DAML-S (version 0.6) Walk-Through

This document provides a walk-through example of the use of the DAML-S Web Service Ontologies (version 0.6) for describing Web services. DAML-S is written in DAML+OIL (March 2001). This walk-through is not intended as a complete description of DAML-S. For a complete specification of DAML-S, please refer to the DAML-S reference document http://www.daml.org/services/daml-s/2001/10/daml-s.html.

DAML-S comprises several ontologies in the DAML+OIL (March 2001) markup language. Throughout this walk-through, we will refer to the *profile ontology* and the *process ontology*. Both are described in more detail in the technical section of the DAML-S version 0.6 distribution. The DAML+OIL encoding of these ontologies can be found at http://www.daml.org/services/daml-s/2001/10/Profile.daml and http://www.daml.org/services/daml-s/2001/10/Process.daml., respectively.

The Congo Example
Our walk-through utilizes the example of a fictitious book-buying service offered by the Web service provider, Congo Inc. Congo has a suite of programs that they are making accessible on the Web. These programs (self-described by their names) are LocateBook, PutInCart, SignIn, CreateAcct, CreateProfile, LoadProfile, SpecifyDeliveryDetails, FinalizeBuy. We walk through the exercise of Congo Inc., marking up a subset of these Congo web services in the DAML-S ontology. We focus on the composite service, CongoBuy that composes these smaller programs into a program that enables a user to buy books. DAML-S descriptions of the Congo example can be found at http://www.daml.org/services/daml-s/2001/10/CongoProfile.daml (the service profile) and http://www.daml.org/services/daml-s/2001/10/Congo.daml (the process model). We present excerpts of these examples here.

Task-Driven Markup of Web Services
In this walk-through, we take the perspective of the typical Web service provider (Congo Inc.) and consider three automation tasks that a Web service provider might wish to enable with DAML-S version 0.6 markup:

- automatic Web service discovery
- automatic Web service invocation, and
- automatic Web service composition and interoperation.

These automation tasks are described in more detail in the technical overview section of the DAML-S release http://www.daml.org/services/daml-s/2001/10/daml-s.html and in [McIlraith et al., 2001]. The tasks that the Web service provider wishes to automate drives what DAML-S descriptions they may wish to create.

An important distinction when describing a Web service is the difference between the description of the physical program(s) that you are making Web-accessible, and the description you use to describe the service(s) provided by those program. In what follows, we take a bottom-up approach, first providing a description of the Web-accessible programs through a *process model* description, and then subsequently providing a means of advertising the properties and capabilities of that program through a *profile* description.

Describing Your Programs

The first step in creating a semantic Web service is to describe the program that realizes the service. DAML-S provides for a declarative description of the programs in terms of a process model, thus providing the necessary descriptors to automate Web service invocation, and/or Web service composition and interoperation. If the service provider does not wish to do this, then the may skip this section and move direction to the section entitled “Advertising Your Programs”.

To enable automated Web service invocation, a Web service must be able to tell an external user (henceforth referred to as an *agent*) how to actually interact with the Web service -- how to automatically construct an (http) call to execute or invoke a Web service, and what output(s) may be returned from the

service. To compose services, the process model must describe the preconditions necessary for its execution, and the effects. To enable such functionality, DAML-S provides a process ontology. The markup enables the Web service provider to include sufficient information for automating Web service invocation as well as automating Web service composition.

1. Define the Programs as Processes

Congo Inc. provides the CongoBuy book buying Web service for its customers. This service is actually a collection of smaller Congo programs, each Web-accessible and composed together to form the CongoBuy program. Congo Inc. must first describe each of these individual programs, and then describe their composition, which the external world sees as the CongBuy interactive program/service.

1.1 Define the Individual Programs as Processes

The CongoBuy program comprises a number of different programs including LocateBook, PutInCart, etc. It is the process model that provides a declarative description of the properties of the Web-accessible programs we wish to reason about.

The process model conceives each program as either an *atomic process* or as a *composite process*. It additionally allows for the notion of a *simple process*, which we will use later on to provide a simplied view of the composite CongoBuy program. In general the notion of a *simple process* is used to describe a view, abstraction or default instantiation of an atomic or composite service to which it *expands*.

```
<daml:Class rdf:ID="Process">
  <rdfs:comment> The most general class of processes </rdfs:comment>
  <daml:unionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#AtomicProcess"/>
    <daml:Class rdf:about="#SimpleProcess"/>
    <daml:Class rdf:about="#CompositeProcess"/>
  </daml:unionOf>
</daml:Class>
```

A non-decomposable Web-accessible program is described as an atomic process. An atomic process is characterized by its ability to be executed by a single (e.g., http) call, that returns a response. It does not require an extended conversation between the calling program or agent, and the program.

```
<daml:Class rdf:ID="AtomicProcess">
  <daml:subClassOf rdf:resource="#Process"/>
</daml:Class>
```

An example of an atomic process is the `LocateBook` program that takes as input the name of a book and returns a description of the book and its price, if the book is in Congo's catalogue. The simplest way to proclaim LocateBook an atomic process is as follows. We could then go on to describe its inputs (e.g., bookName), outputs (e.g., bookDescription), preconditions and effects separately. For brevity, we will henceforth refer to the inputs, outputs, preconditions and effects as "iope's."

```
<daml:Class rdf:ID="LocateBook">
  <rdfs:subClassOf rdf:resource="&process;#AtomicProcess"/>
</daml:Class>
```

Nevertheless, if we want to put restrictions on the iope's, such as cardinality restrictions, then we may want to define `LocateBook` as follows with some or all of its iope's listed in the class definition.

```
<daml:Class rdf:ID="LocateBook">
  <rdfs:subClassOf rdf:resource="&process;#AtomicProcess"/>
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
```

```
      <daml:onProperty rdf:resource="#bookName"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

Associated with each process is a set of properties. Using a program or function metaphor, a process has parameters to which it is associated. Two types of parameters are the DAML-S properties *input* and (conditional) *output*.

```
<rdf:Property rdf:ID="parameter">
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range  rdf:resource="http://www.daml.org/2001/03/daml+oil#Thing"/>
</rdf:Property>

<rdf:Property rdf:ID="input">
 <rdfs:subPropertyOf rdf:resource="#parameter"/>
</rdf:Property>
```

An example of an input for `LocateBook` might be the name of the book.

```
<rdf:Property rdf:ID="bookName">
  <rdfs:subPropertyOf rdf:resource="&process;#input"/>
  <rdfs:domain rdf:resource="#LocateBook"/>
  <rdfs:range rdf:resource="&xsd;#string"/>
</rdf:Property>
```

Inputs can be mandatory or optional. In contrast, outputs are generally conditional. This is important. For example, when you search for a book in the Congo catalogue, the output may be a detailed description of the book, if Congo carries it, or it may be a “Sorry we don’t carry.” message. Such outputs are characterized as conditional outputs. We define a conditional output class, `ConditionalOutput`, that describes both a condition and the output based on this condition. An unconditional output has a zero cardinality restriction on its condition.

```
<rdf:Property rdf:ID="output">
  <rdfs:domain rdf:resource="#parameter"/>
  <rdfs:range rdf:resource="#ConditionalOutput"/>
</rdf:Property>

<daml:Class rdf:ID="ConditionalOutput">
  <daml:subClassOf
rdf:resource="http://www.daml.org/2001/03/daml+oil#Thing"/>
</daml:Class>

<rdf:Property rdf:ID="coCondition">
   <rdfs:domain rdf:resource="#ConditionalOutput"/>
  <rdfs:range rdf:resource="#Condition"/>
</rdf:Property>

<rdf:Property rdf:ID="coOutput">
  <rdfs:domain rdf:resource="#ConditionalOutput"/>
  <rdfs:range rdf:resource="http://www.daml.org/2001/03/daml+oil#Thing"/>
</rdf:Property>
```

An example of a conditional output is `bookDescription`, which is an output conditional upon the book being in the Congo catalogue. If the book is not in Congo's catalogues, then the output is a message to this effect. We do not illustrate this second output below.

```
<rdf:Property rdf:ID="bookDescription">
  <rdfs:subPropertyOf rdf:resource="&process;#conditionalOutput"/>
  <rdfs:domain rdf:resource="#LocateBook"/>
  <rdfs:range rdf:resource="InCatalogueBookDescription"/>
</rdf:Property>

<daml:Class rdf:ID="InCatalogueBookDescription">
  <rdfs:subClassOf rdf:resource="&process;#ConditionalOutput"/>
  </daml:Class>

<rdf:Property rdf:ID="condInCatalogueBookDescription">
  <rdfs:subPropertyOf rdf:resource="&process;#coCondition"/>
  <rdfs:domain rdf:resource="#InCatalogueBookDescription"/>
  <rdfs:range rdf:resource="#InCatalogueBook"/>
</rdf:Property>

<rdf:Property rdf:ID="outInCatalogueBookDescription">
  <rdfs:subPropertyOf rdf:resource="&process;#coOutput"/>
  <rdfs:domain rdf:resource="#InCatalogueBookDescription"/>
  <rdfs:range rdf:resource="#TextBookDescription"/>
</rdf:Property>

<daml:Class rdf:ID="TextBookDescription">
  <rdfs:subClassOf rdf:resource="&daml;#Thing"/>
  </daml:Class>
```

The designation of inputs and outputs enables the programs/services that we are describing in DAML-S to be used for automated Web service **invocation**. In order to enable the programs/services to be used for automated **composition and interoperation**, we must additionally describe the side-effects of the programs, if any exist. To do so, it is useful to use an action metaphor to conceive services. In this context we can consider services to have the properties *precondition*s and (conditional) *effect*. Preconditions and conditional effects are described analogously to inputs and conditional outputs.

Preconditions specify things that must be true of the world in order for an agent to execute a service. Many Web services that are embodied as programs on the Web have no preconditions except that the input parameters are known. At the level of abstraction we are modeling Web services, there are no physical preconditions to the execution of a piece of software on the Web. In contrast, Web-accessible devices may have many physical preconditions including bandwidth resources or battery power. Preconditions are described in the process ontology as follows. Since they are a property, like `input`, they are described in exactly the same way as inputs above.

```
<rdf:Property rdf:ID="precondition">
<rdfs:domain rdf:resource="#Process"/>
<rdfs:range  rdf:resource="http://www.daml.org/2001/03/daml+oil#Thing"/>
</rdf:Property>
```

Effects, like outputs, are conditional. Conditional effects characterize the physical side-effects, execution of a Web-service have on the world. An example of a conditional effect for the `FinalizeBuy` service might be `Own(bookName)`, when `cleared(creditCardNumber)`. Note that not all services have physical side-effects. In particular, services that are strictly information providing do not. In the process model, conditional effects are described as follows.

```
<rdf:Property rdf:ID="effect">
<rdfs:domain rdf:resource="#Process"/>
<rdfs:range  rdf:resource="#ConditionalEffect"/>
</rdf:Property>

<daml:Class rdf:ID="ConditionalEffect">
  <daml:subClassOf
rdf:resource="http://www.daml.org/2001/03/daml+oil#Thing"/>
</daml:Class>

<rdf:Property rdf:ID="ceCondition">
   <rdfs:domain rdf:resource="#ConditionalEffect"/>
  <rdfs:range rdf:resource="#Condition"/>
</rdf:Property>

<rdf:Property rdf:ID="ceEffect">
  <rdfs:domain rdf:resource="#ConditionalEffect"/>
  <rdfs:range rdf:resource="http://www.daml.org/2001/03/daml+oil#Thing"/>
</rdf:Property>
```

1.2 Define Compositions of Programs as Composite Processes

With a description of each of the atomic programs/processes in hand, and all their iope's, we are now ready to describe compositions of these programs that provide specific services. Here we examine the CongoBuy composite service, that enables a user to buy a book.

In contrast to atomic processes, composite processes are composed of other composite or atomic processes. They are composed through the use of *control constructs*, typical programming language constructs such as *sequence, if-then-else, while, fork*, etc, that dictate the ordering and the conditional execution of processes in the composition. The following is the definition of a composite process in the process model. As you can see, a composite process is composed of other processes.

```
<daml:Class rdf:ID="CompositeProcess">
  <daml:intersectionOf rdf:parseType="daml:collection">
      <daml:Class rdf:about="#Process"/>
      <daml:Restriction daml:minCardinality="1">
         <daml:onProperty rdf:resource="#composedOf"/>
      </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>

<rdf:Property rdf:ID="composedOf">
<rdfs:domain rdf:resource="#CompositeProcess"/>
<rdfs:range rdf:resource="#ControlConstruct"/>
</rdf:Property>
```

We build the DAML-S composite process recursively in a top-down manner. Each CompositeProcess is composedOf a ControlConstruct, which may be a Sequence, Alternative, If-then-else, etc. Each such ConstrolConstruct, in turn, has a "components" property (a list or bag), which specifies the classes of the subcomponents of the ControlConstruct. These classes may themselves be processes or control constructs. Finally we bottom out when the components of a composite process are atomic processes.

In the Congo example, CongoBuy was described in terms of two main steps – locating the book, and then buying the book. While (for exposition) we assume that the locate book is an atomic process (without

components), the buying of a book involves a sequence of subprocesses (other atomic or composite processes) that correspond to specifying a payment method, specifying the details of delivery (address, wrapping type, etc.) and finalizing the buy process. In this walk-through, we show how to describe a simple composition and refer the reader to the Congo encoding for the full encoding of the CongoBuy composite process.

ExpandedCongoBuy is the name we use for the sequence of the atomic process LocateBook, followed by the, as yet undefined composite process CongoBuyBook. As was the case with atomic processes, composite processes have iopes's (inputs, outputs, preconditions and effects) and we can likewise place restrictions on these iope's within our class definition.

```
<daml:Class rdf:ID="ExpandedCongoBuy">
  <rdfs:subClassOf rdf:resource="&process;#CompositeProcess"/>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="&process;#composedOf"/>
      <daml:toClass>
       <daml:Class>
        <daml:intersectionOf rdf:parseType="daml:collection">
          <daml:Class rdf:about="process:Sequence"/>
          <daml:Restriction>
            <daml:onProperty rdf:resource="process:components"/>
            <daml:toClass>
             <daml:Class>
              <daml:listOfInstancesOf rdf:parseType="daml:collection">
                <daml:Class rdf:about="#LocateBook"/>
                <daml:Class rdf:about="#CongoBuyBook"/>
              </daml:listOfInstancesOf>
             </daml:Class>
            </daml:toClass>
          </daml:Restriction>
        </daml:intersectionOf>
       </daml:Class>
      </daml:toClass>
    </daml:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#expCongoBuyBookName"/>
    </daml:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#expCongoBuyCreditCardNumber"/>
    </daml:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#expCongoBuyCreditCardType"/>
    </daml:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#expCongoBuyCreditCardExpirationDate"/>
    </daml:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
```

```
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#expCongoBuyDeliveryAddress"/>
    </daml:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#expCongoBuyPackagingSelection"/>
    </daml:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <daml:Restriction daml:cardinality="1">
      <daml:onProperty rdf:resource="#expCongoBuyDeliveryTypeSelection"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

The DAML-S description of CongoBuyBook can be found at http://www.daml.org/services/daml-s/2001/10/daml-s.html.

1.3 Creating a Simplified View of a Service (Optional)

Although the CongoBuy service is actually a predetermined composition of several of Congo's Web-accessible programs, it is useful to initially view it as a black-box process that expands to the composite process. The value of the black-box process is that it enables the details of the operation of the composite process to be hidden. In our Congo example, we do this by definition a *simple process* called CongoBuy

```
<daml:Class rdf:ID="CongoBuy">
  <rdfs:subClassOf rdf:resource="&process;#SimpleProcess"/>
</daml:Class>
```

We related the simple process to our composite process, ExpandedCongoBuy through the expand property. CongoBuy expands to ExpandedCongoBuy.

```
<daml:Class rdf:about="#CongoBuy">
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="&process;#expand"/>
      <daml:toClass rdf:resource="#ExpandedCongoBuy"/>
    </daml:Restriction>
  </rdfs:subClassOf>
</daml:Class>
```

As was the case with ExpandedCongoBuy, the black-box process, CongoBuy has a variety of properties, that characterize its blackbox view. The iope's for the black-box process are designed to be a useful shorthand. These are generally the computed iope's of the associated composite process, but we do not discuss details of computed iope's in this release. For now, DAML-S leaves this decision up to the Web service provider.

This completes our walk-through of how to describe Congo Inc.'s programs.

2. Advertising the Services Provided by your Programs

DAML-S has been designed to enable automated Web service discovery by providing DAML+OIL descriptions of the properties and capabilities of a Web service, typically used to locate or select a service. These descriptors can be used to populate a registry of services, to provide better indexing and retrieval features for search engines, or they can be used as input to a match-making system (e.g., [Sycara et al., 1999]). Markup for Web service discovery is likely the simplest form of markup a service

provider will wish to provide. It does not depend upon a process model being constructed, however if a process model is constructed, it should be used to populate some of the profile, in order to maintain consistency.

In this section we walk the reader through DAML-S 0.6 profile construction. We continue with our CongoBuy example. The complete DAML-S Congo profile can be found at http://www.daml.org/services/daml-s/2001/10/Congo-profile.daml. The complete profile ontology from which the Congo profile is subclassed can be found at http://www.daml.org/services/daml-s/2001/10/Profile. In the rest of the example we will assume the following XML namespaces. XML entities such as "&concepts;" are also resolved using the list below.

```
      concepts    "http://www.daml.ri.cmu.edu/ont/DAML-S/concepts.daml"
         congo    "http://www.daml.org/services/daml-s/2001/10/Congo.daml"
       country    "http://www.daml.ri.cmu.edu/ont/Country.daml"
          daml    "http://www.daml.org/2001/03/daml+oil"
       profile    "http://www.daml.org/services/daml-s/2001/10/Profile"
           rdf    "http://www.w3.org/1999/02/22-rdf-syntax-ns"
          rdfs    "http://www.w3.org/2000/01/rdf-schema"
       service    "http://www.daml.org/services/daml-s/2001/10/Service"
           xsd    "http://www.w3.org/2000/10/XMLSchema"
```

Unlike our process model which describes services as classes, a service profile is an instance of the class Profile defined in the profile ontology. rdf:ID provides an ID to the instance so it can be referred to by otherontologies.

```
<profile:profile rdf:ID="Profile_Congo_BookBuying_Service">
```

The first set of information that the service profile provides is descriptive information about the service and information about the provider of the service.

isPresentedBy relates the profile to the service it describes, in this case Congo_BookBuying_Service.

```
<service:isPresentedBy>
  <service:Service df:resource = "&congo;#Congo_BookBuying_Service"/>
</service:isPresentedBy>
```

serviceName is an identifier of the service that can be used to refer to the profile.

```
 <profile:serviceName>Congo_BookBuying_Agent</profile:serviceName>
```

textDescription is a human readable description of the service

```
<profile:textDescription>
    This agentified service provides the opportunity to browse a
    book selling site and buy books there
</profile:textDescription>
```

providedBy describes the provider of the service.

```
<profile:providedBy>
    <profile:ServiceProvider rdf:ID="CongoBuy">
       <profile:name>CongoBuy</profile:name>
          <profile:phone>412 268 8780 </profile:phone>
          <profile:fax>412 268 5569 </profile:fax>
          <profile:email>Bravo@Bravoair.com</profile:email>
          <profile:physicalAddress>
             somewhere 2,
             OnWeb,
```

```
                Montana 52321,
                USA
            </profile:physicalAddress>
        <profile:webURL>
                http://www.daml.org/services/daml-s/2001/10/CongoBuy.html
        </profile:webURL>
    </profile:ServiceProvider>
</profile:providedBy>
```

In the next section the profile specifies additional attributes of the service. These are attributes like quality guarantees, service constraints like geographic radius and so on.

geographicRadius specifies whether there is a limitation on the distribution of the service. By restricting the geographicRadius to United States, we specify that the service will not be offered to requesters outside this area. This field is used either by the register during matching, or by the requester to decide whether to use this service.

```
<profile:geographicRadius rdf:resource="&country;#UnitedStates"/>
```

qualityRating specify the quality of the service provided. For now it is just a placeholder. This field is used either by the register during matching to make sure that the quality requested is matched, or by the requester to decide whether to use this service.

```
<profile:qualityRating rdf:resource="&concepts;#qualityRatingGood"/>
```

The next section of the profile is a set of attributes for describing key elements of the process that this profile is a characterization of. The four key elements of the process model that may want to be described are the input parameters (data to be sent to the service), the outputs (data to be returned by the service provider), and potentially perhaps key constraints of service usage, such as the preconditions and conditional (side-)effects of the service. When a process model exists, these attributes of the service are generally derived from the process model, however when there is no process model, they must be manually encoded by the service providers.

The profile uses a somewhat different way to describe inputs and outputs than does the process model, as it is not directly describing the process actions. The property profile:input is used to describe each key input to the corresponding process, by using as values the descriptions of the class profile:ParameterDescription. ParameterDescriptions name the corresponding parameter properties of the process, and their value restrictions. Currently, due to limitations of the DAML language, there is no logical relationship between ParameterDescriptions in the Profile and the actual input, output, etc. parameters of the corresponding process model. Therefore, at least in theory, the two descriptions may be inconsistent. Nevertheless, the intended use of these descriptions is to correctly characterize the key parameters of the process model so that potential clients can properly find offered services that meet their needs.

input refers to inputs to the process model. Each input requires a name and a restriction to what information is requested and a reference to the process model input used. There are no encoded logical constraints between the inputs in the process model and the inputs in the profile model, therefore, at least in theory, the two sets may be totally unrelated. Nevertheless, the iope's of the profile should be consistent with the iope's of the process model, since discovery of a service often leads to its execution.

An input parameter is described by a name, a restriction on its values, and a reference to the input parameter in the profile it represents. The value restriction is used during matching to check whether the inputs that the requester is willing to provide match what the provider needs. The requester uses the inputs to know what additional information it needs to provide to the service to have a successful run.

```
<input>
    <profile:ParameterDescription>
        <profile:parameterName rdf:resource="bookTitle"/>
        <profile:restrictedTo rdf:resource="&xsd;#string"/>
        <profile:refersTo rdf:resource="&congo;#congoBuyBookName"/>
```

```
    </profile:ParameterDescription>
</input>
```

Outputs are represented similarly to inputs. As with the inputs the restriction is used by the web register to specify whether the service provides the outputs that are expected by the requester. The requester uses the outputs to know what additional knowledge it will acquire from the service.

```
<output>
    <profile:ParameterDescription>
       <profile:parameterName rdf:resource="EReceipt"/>
       <profile:restrictedTo rdf:resource="&congoProcess;#EReceipt"/>
       <profile:refersTo rdf:resource="&congo;#congoBuyReceipt"/>
    </profile:ParameterDescription>
</output>
```

Preconditions and effects have a structure similar to the structure of inputs and outputs. The main difference is that instead of a restriction to some class they have a statement which are defined as daml:Thing. Preconditions and effects are used by the registry in a way that is similar to the inputs and outputs. Furthermore, the requester uses preconditions to make sure that indeed it can run the service; while it uses effects to know what will result after the interaction with the service completes.

```
<precondition>
    <profile:ConditionDescription>
       <profile:ConditionName rdf:resource="AcctExists"/>
       <profile:statement rdf:resource="&congoProcess;#AcctExists"/>
       <profile:refersTo
          rdf:resource="&congo;#congoBuyAcctExistsPrecondition"/>
    </profile:ParameterDescription>
</precondition>
```

Finally, close the description of the service.

```
</profile:OfferedService>
```