# 12 OWL-S: An Upper Ontology to Describe Web Services

As discussed earlier, Semantic Web services are Web services with explicit semantic annotation. As usual, in order to semantically markup a given Web service description, a markup language has to be available. Therefore, before the annotation can occur, we need to learn such a language.

OWL-S[49] is currently the standard for Web service annotation. There are other languages available, but in this chapter, let us study this language in detail.

## 12.1 WHAT IS UPPER ONTOLOGY?

Let us think about the process of semantically marking up a Web service. Different Web services may offer different functionalities, take different input parameters, and certainly return different output parameters too. Nevertheless, each of these Web services can be described by answering some common and general questions such as the following:

- What does this Web service do?
- How does it do it?
- How is it invoked?

These questions have nothing to do with any special domains and do not require any assumptions related to any given Web service.

If we create an ontology to provide terms that can be used to answer these general questions, it will also be a general ontology: the classes and properties defined in this ontology should not be related to any specific domain or any special assumptions.

This is quite different from a normal ontology, which is always tied to a particular knowledge domain. If a given ontology is not tied to a particular domain and also attempts to describe general classes and properties, it is called a *foundation ontology* or *upper ontology.*

OWL-S [49] is such an upper ontology. OWL-S stands for "Web Ontology Language — Services"; it is written using OWL, and its goal is to provide general terms and properties to describe Web services.

## 12.2 THE CONCEPT OF OWL-S

### 12.2.1 Overview of OWL-S

OWL-S is an upper ontology for services. At the time of this writing, OWL-S has been submitted to W3C for consideration as a standard. It defines classes and properties that one can use to describe the following three essential aspects of a service (we have mentioned these aspects in the previous section, but here is a more detailed and precise rundown):

- What does the service do?
  The OWL-S upper ontology contains a subontology called *Profile ontology* (`profile.owl`) to define classes and properties to answer this type of question. This subontology is mainly used to advertise the service, thereby enabling a service requester to determine whether the given service meets the needs or not.
- How does the service work?
  OWL-S upper ontology's second subontology, *Process ontology* (`process.owl`), defines all the terms you need to describe how the service works. More precisely, you describe how the service works by describing the procedures necessary to interact with the service from a client's point of view. In other words, this subontology is used to describe how it should be used.
- How is the service invoked?
  The third subontology, namely, *Grounding ontology* (`grounding.owl`), is included by OWL-S to provide terms that one can use to describe how the service can be accessed technically. This includes the terms to describe the supported protocol and the exchanged message formats and other related low-level information.

The preceding subontologies are the main pieces OWL-S has included for the purpose of describing a Web service. To connect these three subontologies together, OWL-S introduces one final higher-level ontology called *Service ontology* (`service.owl`) to describe how these three pieces work together to completely describe a Web service.

The good news is that all these subontologies are written using OWL, a language we have covered already, and they are not large documents either. If you read these documents, you will quickly come up with the structure shown in Figure 12.1. This structure represents the world of Web services as far as OWL-S is concerned.

Note that when we describe a Web service, we will not be using any terms from `service.owl`; we will mainly use terms from the other three ontologies (`profile.owl`, `process.owl`, and `grounding.owl`). As shown in Figure 12.1, `service.owl` exists only to describe the semantic relationships among the other three ontologies.

It is also important to understand the obvious difference between OWL and OWL-S: OWL is an ontology language; it provides constructs and features we can use to create an ontology document. OWL-S, on the other hand, is just one such ontology

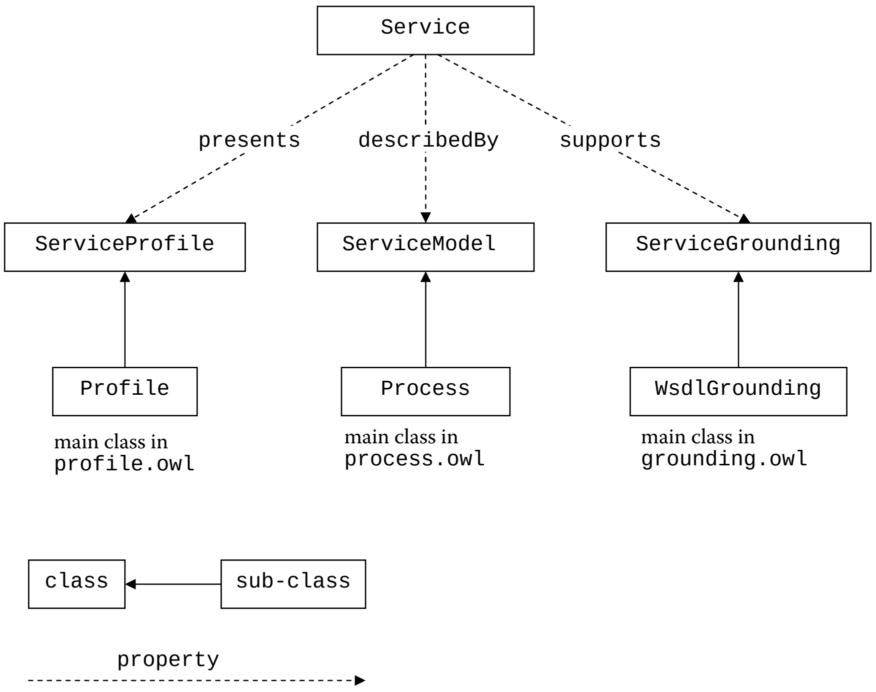

**FIGURE 12.1** The OWL-S upper ontology for describing Web services.

created by using OWL. In other words, learning OWL-S is not about learning a new language; instead, it is about understanding and using three new ontologies.

### 12.2.2 How Does OWL-S Meet Expectations?

We have earlier discussed the expectations the real world has regarding Web services in the previous sections. To recapitulate, a soft agent should be able to do the following to make Web services appeal to the application world:

- Automatic discovery of Web services
- Automatic invocation of Web services
- Automatic composition of Web services
- Automatic monitoring of Web services

The key question is, how does OWL-S manage to meet all these expectations? The answer is summarized in Table 12.1.

Table 12.1 should motivate you to continue your exploration of OWL-S, which will be covered in the rest of this chapter. Again, we will concentrate on the automatic discovery of Web services; automatic invocation, composition, and monitoring are beyond the scope of this book.

**TABLE 12.1**
**OWL-S and Web Services**

| Objectives | OWL-S Subontology |
|---|---|
| Automatic discovery of Web services | `Profile.owl` provides terms that can used to describe what the service does, so this can be used to find the requested service |
| Automatic invocation of Web services | `Grounding.owl` offers the terms that can be used to describe the technical details about how to access the service; therefore, this can be used to dynamically invoke a given service |
| Automatic composition of Web services<br>Automatic monitoring of Web services | `Process.owl` provides terms we can use to describe how the service should be used. Therefore, this information can be used to compose service descriptions from multiple services to accomplish a specific task, and this information can also be used to monitor the execution of a given service |

## 12.3 OWL-S BUILDING BLOCKS

### 12.3.1 OWL-S `Profile` Ontology

As discussed earlier, the purpose of describing the profile of a given service is to provide enough information so that a soft agent can decide whether the given service meets the requester's need. The OWL-S upper ontology's `profile.owl` document provides terms that can be used to accomplish this goal.

An OWL-S profile describes a service as a collection of four basic types of information:

- Service information and contact information: the name of the service, descriptions, and the organization that provides the service
- Functional description: the input/output/precondition/effect (IOPE) of the service
- Classification information of the service: for instance, the UDDI classification schemas discussed in Chapter 11
- Nonfunctional information of the service: including information such as QoS (quality of service)

You can easily find the detailed definitions of the related classes and properties of these four types by reading the `profile.owl` document, and they are indeed quite intuitive and straightforward, so we will directly go to the examples.

We will again use the `getMegaPixel` Web service as an example to show the step-by-step creation of the profile document. The first step is to include the service and contact information, as shown in List 12.1.

The first point to note is that we have used the `<!ENTITY>` tag to shorten the namespaces, and the real namespaces are expressed using lines 13 to 22. The

**LIST 12.1**
**The Profile Document for Our `getMegaPixel` Service, Step 1 of 3**

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE uridef[
   <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
   <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
   <!ENTITY owl "http://www.w3.org/2002/07/owl">
   <!ENTITY service
            "http://www.daml.org/services/owl-s/1.1/Service.owl">
   <!ENTITY profile
            "http://www.daml.org/services/owl-s/1.1/Profile.owl">
   <!ENTITY process
            "http://www.daml.org/services/owl-s/1.1/Process.owl">
   <!ENTITY actor
            "http://www.daml.org/services/owl-s/1.1/ActorDefault
             .owl">
   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
   <!ENTITY DEFAULT
            "http://localhost/chap11/GetMegaPixelProfile.owl">
]>

<rdf:RDF
   xmlns:rdf     = "&rdf;#"
   xmlns:rdfs    = "&rdfs;#"
   xmlns:owl     = "&owl;#"
   xmlns:actor   = "&actor;#"
   xmlns:profile = "&profile;#"
   xmlns:xsd     = "&xsd;#"
   xmlns         = "&DEFAULT;#"
   xml:base      = "&DEFAULT;"
>

   <owl:Ontology about="">
      <owl:imports rdf:resource="&service;"/>
      <owl:imports rdf:resource="&profile;"/>
      <owl:imports rdf:resource="&process;"/>
      <owl:imports rdf:resource="&actor;"/>
   </owl:Ontology>

   <profile:profile>

      <profile:serviceName>getMegaPixel</profile:serviceName>
      <profile:textDescription>
         text description of the service for human read
      </profile:textDescription>

      <profile:contactInformation>
         <actor:Actor rdf:ID="getMegaPixel_provider">
```

```
36:           <actor:name>name of the service representative
              </actor:name>
37:           <actor:title>person's title</actor:title>
38:           <actor:phone>(area)+phoneNumber</actor:phone>
39:           <actor:fax>(area)+faxNumber</actor:fax>
40:           <actor:email>service@Camera.com</actor:email>
41:           <actor:physicalAddress>
42:               some street address
43:               State, zip,USA
44:           </actor:physicalAddress>
45:           <actor:webURL>http://someURL</actor:webURL>
46:         </actor:Actor>
47:       </profile:contactInformation>

48        <!-- more to come ... --!>
```

`ActorDefault.owl` ontology on line 9 was originally part of the `profile.owl` ontology, and it has been taken out for readability. Also note that line 21 tells us the name of this profile document and its physical location; i.e., `getMegaPixelProfile.owl` is the name of the profile document and http://localhost/chap11/GetMegaPixelProfile.owl is the physical location of the document. Lines 23 to 28 use the `<owl:ontology>` class to import all the necessary ontologies for this document. The interesting part starts from line 29, which creates a `profile` instance to describe the service. For now, you do not see the closing tag (`</profile:profile>`), because we are not done with this document.

`profile:serviceName` (line 30) is the first property added to this `profile` instance, and the second is `profile:textDescription` (lines 31 to 33). The third property we included is `profile:contactInformation` (lines 34 to 47). All these are quite easy to follow, so we are not going to get into the details. Note that so far all these properties are mainly for human readers; they are not designed to be machine-readable.

The function description component (the second type of the information, as mentioned previously) of the profile is, however, designed to be machine-readable. It is represented by the IOPE model; i.e., this component is described by specifying the relevant input, output, precondition, and effects. Two transformations are described by using this IOPE model: the information transformation is represented by the switch from input to output, and the state transformation is represented by the change from precondition to effects.

It is important to realize the `profile.owl` ontology does not offer classes for modeling IOPE. A profile instance (such as the one we are creating here) will be able to define IOPE using the terms defined by the `process.owl` ontology. List 12.2 is the part of the `profile.owl` ontology related to the definitions of IOPE.

**LIST 12.2**
**IOPE Terms Defined in the `process.owl` Ontology**

```
<owl:ObjectProperty rdf:ID="hasParameter">
  <rdfs:domain rdf:resource="#Profile"/>
  <rdfs:range rdf:resource="&process;#Parameter"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasInput">
  <rdfs:subPropertyOf rdf:resource="#hasParameter"/>
  <rdfs:range rdf:resource="&process;#Input"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasOutput">
  <rdfs:subPropertyOf rdf:resource="#hasParameter"/>
  <rdfs:range rdf:resource="&process;#Output"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasPrecondition">
  <rdfs:domain rdf:resource="#Profile"/>
  <rdfs:range rdf:resource="&expr;#Condition"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasResult">
  <rdfs:domain rdf:resource="#Profile"/>
  <rdfs:range rdf:resource="&process;#Result"/>
</owl:ObjectProperty>
```

Do this exercise: read the `process.owl` ontology to understand the relevant definitions, including `process:hasParameter`, `process:parameter`, `process:input`, and `process:output`, and you should be able come up with the relationship between `profile.owl` and `process.owl`. This relationship is summarized in Figure 12.2.

Clearly, `profile:hasInput` and `profile:hasOutput` properties simply provide pointers to the input and output instances created by using the terms defined in `process.owl`. Let us add the input and output information for the `getMegaPixel` service; the result is shown in List 12.3.

Note that the input and output descriptions are added on lines 43 and 44; our Web service is quite simple so we need only two lines. However, as we said earlier, both the input and output point to some instances created in the `process` file; in our case, this document is called `getMegaPixelProcess.owl`. We will see the details of these instances in the next section when we talk about `process` ontology. Also, as we have made references to some instances created in other documents, we have to explicitly declare the namespace and import the document as well (lines 10, 20, 21, and 31).

What about `profile:hasPrecondition` and `profile:hasResult`? These properties are exactly like the `profile:hasInput` and `profile:hasOutput` properties, and they are simply pointers to the instances created by using the relevant terms defined in `process.owl`. Based on the understanding of `profile:hasInput` and `profile:hasOutput`, it will be fairly easy to understand the precondition and result property. I will leave this as an exercise for you. For our simple Web service example, we do not have any precondition and result, so let us move on.

The other information we can add to the `profile` document is the classification and other nonfunctional information of the given service. Let us use the

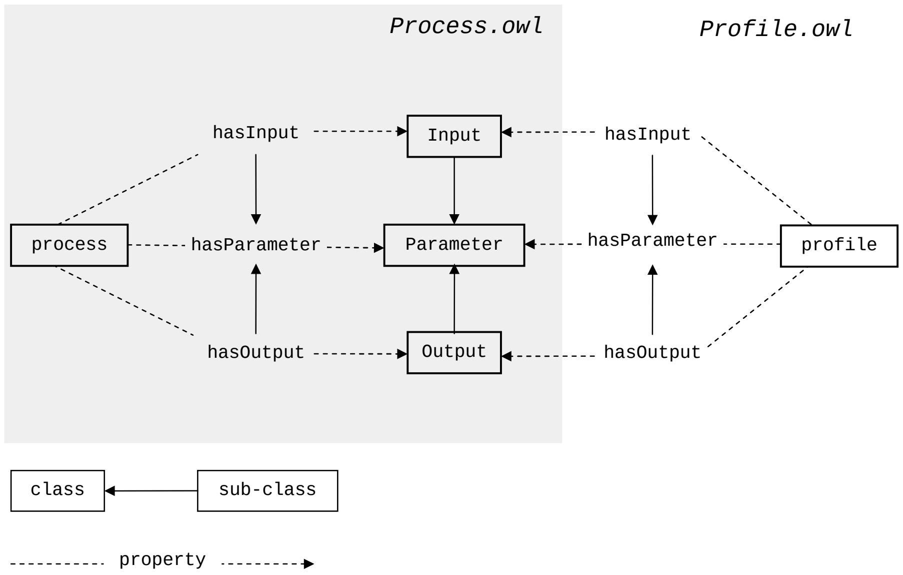

**FIGURE 12.2** Describing IOPE in profile document using terms defined in `process.owl`.

## LIST 12.3
## The Profile Document for Our `getMegaPixel` Service, Step 2 of 3

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE uridef[
   <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
   <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
   <!ENTITY owl "http://www.w3.org/2002/07/owl">
   <!ENTITY service
            "http://www.daml.org/services/owl-s/1.1/Service.owl">
   <!ENTITY profile
            "http://www.daml.org/services/owl-s/1.1/Profile.owl">
   <!ENTITY process
            "http://www.daml.org/services/owl-s/1.1/Process.owl">
   <!ENTITY actor
            "http://www.daml.org/services/owl-s/1.1/ActorDefault
             .owl">
   <!ENTITY getMegaPixelProcess
            "http://localhost/chap11/GetMegaPixelProcess.owl">
   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
   <!ENTITY DEFAULT
            "http://localhost/chap11/GetMegaPixelProfile.owl">
]>

<rdf:RDF
   xmlns:rdf      = "&rdf;#"
```

```
16:    xmlns:rdfs    = "&rdfs;#"
17:    xmlns:owl     = "&owl;#"
18:    xmlns:actor   = "&actor;#"
19:    xmlns:profile = "&profile;#"
20:    xmlns:process = "&process;#"
21:    xmlns:getMegaPixelProcess = "&getMegaPixelProcess;#"
22:    xmlns:xsd     = "&xsd;#"
23:    xmlns         = "&DEFAULT;#"
24:    xml:base      = "&DEFAULT;"
25: >

26:    <owl:Ontology about="">
27:       <owl:imports rdf:resource="&service;"/>
28:       <owl:imports rdf:resource="&profile;"/>
29:       <owl:imports rdf:resource="&process;"/>
30:       <owl:imports rdf:resource="&actor;"/>
31:       <owl:imports rdf:resource="&getMegaPixelProcess;"/>
32:    </owl:Ontology>

33:    <profile:profile>

34:       <profile:serviceName>getMegaPixel</profile:serviceName>
35:       <profile:textDescription>
             <!-- nothing changed here --!>
37:       </profile:textDescription>

38:       <profile:contactInformation>
39:          <actor:Actor rdf:ID="getMegaPixel_provider">
                <!-- nothing changed here --!>
40:          </actor:Actor>
41:       </profile:contactInformation>

42:       <!-- IOPE --!>
43:       <profile:hasInput rdf:resource="&getMegaPixelProcess;
                   #model"/>
44:       <profile:hasOutput rdf:resource="&getMegaPixelProcess;
                   #pixel"/>
45:       <!-- end of IOPE -->

46:       <!-- more to come ... --!>
```

`profile:serviceCategory` property as an example to show how this is done; see List 12.4. Again, the changed part in the document is highlighted by the bold lines. Note that we have closed the `profile` instance at line 59. We have reached a fairly complete description of the `getMegaPixel` Web service.

There is other information that can be added to this profile instance, and we have not covered all these other terms. As usual, you can always understand these terms by reading `profile.owl`. Again, remember it is not a new language, but an upper ontology written in OWL.

**LIST 12.4**
**The Profile Document for Our `getMegaPixel` Service, Step 3 of 3**

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE uridef[
   <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
   <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
   <!ENTITY owl "http://www.w3.org/2002/07/owl">
   <!ENTITY service
            "http://www.daml.org/services/owl-s/1.1/Service.owl">
   <!ENTITY profile
            "http://www.daml.org/services/owl-s/1.1/Profile.owl">
   <!ENTITY process
            "http://www.daml.org/services/owl-s/1.1/Process.owl">
   <!ENTITY actor
            "http://www.daml.org/services/owl-s/1.1/ActorDefault
            .owl">
   <!ENTITY categorization
"http://www.daml.org/services/owl-s/1.1/ProfileAddtionalParameters
.owl">
   <!ENTITY getMegaPixelProcess
            "http://localhost/chap11/GetMegaPixelProcess.owl">
   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
   <!ENTITY DEFAULT
            "http://localhost/chap11/GetMegaPixelProfile.owl">
]>

<rdf:RDF
   xmlns:rdf     = "&rdf;#"
   xmlns:rdfs    = "&rdfs;#"
   xmlns:owl     = "&owl;#"
   xmlns:actor   = "&actor;#"
   xmlns:profile = "&profile;#"
   xmlns:process = "&process;#"
   xmlns:categorization = "&categorization;#"
   xmlns:getMegaPixelProcess = "&getMegaPixelProcess;#"
   xmlns:xsd     = "&xsd;#"
   xmlns         = "&DEFAULT;#"
   xml:base      = "&DEFAULT;"
>

   <owl:Ontology about="">
      <owl:imports rdf:resource="&service;"/>
      <owl:imports rdf:resource="&profile;"/>
      <owl:imports rdf:resource="&process;"/>
      <owl:imports rdf:resource="&actor;"/>
      <owl:imports rdf:resource="&categorization;"/>
      <owl:imports rdf:resource="&getMegaPixelProcess;"/>
   </owl:Ontology>
   <profile:profile>

      <profile:serviceName>getMegaPixel</profile:serviceName>
      <profile:textDescription>
         <!-- nothing changed here --!>
      </profile:textDescription>

      <profile:contactInformation>
         <actor:Actor rdf:ID="getMegaPixel_provider">
            <!-- nothing changed here --!>
         </actor:Actor>
      </profile:contactInformation>

      <!-- IOPE --!>
      <profile:hasInput rdf:resource="&getMegaPixelProcess;
               #model"/>
      <profile:hasOutput rdf:resource="&getMegaPixelProcess;
               #pixel"/>
      <!-- end of IOPE --!>

      <!-- service category --!>
      <profile:serviceCategory>
         <categorization:ServiceCategory rdf:ID="UNSPSC-OnLine">
            <categorization:taxonomy>
               http://www.unspsc.org
            </categorization:taxonomy>
            <categorization:value>
               On-Line Information Services
            </categorization:value>
            <categorization:code>514191</categorization:code>
         </categorization:ServiceCategory>
      </profile:serviceCategory>
      <!-- end of the service category --!>

   </profile:profile>

</rdf:RDF>
```

### 12.3.2 OWL-S Process Ontology

The bad news is that the OWL-S `process` upper ontology is very complex; discussing it in detail would require many more pages. The good news is that our goal in this last part of the book is to see how the added semantics can help the automatic discovery of the Web services, and the `process` model does not have to play a significant role in this discovery process. In fact, the `profile` of a service provides a detailed description of the service mainly to facilitate its discovery. Once the service has been selected, the `profile` is useless; rather, the client will use the `process` document to control the interaction with the service.

Although the more detailed IOPE information expressed in the process document may provide more accuracy when a soft agent is searching for the required service, most popular matchmaking engines (discussed in later chapters) just make use of the `profile` documents without digging into the `process` documents. The real power of the process documents is to help automatic composite and monitoring of the services, which is beyond the scope of this book.

Therefore, we are not going to present a systematic description of the OWL-S process ontology. However, recall that `profile:hasInput` and `profile:hasOutput` properties simply point to the input and output instances created by using the terms defined in `process.owl`; we will in this section make this piece complete by taking a look at the creation of these input and output instances.

First, List 12.5 shows the definition of `hasParameter`, and its subproperties `hasInput` and `hasOutput`.

**LIST 12.5**
**Definitions of `hasParameter`, `hasInput`, and `hasOutput`**

```
<owl:ObjectProperty rdf:ID="hasParameter">
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Parameter"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasInput">
  <rdfs:subPropertyOf rdf:resource="#hasParameter"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Input"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasOutput">
  <rdfs:subPropertyOf rdf:resource="#hasParameter"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Output"/>
</owl:ObjectProperty>
```

This specifies the following rules:

- Property `hasInput` is a subproperty of `hasParameter`; it is used to describe a Process instance, and its value has to be an instance of `Input` class.
- Property `hasOutput` is a subproperty of `hasParameter`; it is used to describe a Process instance, and its value has to be an instance of `Output` class.

The definitions of `Input` and `Output` classes (and their related classes) are shown in List 12.6.

These definitions specify the rules for `Input` and `Output` classes:

- `Input` and `Output` classes are subclasses of `Parameter`, and each instance of `Input` and `Output` class must have a `parameterType` value specified.

**LIST 12.6**
**Definitions of `Input`, `Output`, and Related Classes**

```
<owl:Class rdf:ID="Input">
  <rdfs:subClassOf rdf:resource="#Parameter"/>
</owl:Class>

<owl:Class rdf:ID="Output">
  <rdfs:subClassOf rdf:resource="#Parameter"/>
</owl:Class>

<owl:Class rdf:ID="Parameter">
 <rdfs:subClassOf>
   <owl:Restriction>
     <owl:onProperty rdf:resource="#parameterType" />
     <owl:minCardinality
          rdf:datatype="&xsd;#nonNegativeInteger">1</owl:minCardinality>
   </owl:Restriction>
 </rdfs:subClassOf>
</owl:Class>

<owl:DatatypeProperty rdf:ID="parameterType">
  <rdfs:domain rdf:resource="#Parameter"/>
  <rdfs:range rdf:resource="&xsd;#anyURI"/>
</owl:DatatypeProperty>
```

At this point, these definitions have given us enough information to create the `input` and `output` instances in the process document. For our simple example, these two instances in the `process` document should look like the ones in List 12.7.

**LIST 12.7**
**Creating the `Input` and `Output` Instance for Our Service**

```
<process:Input rdf:ID="model">
  <process:parameterType rdf:datatype="&xsd;#anyURI">
     http://www.yuchen.net/photography/Camera.owl#model
  </process:parameterType>
</process:Input>

<process:Output rdf:ID="pixel">
  <process:parameterType rdf:datatype="&xsd;#anyURI">
     http://www.yuchen.net/photography/Camera.owl#pixel
  </process:parameterType>
</process:Input>
```

Review the `profile` document we have created (List 12.4); note that lines 45 and 46 are the `input` and `output` pointers:

```
<profile:hasInput rdf:resource="&getMegaPixelProcess;#model"/>
<profile:hasOutput rdf:resource="&getMegaPixelProcess;#pixel"/>
```

The reference relationship is built by using the same instance ID; in this case, `model` and `pixel` are used as IDs to point to the instances in the `process` document (List 12.7, lines 1 to 10).

Up to this point, we have completed the task of using the OWL-S upper ontologies to describe a given service. Before we move on to the next section, there is one more change we need to make to our `profile` document in List 12.4. If we are interested mainly in the automatic discovery of the Web services, we might not create the `process` document at all. In this case, we have to create the `input` and `output` instances in the `profile` document instead of the `process` document. List 12.8 shows how this is done.

## LIST 12.8
**The Profile Document for Our `getMegaPixel` Service, with `input` and `output` Instances**

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<!DOCTYPE uridef[
   <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
   <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
   <!ENTITY owl "http://www.w3.org/2002/07/owl">
   <!ENTITY service
            "http://www.daml.org/services/owl-s/1.1/Service.owl">
   <!ENTITY profile
            "http://www.daml.org/services/owl-
s/1.1/Profile.owl">
   <!ENTITY process
            "http://www.daml.org/services/owl-s/1.1/Process.owl">
   <!ENTITY actor
            "http://www.daml.org/services/owl-s/1.1/ActorDefault
            .owl">
   <!ENTITY categorization
  "http://www.daml.org/services/owl-s/1.1/ProfileAddtionalParameters
  .owl">
   <!ENTITY getMegaPixelProcess
            "http://localhost/chap11/GetMegaPixelProcess.owl">
   <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
   <!ENTITY DEFAULT
            "http://localhost/chap11/GetMegaPixelProfile.owl">
 ]>

<rdf:RDF
   xmlns:rdf     = "&rdf;#"
   xmlns:rdfs    = "&rdfs;#"
   xmlns:owl     = "&owl;#"
   xmlns:actor   = "&actor;#"
   xmlns:profile = "&profile;#"
```

```
    xmlns:process = "&process;#"
    xmlns:categorization = "&categorization;#"
    xmlns:getMegaPixelProcess = "&getMegaPixelProcess;#"
    xmlns:xsd     = "&xsd;#"
    xmlns         = "&DEFAULT;#"
    xml:base      = "&DEFAULT;"
>

    <owl:Ontology about="">
       <owl:imports rdf:resource="&service;"/>
       <owl:imports rdf:resource="&profile;"/>
       <owl:imports rdf:resource="&process;"/>
       <owl:imports rdf:resource="&actor;"/>
       <owl:imports rdf:resource="&categorization;"/>
       <owl:imports rdf:resource="&getMegaPixelProcess;"/>
    </owl:Ontology>

    <profile:profile>

       <profile:serviceName>getMegaPixel</profile:serviceName>
       <profile:textDescription>
          <!-- nothing changed here --!>
       </profile:textDescription>

       <profile:contactInformation>
          <actor:Actor rdf:ID="getMegaPixel_provider">
             <!-- nothing changed here --!>
          </actor:Actor>
       </profile:contactInformation>

       <!-- IOPE --!>
       <profile:hasInput>
          <process:Input rdf:ID="model">
             <process:parameterType>
                http://www.yuchen.net/photography/Camera.owl#model
             </process:parameterType>
          </process:Input>
       </profile:hasInput>
       <profile:hasOutput>
          <process:Output rdf:ID="pixel">
             <process:parameterType>
                http://www.yuchen.net/photography/Camera.owl#pixel
             </process:parameterType>
          </process:Output>
       </profile:hasOutput>
       <!-- end of IOPE --!>

       <!-- service category --!>
       <profile:serviceCategory>
          <categorization:ServiceCategory rdf:ID="UNSPSC-OnLine">
```

```
            <categorization:taxonomy>
               http://www.unspsc.org
            </categorization:taxonomy>
            <categorization:value>
               On-Line Information Services
            </categorization:value>
            <categorization:code>514191</categorization:code>
         </categorization:ServiceCategory>
      </profile:serviceCategory>
      <!-- end of the service category --!>

   </profile:profile>

</rdf:RDF>
```

## 12.3.3 OWL-S `Grounding` Ontology

It seems to be true that the more we learn, the more questions we have. For example, after learning how to describe a Web service using the OWL-S upper ontology, the following questions may occur to us:

- The OWL-S upper ontology can be used to describe a Web service, and WSDL is also used to describe a Web service. What is the relationship between them?
- Using OWL-S, how do we specify the details of a Web service, for instance, the particular message formats, protocols, and network address by which a Web service can be instantiated?

These questions are answered by the OWL-S `grounding` ontology. For the purpose of this book, here is a very short answer:

> The OWL-S grounding ontology reuses WSDL to describe the detailed information needed to access the service. Therefore, OWL-S and WSDL are complementary; none can replace the other and provide a complete picture of a given Web service.

In other words, because we are mainly interested in the automatic discovery of Web services by using Semantic Web technology, we can be rest assured that all has been taken care of. As long as we can automatically discover the Web service, the grounding document together with the WSDL document has provided enough information for a soft agent to automatically invoke the service.

Let us now discuss more details. We will first take a look at the connection between the OWL-S grounding ontology and the WSDL document. We will then discuss some main terms defined in the OWL-S `grounding` ontology and, finally, we will construct a grounding document using the `getMegaPixel` service as an example. You can skip this part if you are not interested.

The OWL-S upper ontology has defined a set of classes and properties that can be used to describe a given Web service. It has three main subontologies. The first two subontologies, namely, the `profile` ontology and the `process` ontology, are all considered to be abstract specifications; they do not specify any information needed to invoke the service. The last subontology, namely the `grounding` ontology, is therefore the last chance to provide some concrete details that a soft agent can use to invoke the service automatically.

On the other hand, WSDL, developed independently of OWL-S, provides a well-developed vehicle to specify the needed details to invoke a service. It has also been widely accepted in the real world of Web services; numerous tools supporting WSDL have been developed and adopted by developers. Therefore, reusing WSDL is the way to go. In fact, the OWL-S `grounding` ontology is quite straightforward; it simply defines conventions for using WSDL to ground OWL-S services.

Let us take a look at some of the main classes and properties defined in the OWL-S `grounding` ontology. The basic idea when implementing the grounding is to create an instance of the OWL-S grounding class, and add instances of properties/class described in Table 12.2 to express the relationships between the relevant OWL-S constructs and the related WSDL constructs.

**TABLE 12.2**
**Properties and Classes Defined in `Grounding.owl`**

| Properties/Classes | Meaning |
|---|---|
| `wsdlReference` | Property. A URI indicating the version of WSDL. The current OWL-S `grounding` ontology was created to work with WSDL version 1.1 |
| `wsdlDocument` | Property. A URI indicating the WSDL document to which this `grounding` refers. This links the grounding instance to the corresponding WSDL document |
| `wsdlOperation` | Property. This links the OWL-S `process` instance to a WSDL `operation`; this operation is uniquely identified by a `wsdlOperationRef` instance |
| `wsdlOperationRef` | Class. This class provides a unique specification of a WSDL operation. WSDL 1.1 does not have a way to uniquely identify an operation with a single URI |
| `wsdlInputMessage` | Property. A URI for the WSDL `input` message element corresponding to the `inputs` of the process |
| `wsdlInputMessageMap` | Class. This class indicates how to derive a WSDL `message part` from (one or more) `inputs` of a process. In other words, an instance of this class maps `inputs` of a `process` to the message part of the WSDL document. It must have one `wsdlMessagePart`, and either one `owlsParamter` or one `xlsTransformation`. If there is a direct correspondence between a particular OWL-S input and the `wsdlMessagePart`, use `owlsParameter` to show that; otherwise, use `xslTransformation` to give a transformation from OWL-S inputs to the `wsdlMessagePart` |
| `wsdlMessagePart` | Property. This property represents a URI for a WSDL `message part` element |
| `owlsParameter` | Property. This property represents an `input` or `output` property of a process |
| `wsdlOuputMessage` `wsdlOutputMessageMap` | Similar to the `wsdlInputMessage` and `wsdlInputMessageMap`, but for `outputs` |

Table 12.2 shows the main terms we can use when mapping OWL-S `inputs` and `outputs` to the WSDL constructs. Similarly, the WSDL document has to be extended by including constructs that relate it back to the OWL-S code.

Using `getMegaPixel` service as an example, List 12.9 shows the OWL-S `grounding` instance and the WSDL document. The cross-reference between the two documents is shown by the bold lines.

## LIST 12.9
## OWL-S Grounding Instance and Cross-References with the WSDL Document

```
The OWL-S grounding document:

<!DOCTYPE uridef[
  <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns">
  <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema">
  <!ENTITY owl "http://www.w3.org/2002/07/owl">
  <!ENTITY xsd "http://www.w3.org/2001/XMLSchema">
  <!ENTITY grounding
           "http://www.daml.org/services/owl-
s/1.1/Grounding.owl">
  <!ENTITY process
           "http://www.daml.org/services/owl-s/1.1/Process.owl">
  <!ENTITY getPixelProfile
           "http://localhost/chap11/getMegaPixelProfile.owl">
  <!ENTITY DEFAULT
           "http://localhost/chap11/getMegaPixelGrounding.owl">
]>

<rdf:RDF
   xmlns:rdf="&rdf;#"
   xmlns:rdfs="&rdfs;#"
   xmlns:owl="&owl;#"
   xmlns:xsd="&xsd;#"
   xmlns:process="&process;#"
   xmlns:grounding="&process;#"
   xmlns:getPixelProfile="&getPixelProfile;#"
   xmlns="&DEFAULT;#">

<!-- OWL-S Grounding Instance -->

<grounding:WsdlGrounding rdf:ID="allGeMegaPixelGrounding">
  <grounding:hasAtomicProcessGrounding
                        rdf:resource="#getMegaPixelGrounding"/>
</grounding:WsdlGrounding>

<grounding:WsdlAtomicProcessGrounding rdf:ID="getMegaPixel
Grounding">
  <grounding:owlsProcess rdf:resource="#getMegaPixel"/>
  <grounding:wsdlOperation>
```

```
    <grounding:WsdlOperationRef>
      <grounding:portType>
        <xsd:uriReference
rdf:value="http://localhost/chap11/getMegaPixel.wsdl#getMegaPixel
          PortType"/>
      </grounding:portType>
      <grounding:operation>
        <xsd:uriReference
rdf:value="http://localhost/chap11/getMegaPixel.wsdl#getMegaPixel"/>
      </grounding:operation>
    </grounding:WsdlOperationRef>
  </grounding:wsdlOperation>

  <grounding:wsdlInputMessage
rdf:resource="http://localhost/chap11/getMegaPixel.wsdl#getMegaPixel
Input"/>
  <grounding:wsdlInput>
    <grounding:wsdlInputMessageMap>
      <grounding:owlsParameter rdf:resource="&getPixelProfile;
      #model"/>
      <grounding:wsdlMessagePart>
        <xsd:uriReference
          rdf:value="http://localhost/chap11/getMegaPixel
          .wsdl#model">
      </grounding:wsdlMessagePart>
    </grounding:wsdlInputMessageMap>
  </grounding:wsdlInput>

  <grounding:wsdlOutputMessage
rdf:resource="http://localhost/chap11/ getMegaPixel.wsdl#getMegaPixel
Output"/>
  <grounding:wsdlOutput>
    <grounding:wsdlOutputMessageMap>
      <grounding:owlsParameter rdf:resource="&getPixelProfile;
      #pixel"/>
      <grounding:wsdlMessagePart>
        <xsd:uriReference
          rdf:value="http://localhost/chap11/getMegaPixel.wsdl
          #pixel">
      </grounding:wsdlMessagePart>
    </grounding:wsdlOutputMessageMap>
  </grounding:wsdlOutput>

  <grounding:wsdlVersion
   rdf:resource="http://www.w3.org/TR/2001/NOTE-wsdl-20010315"/>

  <grounding:wsdlDocument>
     http://localhost/chap11/getMegaPixel.wsdl
  </grounding:wsdlDocument>

</grounding:WsdlGrounding>
```

**WSDL document:**

```
<?xml version="1.0" encoding="utf-8"?>
<definitions name="getMegaPixelDef"
  targetNamespace="http://localhost/chap11/getMegaPixel.wsdl"
  xmlns:tns="http://localhost/chap11/getMegaPixel.wsdl"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:owls-wsdl="http://www.daml.org/services/owl-s/wsdl/"
  xmlns:getPixelGrounding
             ="http://localhost/chap11/getMegaPixelGrounding.owl"
  xmlns:getPixelProfile
             ="http://localhost/chap11/getMegaPixelprofile.owl"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

<!-- no need to have the <types> definition any more --!>

 <message name="getMegaPixelInput">
  <part name="model" owls-wsdl:owl-s-parameter="getPixelProfile
  :model"/>
 </message>
 <message name="getMegaPixelOutput">
  <part name="pixel" owls-wsdl:owl-s-parameter="getPixelProfile
  :pixel"/>
 </message>

 <portType name="getMegaPixelPortType">
   <operation name="getMegaPixel"
        owls-wsdl:owl-s-process="getPixelGrounding:getMegaPixel">
     <input message="tns:getMegaPixelInput" />
     <output message="tns:getMegaPixelOutput" />
   </operation>
 </portType>

 <binding name="getMegaPixelBinding" type="tns:getMegaPixelPort
 Type">
   <soap:binding
   transport="http://schemas.xmlsoap.org/soap/http" style=
   "document"/>
   <operation name="getMegaPixel">
     <soap:operation
  soapAction="http://localhost/chap11/getMegaPixel.wsdl#getMega
             Pixel"
  style="document" />
     <input>
       <soap:body parts="model" use="encoded"
            namespace="http://localhost/chap11/getMegaPixel.wsdl"
            encodingStyle="http://www.daml.org/2001/03/"/>
     </input>
     <output>
```

```
30:         <soap:body parts="pixel" use="encoded"
                 namespace="http://localhost/chap11/getMegaPixel.wsdl"
                   encodingStyle="http://www.daml.org/2001/03/"/>
31:       </output>
32:     </operation>
33:   </binding>

34:   <service name="getMegaPixelService">
35:     <port name="getMegaPixelPort" binding="tns:getMegaPixel
        Binding">
36:       <soap:address
          location="http://localhost/chap11/GetMegaPixelWS/Service1
                   .asmx"/>
37:     </port>
38:   </service>
39: </definitions>
```

Let us look at these two documents briefly to see how the connection is built. First of all, an instance of `grounding` is created using lines 20 and 21 in the OWL-S `grounding` document, and line 28 indicates that this `grounding` is for the `portType`, which has the following URI:

```
http://localhost/chap11/getMegaPixel.wsdl#getMegaPixel-
  PortType
```

This uniquely identifies the `portType` defined in the WSDL document on line 17. Further, the OWL-S grounding document continues to specify the operation that is being grounded (line 31):

```
http://localhost/chap11/getMegaPixel.wsdl#getMegaPixel
```

This is necessary because in WSDL documents, a single `portType` can contain several `operations`. This URI maps to line 18 in the WSDL document. Now, the initial connection has been built: both parties know exactly which operation is under the grounding process.

Line 35 in OWL-S `grounding` document starts the grounding of the `input` parameter by specifying the following URI:

```
http://localhost/chap11/getMegaPixel.wsdl#getMegaPixelInput
```

which uniquely identifies the message on line 11 in the WSDL document. Then, lines 36 to 42 in the OWL-S `grounding` document together with line 12 in the WSDL document indicate the following grounding: the `input` parameter

```
http://localhost/chap11/getMegaPixelProfile.owl#model
```

maps to the `model` part contained in the message expressed at line 11 of the WSDL document. Similarly, lines 44 to 52 from the OWL-S grounding document will map the `output` parameter to the `pixel` part expressed at line 15 in the WSDL document.

Up to this point, the grounding is finished for this simple example. Note that the final exact data type and format is retrieved by the understanding of the `range` definition of the `parameterType` property.

For example, by reading these two grounding documents, an automatic tool will understand that the `part` model in the WSDL document is mapped to the `model` instance identified by `getPixelProfile:model`. Now, by looking at the definition of `model` instance contained in the OWL-S document (List 12.8, lines 45 to 46), it will understand that the `input` parameter model has a user-defined type identified by the following URI:

```
http://www.yuchen.net/photography/Camera.owl#model
```

Following this URI and its definition in the ontology file `Camera.owl`, the automatic tool is finally able to reach a familiar data type such as `xsd:string` and, thus, the data type issue is solved.

In this section we briefly discuss the grounding issue involved in describing a Web service using OWL-S upper ontologies. Again, for our purpose, you do not have to understand this; just remember the following:

> The OWL-S grounding document, together with the WSDL document, will give any tool or agent enough information to actually access the service.

Take a look at line 55 in the OWL-S `grounding` document (List 12.9); you will see a link pointing to the corresponding WSDL document, so you know the connection is there.

## 12.4 VALIDATING YOUR OWL-S DOCUMENTS

It is often useful to validate your OWL-S documents before you publish them on your Web server to describe your Web services; potentially, there will be automatic tools or agents reading these documents and you do not want to confuse these agents by having syntax or other errors in your documents.

MindSwap (Maryland Information and Network Dynamics Lab Semantic Web Agents Project) [50] provides several OWL-S tools, including an OWL-S validator. You can find all these tools at `http://www.mindswap.org/2004/owl-s/index.shtml`.

## 12.5 WHERE ARE THE SEMANTICS?

Before we bring this chapter to a close, we have one more important question to answer: where are the semantics expressed in the OWL-S upper ontologies?

The answer is that you can find semantics in all the three subontologies. As far as the issue of automatic discovery of the required Web services is concerned, the semantics are mainly added in the `profile` document.

For our simple `getMegaPixel` service, if we were to use the WSDL document to describe this service, we would only know that this service takes an `xsd:string` as input and returns a `xsd:double` as output and, clearly, there are no semantics at all; there might be hundreds of Web services out there that take exactly the same input and return exactly the same output. Therefore, there will be no way to select this particular service correctly and automatically from all the candidates having a similar look and feel.

Now, we have used the OWL-S profile subontology to describe this service. Review List 12.8; lines 44 to 57 have told us the following:

- The input parameter of this service is called `model`, and its semantics are expressed in the `Camera.owl` ontology using either a class or a property named `model`; the exact context of using such a concept is clearly indicated in this ontology.
- The output parameter of this service is called `pixel`, and its semantics are expressed in the `Camera.owl` ontology using either a class or a property named `pixel`; the exact context of using such a concept is clearly indicated in this ontology.

These are exactly the semantics we are looking for. To make matters clearer, the following discussion shows how these semantics will enable us to automatically find this service. If you want to find a service that takes a camera model as input and returns a megapixel value of this particular model, you need to follow these steps:

*Step 1:* Create a service request file to express what you are looking for by using the concepts from the right ontology (in our case, the `Camera.owl` ontology). This is to ensure you are talking about a potential service in the right context.

*Step 2:* Submit this request file to some smart agent that is capable of reading the OWL-S `profile` document of each candidate service and using a matchmaking engine to conduct semantics matching.

*Step 3:* The agent will compare your request document against each profile document it has located and include the matched service as a potential candidate.

If you still have doubts, please read on: you will see all these steps in action in the next few chapters and obtain a better understanding of how the Semantic Web can help us discover desired Web services automatically.