1 DAML-S (version 0.5) Walk-Through

This document provides a walk-through example of the DAML-S Web Service Markup Language, version 0.5, which is built on top of DAML+OIL (March 2001). This walk-through is not intended as a complete description of the DAML-S language. For a complete specification of DAML-S, please refer to the DAML-S reference document http://www.daml.org/services/daml-s/2001/05/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 process ontology. The process ontology is described in more detail in the technical section of the DAML-S version 0.5 distribution. The DAML+OIL markup for the process ontology can be found at http://www.daml.org/services/daml-s/2001/05/Process.daml.

1.1 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 program (self-described by their names) are LocateBook, PutInCart, SignIn, CreateAcct, CreateProfile, LoadProfile, SpecifyDeliveryDetails, FinalizeBuy. Congo wishes to compose these individual programs into Web services that it offers to its users. We focus here on the Web service of buying a book, CongoBuy. Our walk-through steps through the process of creating DAML-S markup for Congo.

1.2 Task-Driven Markup of Web Services

In this walk-through, we take the perspective of the typical Web service provider and consider three automation tasks that a Web service provider might wish to enable with DAML-S version 0.5 markup:

1. automatic Web service discovery
2. automatic Web service invocation, and
3. automatic Web service composition and interoperation.

These automation tasks are described in more detail in the technical overview section of the DAML-S release https://www.daml.org/services/daml-s/2001/05/daml-s.html and in [McIlraith et al., 2001].

1.3 Web Service Discovery

DAML-S has been designed to enable automated Web service discovery by providing a markup language for encoding the properties and capabilities of a Web service so that those services can be either included in a larger registry, or indexed and retrieved via a search engine or 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. In this section we walk the reader through DAML-S 0.5 markup for automating service discovery. The complete example of DAML- S markup for Congo Web service discovery can be found at https://www.daml.org/services/daml-s/2001/05/Congo-profile1.daml.

¡This still needs to be filled in.¿

1.4 Web Service Invocation

While the markup presented in the previous section enables automated Web service discovery, it does not tell a program (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 enable such functionality, DAML-S provides a process ontology. This process ontology provides markup to describe individual and composite Web-accessible programs as either atomic or composite processes. The markup enables the Web service provider to include sufficient information for automating Web service invocation as well as automating Web service composition. We focus on the subset of the process ontology that enables Web service invocation first, leaving discussion of other aspects of the process ontology to the next section.

1.4.1 Define the Service as a Process

Congo Inc. provides the CongoBuy Web service to its customers. We view the CongoBuy Web service as a Process, i.e., it is a subclass of the class Process in the process ontology.

```
<rdfs:Class rdf:ID="CongoBuy">
  <rdfs:subClassOf rdf:resource="http://www.www.daml.org/services/daml-
s/2001/05/Process#Process"/>
</rdfs:Class>
```

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. Later we will see how to open up or expand this black box to look at the details of the composition.

The black-box process, CongoBuy has a variety of properties. Those relevant for automating Web service invocation include its name, parameter(s), and in particular the service's input(s) and (potentially conditional) output(s). For example, input to the CongoBuy book-buying service includes the name of the book (bookName), the customer's credit card number, and their account number and password. If the service being described is simple in that it is not the composition of other services or programs, then the service inputs are simply the set of inputs that must be provided in the http service invocation. The outputs are the outputs returned from the http service invocation. Note that these outputs may be conditional. For example the output of a book-buying service will be different dependent upon whether the book is in or out of stock.

In contrast, if the service is composed of other services, as is the case with CongoBuy, then the rationale for specification of the inputs, outputs and parameters is more difficult, and the utility of these properties is limited. In the simplest case, the inputs and outputs of the black-box process can be defined to be the composition of all the possible inputs and all the possible (conditional) outputs of the simple services that the black-box process may invoke, taking every possible path through the composition of simple services. Note however that this is not a very exacting specification. In particular, the collection of outputs may be contradictory. For example, in most cases, the output of CongoBuy will be an eReceipt, but in cases where the book is out of stock, the output may be a failure message. The conditions under which inputs and outputs arise are encoded exactly in the expand of this black-box process, and can be retrieved from this process. The inputs, outputs and parameters for the black-box process are designed to be a useful shorthand. Thus, it could be argued that the inputs and outputs should describe the most likely inputs and outputs through the system. However, in some cases, even this is difficult to define. For now, DAML-S leaves this decision up to the Web service provider.

The following is an example of one input to CongoBuy. Note that it is a subproperty of the property input of Process, from the process model.

```
<rdf:Property rdf:ID="bookName">
  <rdfs:subPropertyOf rdf:resource="http://www.www.daml.org/services/daml-
s/2001/05/Process#input"/>
   <rdfs:domain rdf:resource="#CongoBuy"/>
   <rdfs:range rdf:resource="#rdfs:Literal"/>
</rdf:Property>
```

This says that bookName is an input of CongoBuy whose range is restricted to Literal. Some properties my additionally require the definition of new classes over which the properties range. For example:

```
<rdfs:Class rdf:ID="CreditCardTypes">
  <daml:oneOf rdf:parseType="daml:collection">
    <CreditCardType rdf:ID="MasterCard"/>
    <CreditCardType rdf:ID="VISA"/>
    <CreditCardType rdf:ID="AmericanExpress"/>
    <CreditCardType rdf:ID="DiscoverCard"/>
   </daml:oneOf>
</rdfs:Class>

<rdf:Property rdf:ID="creditCardType">
  <rdfs:subPropertyOf rdf:resource="https://www.daml.org/services/daml-
s/2001/05//Process#input"/>
  <rdfs:range rdf:resource="#CreditCardTypes"/>
</rdf:Property>
```

An output of CongoBuy is that and an electronic receipt (eReceipt) is returned by the service. Again this is a subproperty of the property output of Process. In a real book-buying service, this output would likely be conditioned on the book being in stock, or the customer's credit card being valid, but to simplify our example, we assume Congo has an infinite supply of books, and infinite generosity.

```
<rdf:Property rdf:ID="eReceiptOutput">
  <rdfs:subPropertyOf rdf:resource="https://www.daml.org/services/daml-
s/2001/05//Process#output"/>
  <rdfs:range rdf:resource="#EReceipt"/>
</rdf:Property>
```

In addition to input and output properties, each service has parameter properties. A parameter is something that affects the outcome of the process, but which is not an input provided by the invoker of the process. It may be known by the service, or retrieved by the service from elsewhere. For example, the fact that the customer's credit card is valid, is a parameter in our CongoBuy process, and is relevant when considering the use of the CongoBuy, but it is not an input or output of CongoBuy.

```
<rdf:Property rdf:ID="creditCardValidity">
  <rdfs:subPropertyOf
rdf:resource="https://www.daml.org/services/daml-s/2001/05//Process.daml#pa
rameter"/>
  <rdfs:range rdf:resource="#ValidityType"/>
</rdf:Property>

<rdfs:Class rdf:ID="ValidityType">
  <daml:oneOf rdf:parseType="daml:collection">
    <DeliveryType rdf:ID="Valid"/>
    <DeliveryType rdf:ID="Expired"/>
    <DeliveryType rdf:ID="Invalid CC-Number"/>
    <DeliveryType rdf:ID="Invalid CC-Type"/>
    <DeliveryType rdf:ID="Authorization Refused"/>
  </daml:oneOf>
</rdfs:Class>
```

1.4.2 Define the Process as a Composition of Processes

Given the variability in the specification of inputs, outputs and parameters, it is generally insufficient to simply specify a service as a black-box process, if the objective is to automate service invocation. In such cases, we must expand the black-box service to describe its composite processes. To do so with CongoBuy, we must define each of the simple services in CongoBuy, i.e., LocateBook, PutInCart, SignIn, CreateAcct, CreateProfile, LoadProfile, SpecifyDeliveryDetails, and FinalizeBuy as a subclass of the black-box process, CongoBuy. E.g.,

Define the Individual Processes

```
<rdfs:Class rdf:ID="LocateBook">
  <rdfs:subClassOf rdf:resource="#CongoBuy"/>
</rdfs:Class>
```

We must also identify the relevant input, output and parameters to that individual process. There are many ways to do this. E.g.,

```
<rdf:Property rdf:ID="locateBookInput">
 <daml:samePropertyAs rdf:resource="#bookName"/>
</rdf:Property>
```

The PutInCart process alternately illustrates the use of anonymous subclass with restriction to specify bookName as input.

```
<rdfs:Class rdf:ID="PutInCart">
 <rdfs:subClassOf rdf:resource="#CongoBuy"/>
 <rdfs:subClassOf>
  <daml:Restriction>
   <daml:onProperty rdf:resource="https://www.daml.org/services/daml-
s/2001/05//Process#input"/>
   <daml:samePropertyAs rdf:resource="#bookName"/>
  </daml:Restriction>
 </rdfs:subClassOf>
</rdfs:Class>
```

Other example input is included in http://www.www.daml.org/services/daml- s/2001/05/Congo.daml.

Define the Composition of the Individual Processes

The composition of each of our simple services can be defined by using the composition constructs created in the process ontology, i.e., Sequence, Split, Split + Join, Unordered, Condition, If-Then-Else, Repeat-While, Repeat-Until.

We first create an expand class and then construct the overall expand class recursively in a top- down manner.

```
<expand>
<rdfs:Class> rdfs:about ="#CongoBuy"</rdfs:Class>
<rdfs:Class> rdfs:about ="#ExpandedCongoBuy"</rdfs:Class>
</expand>
```

Each process has a property called components (itself a bag of processes). The processes in the bag may be other simple or composite processes. As such, they recursively define the composition of simple processes that defines the black-box process CongoBuy.

The expanded CongoBuy process (ExpandedCongoBuy) is comprised of a sequence of two processes, a simple process that locates a book (LocateBook), and a complex process that buys the book (CongoBuyBook). We define them as follows:

```
<rdfs:Class rdf:ID="ExpandedCongoBuy">
 <daml:subClassOf rdf:resource="https://www.daml.org/services/daml-
s/2001/05//Process.daml#Sequence"/>
  <daml:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="https://www.daml.org/services/daml-
s/2001/05//Process.daml#components"/>
      <daml:toClass>
       <daml:subClassOf>
        <daml:unionOf rdf:parseType="daml:collection">
          <rdfs:Class rdfs:about="#LocateBook"/>
          <rdfs:Class rdfs:about="#CongoBuyBook"/>
        </daml:unionOf>
       </daml:subClassOf>
      </daml:toClass>
    </daml:Restriction>
  </daml:subClassOf>
</rdfs:Class>
```

In the full Congo.daml example, CongoBuyBook is a composite process that is further decomposed, eventually terminating in a composition of simple processes. With this markup we complete our markup to enable automated service invocation.

1.4.3 Automated Service Composition and Interoperation

The DAML-S markup required to automate service composition and interoperation builds directly on the markup for service invocation. Although the markup itself may be minimal, it can be tricky to articulate correctly. In order to automate service composition and in order for services/agents to interoperate, we must also encode the effects a service has upon the world. For example, when a

human being goes to www.congo.com and successfully executes the CongoBuy service, the human knows that they have purchased a book, that their credit card will be debited, and that they will receive a book at the address they provided. Notice that such consequences of Web service execution are not part of the input/output markup we created for automating service invocation. In order to automate Web service composition and interoperation, or even to select an individual service to meet some objective, preconditions and (conditional) effects of Web service execution must be encoded for computer use.

To provide for this, the process ontology defines the properties precondition and effect. As with our markup for automated service invocation, we define preconditions and effects both for the black-box process CongoBuy and for each of the simple processes that define its composition, and as with defining inputs and outputs, it is easiest to define the preconditions and effects for each of the simple processes first, and then to aggregate them into preconditions and effects for CongoBuy. The markup is analogous to the markup for input and (conditional) output, but is with respect to the properties precondition and (conditional) effect, instead. E.g.,

```
<rdf:Property rdf:ID="acctExistsPrecondition">
  <rdfs:subPropertyOf rdf:resource="https://www.daml.org/services/daml-
s/2001/05//Process.daml#precondition"/>
  <rdfs:range rdf:resource="#AcctExists"/>
</rdf:Property>

<rdfs:Class rdf:ID="BuyEffectType">
  <daml:oneOf rdf:parseType="daml:collection">
    <BuyEffectType rdf:ID ="OrderShipped"/>
    <BuyEffectType rdf:ID ="Failure"/>
  </daml:oneOf>
</rdfs:Class>

<rdf:Property rdf:ID="buyEffect">
  <rdfs:subPropertyOf rdf:resource="https://www.daml.org/services/daml-
s/2001/05//Process.daml#effect"/>
  <rdfs:range rdf:resource="#BuyEffectType"/>
</rdf:Property>
```

Again, Congo.daml provides further examples of preconditions and effects.