choice of the language in which the domain-specific ontology is constructed. On the other hand, this means that the soft agent responsible for processing the WSDL-S documents has to be smarter; it has to be capable of "understanding" several different ontology languages.

It is also important to know that WSDL-S mainly focuses on dynamic discovery of the services. In addition to semantically marking up inputs and outputs of an operation, it also adds the concepts of precondition and effect. However, it does not provide enough semantic information for automatic invocation, composition, and monitoring of a given service. In other words, the WSDL-S proposal focuses on adding semantics to the so-called abstract parts of a WSDL document, leaving the concrete parts untouched. Recall that in a given WSDL document, the abstract parts include `portType` (also named `interface` in the newer versions of WSDL), `operations`, and `messages`; the rest of the document comprises the concrete parts.

13.1.2 WSDL-S Annotations

The main constructs proposed in WSDL-S are summarized in Table 13.1. To see how semantic annotation is done, let us again take our `getMegaPixel` service as example. The first step is to annotate the `operation` element. This is done by adding an attribute that refers to some concept in a given ontology. In our example ontology, `Camera1.owl`, we have not defined any operation; but to see how this annotation is done, let us suppose that a `GetPixelOperation` operation has been defined in the ontology. At this step, the WSDL will be as shown in List 13.1.

We see that appropriate namespaces have been added (lines 9 and 10); also, the operation is annotated by a given concept in the ontology (line 38). In fact, this is also a good place to add the precondition and effect. In our simple example, there was no precondition and effect, but for the sake of completeness, List 13.2 shows how the precondition and effect can be added (lines 39 and 40 in List 13.2).

TABLE 13.1
Summary of the WSDL-S Annotation Constructs

Constructs	Meaning
`wssem:modelReference`	An extension element to allow for one-to-one associations of WSDL input and output type schema elements to the concepts in a domain-specific ontology
`wssem:schemaMapping`	An extension attribute to allow for many-to-many associations of WSDL input and output type schema elements to the concepts in a semantic model — typically associated with XML schema complex types
`wssem:precondition` and `wssem:effect`	Two elements used as child elements of the `operation` element that describe the semantics of the `operation` by specifying the preconditions and effects; primarily used in service discovery
`wssem:category`	An extension attribute used on `interface` (`portType`) element; it consists of service categorization information that could be used when publishing a service in a Web services registry, such as UDDI

wssem represent namespace:
`http://www.ibm.com/xmlns/stdwip/Web-services/WS-Semantics`

LIST 13.1
WSDL-S Document for Our `getMegaPixel` Service, Step 1 of 2

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:s="http://www.w3.org/2001/XMLSchema"
             xmlns:s0="http://tempuri.org/"
             xmlns:soapenc="http://schemas.xmlsoap.org/soap/
                   encoding/"
             xmlns:tm="http://microsoft.com/wsdl/mime/
                   textMatching/"
             xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
             xmlns:wssem
                  ="http://www.ibm.com/xmlns/WebServices/
                   WS-Semantics"
             xmlns:camera
                  ="http://www.yuchen.net/photography/Cameral.owl"
              targetNamespace="http://tempuri.org/"
             xmlns="http://schemas.xmlsoap.org/wsdl/">

   <types>
    ... no change here from List 11.2 ...
   </types>

   <message name="getMegaPixelSoapIn">
      <part name="parameters" element="s0:getMegaPixel" />
   </message>
   <message name="getMegaPixelSoapOut">
      <part name="parameters" element="s0:getMegaPixelResponse" />
   </message>

   <portType name="Service1Soap">
      <operation name="getMegaPixel"
       wssem:modelReference="camera:GetPixelOperation">
         <input message="s0:getMegaPixelSoapIn" />
         <output message="s0:getMegaPixelSoapOut" />
      </operation>
   </portType>

   <binding name="Service1Soap" type="s0:Service1Soap">
    ... no change here from List 11.2 ...
   </binding>

   <service name="Service1">
    ... no change here from List 11.2 ...
   </service>

</definitions>
```

LIST 13.2
WSDL-S Document Presented in List 13.1 with Precondition and Effect Added

```
same as List 13.1

37:    <portType name="Service1Soap">
38:       <operation name="getMegaPixel"
           wssem:modelReference="camera:GetPxielOperation">
39:          <wssem:precondition name="someNameForCondition"
              wssem:modelReference="camera:somePreconditionConcept">
40:          <wssem:effect name="someNameForEffect"
              wssem:modelReference="camera:someEffectConcept">
41:          <input message="s0:getMegaPixelSoapIn" />
42:          <output message="s0:getMegaPixelSoapOut" />
43:       </operation>
44:    </portType>

same as List 13.1
```

The next step is to add annotation for input and output. These annotations can be added to the XML schema definitions of the message-type elements. In our example, it is again quite simple; List 13.3 shows the updated WSDL document (lines 18 and 25).

LIST 13.3
WSDL-S Document for Our `getMegaPixel` Service, Step 2 of 2

```
1:  <?xml version="1.0" encoding="utf-8"?>
2:  <definitions xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
3:               xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
4:               xmlns:s="http://www.w3.org/2001/XMLSchema"
5:               xmlns:s0="http://tempuri.org/"
6:               xmlns:soapenc="http://schemas.xmlsoap.org/soap/
                 encoding/"
7:               xmlns:tm="http://microsoft.com/wsdl/mime/
                 textMatching/"
8:               xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
9:               xmlns:wssem
                      ="http://www.ibm.com/xmlns/WebServices/
                       WS-Semantics"
10:              xmlns:camera
                      ="http://www.yuchen.net/photography/Cameral.owl"
11:               targetNamespace="http://tempuri.org/"
12:              xmlns="http://schemas.xmlsoap.org/wsdl/">

13:    <types>
14:       <s:schema elementFormDefault="qualified"
```

```
                 targetNamespace="http://tempuri.org/">
          <s:element name="getMegaPixel">
             <s:complexType>
                <s:sequence>
                   <s:element name="cameraModel"
                  wssem:modelReference="camera:model" type="s
                  :string"/>
                </s:sequence>
             </s:complexType>
          </s:element>
          <s:element name="getMegaPixelResponse">
             <s:complexType>
                <s:sequence>
                   <s:element name="getMegaPixelResult"
                  wssem:modelReference="camera:pixel" type="s
                  :double"/>
                </s:sequence>
             </s:complexType>
          </s:element>
       </s:schema>
    </types>

    <message name="getMegaPixelSoapIn">
       <part name="parameters" element="s0:getMegaPixel" />
    </message>
    <message name="getMegaPixelSoapOut">
       <part name="parameters" element="s0:getMegaPixelResponse" />
    </message>

    <portType name="Service1Soap">
       <operation name="getMegaPixel"
        wssem:modelReference="camera:GetPxielOperation">
          <wssem:precondition name="someNameForCondition"
           wssem:modelReference="camera:somePreconditionConcept">
          <wssem:effect name="someNameForEffect"
           wssem:modelReference="camera:someEffectConcept">
          <input message="s0:getMegaPixelSoapIn" />
          <output message="s0:getMegaPixelSoapOut" />
       </operation>
    </portType>

    <binding name="Service1Soap" type="s0:Service1Soap">
     ... no change here from List 11.2 ...
    </binding>

    <service name="Service1">
     ... no change here from List 11.2 ...
    </service>

</definitions>
```

In general cases, however, you have several choices. First, if a WSDL `type` element uses a simple type (such as `string` or `integer`), you can directly map it to the ontology concept (class or property), as shown in List 13.3. If this element is a complex type, you can either map the whole element to an ontology concept, or you can individually annotate each leaf element of this complex element.

For instance, you can do the following to map each element:

```
<complexType>
   <all>
      <element name="leaf-node-1" type="xsd:integer"
              wssem:modelReference="camera:someConcept1">
      <element name="leaf-node-2" type="xsd:string"
              wssem:modelReference="camera:someConcept2">
      <element name="leaf-node-3" type="xsd:integer"
              wssem:modelReference="camera:someConcept2">
   </all>
</complexType>
```

To map the whole element to some concept, we can do the following:

```
<complexType wssem:modelReference="camera:someConcept">
   <all>
      <element name="leaf-node-1" type="xsd:integer"/>
      <element name="leaf-node-2" type="xsd:string"/>
      <element name="leaf-node-3" type="xsd:integer"/>
   </all>
</complexType>
```

Finally, you can add some semantics to the service categorization. The main purpose of adding categorization information is to facilitate discovery of the service. Without listing the whole WSDL document again, you can just add the following to the `portType` (`interface`) element, as shown in List 13.3.

```
<portType name="Service1Soap">
   <wssem:category name="On-Line Information Service"
    taxonomyURI="http://www.naics.com/" taxonomyCode="514191" />
   <operation name="getMegaPixel"
      ... details of operation ...
   </operation>
</portType>
```

Now that we have covered the main constructs in WSDL-S, you can appreciate the simplicity of this approach. In the next section, we will take a look at how the annotated information is used for dynamic service discovery.

13.1.3 WSDL-S and UDDI

The advantage of adding semantics into WSDL is the reuse of current available standards and relevant tools. More specifically, tools built based on the standard WSDL will continue to function and, meanwhile, semantic-aware tools can be created to take advantage of the annotations.

To further solve the problems with automatic discovery, it is clear that the service description must be published in some registry; to continuously promote reusability, this registry also has to be common. One good choice along this path would be the UDDI registry.

The Organization for the Advancement of Structured Information Standards (OASIS) has published a recommended mapping schema that implements the mapping of WSDL-S into the UDDI data structure [52], but semantics still have to be added separately into the UDDI registry. What makes this hard is that there is still no unified method to publish a WSDL file to UDDI nor is there a unified way to discover a WSDL file.

Yet, an example of mapping WSDL-S to UDDI is presented in Reference 53. You can find more details in there, but here is a summary of the proposed method:

1. UDDI is enhanced by designing a detailed mapping from WSDL-S to UDDI data structures. More specifically, a WSDL-S service is captured using `businessEntity` in UDDI, whereas `portType` and each `operation` within the WSDL-S service are captured using `tModels`.
2. Tools are then developed to automatically map the WSDL-S document into the enhanced UDDI registry and to further discover the requested services.

We are not going to discuss the details of this method. However, we will discuss the details of another earlier and more mature approach, which takes OWL-S as the starting point and maps it to a UDDI data structure. We will also take a look at the discovery process (matchmaking engine) built upon this mapping. This will give us a much clearer picture of how the added semantics can help achieve the goal of automatically discovering the requested services.

13.2 OWL-S TO UDDI MAPPING

As we have discussed earlier, to semantically describe a given Web service you can either choose the OWL-S upper ontology as a full solution, or you can chose lightweight ones such as WSDL-S to add semantics to your service description. But no matter which one you chose, to facilitate automatic discovery, you still have to somehow collect all these descriptions into a registry.

UDDI, as such a registry, has to be enhanced to hold the added semantics. In this section, we will discuss one such enhancement to the UDDI registry that allows the mapping from OWL-S to a UDDI structure. The purpose of this discussion is to give you a concrete example of how this is done in the real world.

As you will see, `tModel`, one of the major data structures in UDDI, will play the key role in this mapping process. Therefore, a sound understanding of `tModel` is a must. For this reason, before we get into the mapping details, we will first review `tModel` in the next section.

13.2.1 More About UDDI `tModels`

For many developers, the `tModel` concept is just like the XML namespace concept; it is not complex at all, yet it can be very confusing in the beginning. In this section, we will revisit the `tModel` concept to understand its three different roles in the UDDI structure. This understanding is necessary for the mapping from OWL-S to UDDI.

13.2.1.1 `tModel` and Interface Representation

Obviously, UDDI is an online yellow book that is used by both service providers and service consumers. The service providers will register their Web services into UDDI, and service consumers will use UDDI to search for the service they want.

The idea of interface in the world of UDDI is more or less similar to the concept of interface in the world of COM/DCOM; i.e., it is based on a contract that both the service provider and the service consumer will honor: the service provider promises to implement the Web service in such a way that if the consumer invokes the service by following this contract, his or her application will get what it expects.

It is also important to know a fundamental rule of UDDI: when a service provider wants to register a Web service with UDDI, he or she must guarantee that the service would implement an interface and, furthermore, the interface has to be already registered with UDDI.

Note that the interface a Web service implements need not be defined by the service provider. For instance, some major airlines may get together and form a committee that will work out and publish (register) an interface in UDDI for querying the ticket price on a given date, time, and city pairs. This published interface will become the industrial standard, and the implementation work will be left to each specific airline. Each airline will then develop a Web service that implements the interface. It will then register the service with UDDI. In this case, the interface is not defined by the airline that implements it. Also, it is quite obvious that the life of a travel agent will become quite easy: although we have quite a few different airlines, there will be only one querying interface.

It may certainly be true that the Web service a given provider wants to register with has no standard interface at all; in that case, the provider will have to first create and register an interface with UDDI. After this interface is registered, the service that implements it can be developed.

By now, one should be able to tell how important an interface is in UDDI. But how does an interface exist in UDDI? In what kind of language is the interface described? The answer underscores the primary role of `tModel`: every interface in UDDI is represented by a `tModel`.

Recall that in Chapter 11, when we registered our `getMegaPixel` service into UDDI, we had to create a `tModel` first to represent a service type. A service type is nothing but another word for interface. In our case, we assumed there was no current standard for the `getMegaPixel` service; i.e., there was no existing interface we could develop our service against. Therefore, we had to create our own interface first. To recapitulate, the `tModel` interface is shown in List 11.9.

This `tModel` represents the interface of the Web service we are going to develop, and it is really not complex at all. After we register the preceding interface, we can go ahead and develop the service that implements this interface and further register it into UDDI.

In Chapter 11, after the discussion about using `tModels` to represent interfaces, we continued to discuss the need to add more information into the interfaces to help us find the desired services easily. Recall that the UDDI solution is to add categorization information into interfaces (and other UDDI data structures) to make them members of one or more predefined categories; one can therefore find the desired services based on some classification scheme.

The next question, then, is how can we describe these classification schemes; i.e., what kind of constructs can we use to represent them so we can easily add them into `tModels` (and other data structures for that matter)? You might remember what we did: we used `tModels` to represent these categorization schemas. This leads us into the second role of `tModels`, which is discussed in the next section.

13.2.1.2 `tModel` and Categorization to Facilitate Discovery of Web Services

Recall that the `tModels` used to represent classification schemes are predefined, and one can add one or more of these `tModels` as needed into the interface representation. Because the interface itself is represented by a `tModel`, it is interesting that we are inserting a `tModel` (child) into another `tModel` (parent), but the parent `tModel` represents an interface, whereas the child `tModel` represents some classification schemes. We can see that the `tModel` is a very flexible data structure, which can be used to represent quite different entities.

To recapitulate, list 11.12 is our enhanced version of the `tModel` representing the `getMegaPixel` interface, and it uses two well-known (preregistered) `tModels`. You can find the details of these predefined `tModels` in Chapter 11.

Now we are ready to discuss the last role of `tModel`: they can be used to represent namespaces. This will play the key role in the process of mapping OWL-S description into UDDI structures. Let us discuss this role in greater detail.

13.2.1.3 `tModel` and Namespace Representation

Consider the following scenario: A developer has used a smart way to search the UDDI registry and has finally found our `getMegaPixel` interface (List 11.12); his next step is to decide whether the Web service that implements this interface is the service he wants. By studying the interface, he cannot really make any decision: the information is too limited. He has to go to the WSDL file pointed to by this interface for further investigation. Now the question is, can we provide him with more information about exactly what this service does?

One thing we can do is to add some information into this interface to tell the potential users what inputs this service will require and what outputs it will produce. This will help them decide whether the service that implements this interface is the right one for them. This can be implemented by again adding a `keyedReference` structure into the `categoryBag` element, as shown in List 13.4.

Let us assume that this enhancement is a big success: application teams inside and outside the company love this new feature, many developers start using the Web service, and this brings a lot of money into the company. Therefore, the company's

LIST 13.4
Interface `tModel` with `input` and `output` References

```
<tModel tModelKey="uuid:5DD52389-B1A4-4fe7-B131-0F8EF73DD175">
   <name>getMegaPixel</name>
   <description>interface for camera Web service</description>
   <overviewDoc>
      <description xml:lang="en">URL of WSDL document</description>
      <overviewURL>
         http://localhost/chap11/GetMegaPixelWS/Service1.wsdl
      </overviewURL>
   </overviewDoc>
   <categoryBag>
      <keyedReference
          tModelKey="UUID:C1ACF26D-9672-4404-9D70-39B756E62AB4"
          keyName="specification for a Web service described in WSDL"
          keyValue="wsdlSpec"/>
      <keyedReference
          tModelKey="UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"
          keyName="On-Line Information Services"
          keyValue="514191"/>
    <keyedReference
          keyName="input"
      keyValue="xsd:string"/>
    <keyedReference
      keyName="output"
      keyValue="xsd:double"/>
   </categoryBag>
</tModel>
```

management team decides that from now on, every new interface that they register with UDDI should have this new enhancement. Now, problems start to show up: different developers in the company use different names for input; one developer names it `input`, another calls it `myInput`, and yet another uses `input-0` to represent input in his interface definition. The same problems can occur with output, too.

Now, not only is the development team confused, the outside world is also frustrated: there is no uniform naming scheme, so there is no way to search for the input or output to find the requested service, because it can have any name.

How can we solve this problem? Probably you know the answer already: use a `tModel`. List 13.5 is the hypothetical `input_tModel` we created and registered with UDDI. After we create this `input_tModel` and register it with UDDI (we should also do the same for the `output_tModel`), we do not have to worry about the different names used; you can use whatever name you want, as long as you reference the `input_tModel` and `output_tModel` keys when you add your input and output descriptions into the interface. List 13.6 is the latest and greatest interface of our `getMegaPixel` service.

LIST 13.5
The Definition of `input_tModel`

```
<tModel tModelKey="uuid:E27972D8-717F-4516-A82D-B688DC70170C">
 <name>input_tModel</name>
 <description xml:lang="en">namespace of input_tModel</description>
 <overviewDoc>
   <description xml:lang="en">
         whatever description you want
      </description>
   <overviewURL>
      http://www.ourCompany.com/internalDocuments/inputDefinition
      .html
      </overviewURL>
</overviewDoc>
</tModel>
```

LIST 13.6
Interface `tModel` with `input_tMode` and `output_tModel` References

```
<tModel tModelKey="uuid:5DD52389-B1A4-4fe7-B131-0F8EF73DD175">
   <name>getMegaPixel</name>
   <description>interface for camera Web service</description>
   <overviewDoc>
      <description xml:lang="en">URL of WSDL document</description>
      <overviewURL>
         http://localhost/chap11/GetMegaPixelWS/Service1.wsdl
      </overviewURL>
   </overviewDoc>
   <categoryBag>
      <keyedReference
          tModelKey="UUID:C1ACF26D-9672-4404-9D70-39B756E62AB4"
          keyName="specification for a Web service described in WSDL"
          keyValue="wsdlSpec"/>
      <keyedReference
          tModelKey="UUID:C0B9FE13-179F-413D-8A5B-5004DB8E5BB2"
          keyName="On-Line Information Services"
          keyValue="514191"/>
    <keyedReference
          tModelKey="uuid:E27972D8-717F-4516-A82D-B688DC70170C"
          keyName="whatever-input-name-you-want"
          keyValue="xsd:string"/>
    <keyedReference
          tModelKey="uuid:key_for_output_tModel"
          keyName="output"
          keyValue="xsd:double"/>
   </categoryBag>
</tModel>
```

As you can see, the problem is solved. A more interesting fact is that the input_tModel is used as a namespace: a group of developers now has a common and shared concept of the input (and output). For the outside world, let us assume that one developer in some other company knows this enhancement; he will first search for a tModel using the name input_tModel. Once he gets the result back, he can retrieve the key for this input_tModel. Next, he needs to find all the interfaces that use this tModel key in their definitions. The same is true for the output_tModel. The result: he gets all the interfaces that have this enhancement.

Up to this point, we have discussed the three main roles played by tModels in a given UDDI registry. Now we are ready to discuss the mapping of OWL-S into UDDI structures, which will make heavy use of tModels.

13.2.2 Mapping OWL-S Profile Information into the UDDI Registry

The mapping mechanism we are going to discuss here is the work of researchers and students at the Robotics Institute at Carnegie Mellon University [54]; you can find more details of this mapping and related publications at their Web site, http://www.daml.ri.cmu.edu/matchmaker/.

The mapping from OWL-S profile document to the UDDI registry is done on a one-to-one basis, as summarized in Table 13.2. Here, the left column contains the elements from OWL-S, and the right column contains the elements from UDDI data structures. Any given row shows the mapping relationship. If for a given row the mapping does not exist, the corresponding cell in the left or the right column is left blank ("na" is used in Table 13.2 to signify this case). Also, if an OWL-S profile element has a corresponding UDDI element (for example, the contact information is present in both the OWL-S profile and UDDI), the mapping is a direct connection between these two elements. For OWL-S profile elements with no corresponding UDDI elements, tModel-based mapping is used.

Based on the previous discussion of UDDI tModels (Section 13.2.1), this tModel-based mapping should be easily understood. The basic idea is to create specialized UDDI tModels for each unmapped element in the OWL-S profile, such as OWL-S Input, Output, and so on. These tModels have to be first created and registered with UDDI before the mapping can use them. These specialized tModels are in fact used as namespaces in the mapping process. Table 13.3 shows the description of UDDI's input_tModel, which is created and registered to represent the input element from the OWL-S profile document.

Using our getMegaPixel service as example, its UDDI entity can be updated, as shown in List 13.7 (Note the added information is in the categoryBag of the businessService entity, lines 27 to 30.) You can map the rest of the elements by following the same steps. As you can see, you need to create about 15 tModels to accomplish this. However, the final result is a semantically enhanced UDDI.

TABLE 13.2
Mapping Between OWL-S Profile and UDDI

OWL-S Profile Elements	UDDI Elements
na	BusinessEntity:Name
contactInformation:name	BusinessEntity:Contact:person name
contactInformation:title	na
contactInformation:phone	BusinessEntity:Contact:phone
contactInformation:fax	na
contactInformation:email	BusinessEntity:Contact:email
contactInformation:physicalAddres	BusinessEntity:Contact:address
contactInformation:webURL	BusinessEntity:discovery URLs
serviceName	BusinessService:name
textDescription	BusinessService:description
hasProcess	BusinessService:categoryBag:hasProcess_tModel
serviceCategory	BusinessService:categoryBag:serviceCategory_tModel
serviceParameter	BusinessService:categoryBag:serviceParameter_tModel
qualityRating	BusinessService:categoryBag:qualityRating_tModel
input	BusinessService:categoryBag:input_tModel
output	BusinessService:categoryBag:output_tModel
precondition	BusinessService:categoryBag:precondition_tModel
effects	BusinessService:categoryBag:effect_tModel
serviceProduct	BusinessService:categoryBag:serviceProduct_tModel
serviceClassification	BusinessService:categoryBag:classification_tModel

TABLE 13.3
UDDI's `input_tModel`

Name	`input_tModel`
Key	`uuid_of_input_tModel`
Technical Model Descriptions	This preregistered `tModel` represents the `input` element in an OWL-S profile document
Overview URL	Some other document describing this `tModel`

LIST 13.7
UDDI Entry of Our `getMegaPixel` Service

```
<businessEntity
   businessKey="uuid:AAAAAAAA-AAAA-AAAA-AAAA-AAAAAAAAAAAA"
   operator="someOperatorName"
   authorizedName="somePeronsName">
  <name>someCompanyName</name>
  <discoveryURLs>
    <discoveryURL useType="businessEntity">
       http://www.someWebSite.com/someDiscoveryLink
    </dicoveryURL>
  </discoveryURLs>
  <businessServices>
    <businessService
      serviceKey="uuid:BBBBBBBB-BBBB-BBBB-BBBB-BBBBBBBBBBBB"
      businessKey="uuid:AAAAAAAA-AAAA-AAAA-AAAA-AAAAAAAAAAAA">
      <name>getMegaPixel</name>
      <description>returns the mega-pixel for a model</description>
      <bindingTemplates>
        <bindingTemplate
         bindingKey="uuid:CCCCCCCC-CCCC-CCCC-CCCC-CCCCCCCCCCCC"
         serviceKey="uuid:BBBBBBBB-BBBB-BBBB-BBBB-BBBBBBBBBBBB">
          <accessPoint URLType="http">
           http://localhost/chap11/GetMegaPixelWS/Service1.asmx
          </accessPoint>
          <tModelInstanceDetails>
            <tModelInstanceInfo
             tModelKey="uuid:5DD52389-B1A4-4fe7-B131-0F8EF73DD175">
              <instanceDetails>
                <overviewDoc>
                  <overviewURL>
                  http://localhost/chap11/GetMegaPixelWS/Service1
                  .wsdl
                  </overviewURL>
                </overviewDoc>
              </instanceDetails>
            </tModelInstanceInfo>
          </tModelInstanceDetails>
        </bindingTemplate>
      </bindingTemplates>
      <categoryBag>
        <KeyedReference
          keyName="input"
          keyValue=camera:model
          tModelKey="uuid_of_input_tModel"/>
        <KeyedReference
          keyName="output"
          keyValue=camera:pixel
          tModelKey="uuid_of_output_tModel"/>
      </categoryBag>
   </businessService>
  </businessServices>
</businessEntity>
```

13.2.3 Issues of Mapping OWL-S Profile Information into UDDI Registry

There are, however, several issues with the foregoing mapping procedure; let us discuss them briefly in this section. In your real-world practice, you might have come across these issues already.

The first issue is, how to inform a soft agent that a given service advertisement has an OWL-S profile representation. UDDI is essentially a huge database holding a vast amount of service advertisements. If a given UDDI registry is semantically enhanced, some of these advertisements must have used the predefined `tModels`, as we have already discussed. Now, for each service advertisement, the agent has to first see whether it is a semantically marked-up advertisement; if it is not, the agent will simply skip it. But how will the agent know if this service advertisement is semantically marked up or not? As of now, the only way is to look for the specialized `tModels` discussed in the previous section. Clearly, this could be very inefficient.

A better solution is to have a single flag for the agent to read. We can create and register another `tModel` called `OWL-S_tModel`, which has a special meaning: it states that the service using this `tModel` has been semantically marked up and, furthermore, its value can be the URL of the OWL-S profile document. Therefore, not only have the elements in the OWL-S profile document been mapped into some UDDI service entity, but the service entity also has a link pointing back to its original OWL-S document.

There is another issue related to the specialized `tModels`. UDDI itself has provided several specialized `tModels`; one such example is the NACIS `tModel` used for categorization. UDDI has made these `tModels` well known to the users, so anyone who wants to add categorization information to his or her service advertisement can use these predefined `tModels` freely. Similarly, all the `tModels` representing the OWL-S elements (such as `input` and `output`) should also be made well known to the users. Curently, however, it would be unclear how this could be accomplished. Therefore, different users may register their own `input_tModels` with UDDI, and a given soft agent would have a hard time with these repeated or inconsistent `tModels`.

One solution to this problem is to always look for specialized `tModels` before you create you own. For instance, when you are implementing the mapping from OWL-S to UDDI structure, you should first search for the `input_tModel`, the `output_tModel`, or any other specialized `tModel`. If you can find such `tModels`, reuse them; if not, create your own.

13.3 MATCHMAKING ENGINES

Let us now see how far we are from the goal of automatic discovery of Web services. So far, we have collected everything into a centralized registry (UDDI), and this registry is semantically enhanced. Obviously, the next necessary piece is a matching algorithm that a soft agent can use to discover the desired service by sifting through all the service entries in a given UDDI registry. The software piece that implements a given matching algorithm is called a matchmaking engine. A semantically enhanced UDDI, together with a matchmaking engine, will indeed bring us much closer to the goal of automatic discovery of requested services. In this section, we will discuss some basic concepts of a matching algorithm. In Chapter 15, we will design one such algorithm and construct a matchmaking engine using Java.

First, we need to be aware that currently almost all the matchmaking algorithms are based on matching IOPEs (`Input`, `Output`, `Precondition`, and `Effect`), i.e., checking whether the IOPEs of the request match the ones from the providers. Clearly, the OWL-S profile document together with part of the process document will be able to provide enough information for these matchmaking algorithms. In cases where all the input, output, precondition, and effect descriptions are included in the profile documents, we do not even need the corresponding process documents.

In fact, most matching algorithms only consider the inputs and outputs, without worrying about the preconditions and effects. More sophisticated algorithms may consider every aspect of IOPE and also take some categorization information into account.

Another key concept in any matching algorithm is the concept of degree of match; it describes the degree of matching between two concepts. More precisely, the matching between two concepts is not syntactic, but is based on the relation between these concepts in their OWL ontologies. A matching algorithm normally recognizes the following four degrees of matching between two concepts:

Exact matching: Two concepts exactly match each other if they are the same concepts; i.e., if concept A subsumes concept B and concept B subsumes concept A. For example, if one service advertised to have `camera:SLR` as output and a service request asks for `camera:SLR` as output, these two output concepts exactly match each other.

Plug-in matching: If concept A subsumes concept B, then concept A is a set that includes concept B; in other words, concept A could be plugged in place of concept B. For example, consider an advertisement of a service with output as `camera:Digital`, and a request whose output is specified as `camera:SLR`. Although `camera:SLR` does not exactly match `camera:Digital`, this service could still be a candidate because `camera:Digital` subsumes `camera:SLR`.

Subsume matching: This is similar to the aforementioned situation, except that the request's concept subsumes the advertised concept. In this case, the service may not satisfy the needs of the request, but it is still a possible candidate.

Fail matching: In this case, there is no exact matching, and there is no subsumption relationship, either. In other words, the two concepts are unrelated and a failed match is returned.

Note that these degrees of matching are organized along a discrete scale in which an exact match is preferable to any other match, `plugIn` is better than `subsume`, and `fail` is certainly the worst.

In Chapter 14, we will take one step closer to the automatic discovery of Semantic Web services; we will also design and implement such a matching algorithm. Then, you will have a much better understanding of all that we have discussed.