1 Modeling Services as Processes

To give a detailed perspective on how to interact with a service, it can be viewed as a *process*. Specifically, OWL-S 1.1 defines a subclass of `ServiceModel`, the `ProcessModel`, which draws upon well-established work in a variety of fields, including work in AI on standardizations of planning languages [**?**], work in programming languages and distributed systems [**?**, **?**], emerging standards in process modeling and workflow technology such as the NIST's Process Specification Language (PSL) [**?**] and the Workflow Management Coalition effort (http://www.aiim.org/wfmc), work on modeling verb semantics and event structure [**?**], previous work on action-inspired Web service markup [**?**], work in AI on modeling complex actions [**?**], and work in agent communication languages [**?**, **?**].

It is important to understand that a process is not a program to be executed. It is a specification of the ways a client may interact with a service. An *atomic* process is a description of a service that expects one (possibly complex) message and does one thing in response. A *composite* process is one that maintains some state; each message the client sends advances it through the process.

A process can have two sorts of purpose. First, it can generate and return some new information based on information it is given and the world state. Information production is described by the inputs and outputs of the process. Second, it can produce a change in the world. This transition is described by the preconditions and effects of the process.

A process can have any number of inputs (including zero), representing the information that is, under some conditions, required for the performance of the process. It can have any number of outputs, the information that the process provides returns. There can be any number of preconditions, which must all hold in order for the process to be invoked. Finally, the process can have any number of effects. Outputs and effects can depend on conditions that hold true of the world state at the time the process is performed. (We use the term *perform* instead of *execute* to de-emphasize the traditional picture of a single agent being responsible for the occurrence of the process.)

Before we can go into the details of how processes work, it's necessary to explain how inputs, outputs, conditions, and effects (colloquially known as IOPEs) work, because fitting them into the OWL framework requires bending the rules somewhat.

1.1 Infrastructure: Parameters and Expressions

Inputs and outputs are subclasses of a general class called `Parameter`. It's convenient to identify parameters with what are called *variables* in SWRL, the language for expressing OWL Rules.

```
<owl:Class rdf:about="#Parameter">
  <rdfs:subClassOf rdf:resource="&swrl;#Variable"/>
</owl:Class>
```

Every parameter has a type. This is not the OWL class the parameter belongs to, but some specification of the class that *values* of the parameter belong to.

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

<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>
```

Inputs and outputs are subclasses of parameter:

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

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

Two other subclasses of `Parameter`, `Local` and `ResVar`, are described below.

Modeling variables as global, named individuals, as is done for Owl Rules, can be misleading. RDF has no notion of the "scope" of a variable, because an RDF document is nothing but a pile of triples. A variable is named with a URI, like any other resource, and so has global scope, or, more accurately, no notion of scope at all. In spite of this lack of structure, we often use RDF to encode hierarchical entites such as formulas and control structures. Wrapping variable references inside literals allows us to sneak in and impose our own scoping rules. We discuss the OWL-S rules in section 1.2.

A process cannot be invoked unless its *preconditions* are true. If and when it does get triggered, it has various *effects*. For example, an agent can order 1000 bolts from a web service only if it can get the web service to accept its promise to pay. One effect of placing the order is the transfer of ownership of the bolts from the service to the agent (or the legal person for which it is a proxy).

Preconditions and effects are represented as logical formulas. Getting logical formulas into RDF has not been easy, but it is now reasonably clear how to proceed. There are actually several possible approaches, depending on how close to RDF/OWL one wants to remain. Usually having lots of choices for such a crucial job is a bad idea, but in this case most of the differences are superficial; it is fairly easy to translate between alternative notations.

The key idea underpinning our approach is to treat expressions as literals, either string literals or XML literals. The latter case is used for languages whose standard encoding is in XML, such as SWRL [**?**] or RDF [**?**]. The former case is for other languages such as Kif [**?**] and PDDL [**?**].

```
<owl:Class rdf:ID="Expression">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#expressionLanguage"/>
      <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
                   1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#expressionBody"/>
      <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
                   1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

We annotate expressions with the language they are expressed in:

```
<owl:ObjectProperty rdf:ID="&expr;#expressionLanguage">
  <rdfs:domain rdf:resource="&expr;#Expression"/>
  <rdfs:range rdf:resource="&expr;#LanguageDefinition"/> <!-- a URI -->
</owl:ObjectProperty>
```

The ontology [`http://www.daml.org/services/owl-s/1.1/generic/Expression.owl`] defines `Expressions` and their properties. As an example, we might state that in order to send the number of a certain credit card to a web agent, one must know what its number is:

```
  <Description rdf:about="#process2">
     <hasPrecondition>
        <Expression expressionLanguage="&kif;#Kif"
                    rdf:dataType="&xsd;#string">
            (!agnt:know_val_is
                 (!ecom:credit_card_num ?cc)
                 ?num)
        </Expression>
     </hasPrecondition>
  </Description>
```

(where the notation "`!`*ecom*`:`" for namespaces is taken from Lassila's WILBUR system [**?**]). (The declaration of the variable `?c` is not shown. We'll come back to that point shortly.)

In cases where an XML encoding is used, we would declare an expression to be an XML literal. Here's the same example using DRS as the expression language:

```
  <Description rdf:about="#process2">
     <hasPrecondition>
        <Expression expressionLanguage="&drs;#DRS"
                    rdf:parseType="Literal">
            <drs:Atomic_formula>
               <rdf:predicate rdf:resource="&agnt;#Know_val_is"/>
               <rdf:subject>
                  <drs:Functional_term>
n                     <drs:function rdf:resource="&ecom;credit_card_num"/>
                     <drs:term_args rdf:parseType="Collection">
                        <swrl:Variable rdf:resource="#CC"/>
                     </drs:term_args>
                  </drs:Functional_term>
               </rdf:subject>
               <rdf:object rdf:resource="#Num"/>
            </drs:Atomic_formula>
        </Expression>
     </hasPrecondition>
  </Description>
```

The references to `#CC` and `#Num` in the DRS example are to parameters, that same ones written as `?cc` and `?num` in the Kif example. We haven't yet provided a mechanism for declaring the scopes of variables (see section 1.2) and how they acquire values. In the example, `#CC` is an input parameter to the process, that is, supplied by the client, but `#Num` is supposed to be set in the process of reasoning about this very precondition. Verifying that the process is feasible requires computing the available credit limit, which is then associated with the variable `#Num`. We call such a parameter a *local* parameter.

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

The three types of parameter are disjoint:

```
<rdf:Description rdf:about="#Input">
  <owl:disjointWith rdf:resource="#Output"/>
  <owl:disjointWith rdf:resource="#Local"/>
</rdf:Description>

<rdf:Description rdf:about="#Output">
  <owl:disjointWith rdf:resource="#Local"/>
</rdf:Description>
```

Of course, flagging bits of RDF as "Literals" means that an RDF parser should ignore them. (If it did not, then it might turn the RDF into a set of triples with a simple declarative meaning, which is not appropriate.) The trick is to have the OWL-S parser extract the ignored stuff and interpret it appropriately for its context, treating it as ordinary RDF after transformations such as replacing occurrences of variables with their values. In the example above, the occurrences of `#num` and `#cc` are interpreted as the values of these variables, not the variables themselves. In the Kif example, the expressions `?num` and `?cc` must be similarly interpreted. It is usually not too difficult to do this sort of "field engineering" to interface an assertional language to RDF.

There are two special cases of `Expression`: `Condition` and `Effect`. Because they are implemented as literals, there is no way to declare what this difference is, but it's a useful distinction for a human reader of the ontology.

```
<owl:Class rdf:ID="Condition">
   <owl:subClassOf rdf:resource="\&expr;#Expression"/>
</owl:Class>

<owl:Class rdf:ID="Effect">
   <owl:subClassOf rdf:resource="\&expr;#Expression"/>
</owl:Class>
```

1.2 Process Parameters and Results

We connect processes to their "IOPEs" using the properties shown in this table:

Property	*Range*	*Kind*
`hasInput`	`Input`	*Parameter*
`hasOutput`	`Output`	*Parameter*
`hasLocal`	`Local`	*Parameter*
`hasPrecondition`	`Condition`	*Expression*
`hasResult`	`Result`	(see below)

As promised above, the links from a process to its parameters implicitly gives them *scope*. Input, output, and local parameters have as scope the entire process they occur in. We introduce *result vars* below, which have a narrower scope.

In the rest of this section, we will discuss the entries in this table.

- **Inputs and Outputs**

  Inputs and outputs specify the data transformation produced by the process. Inputs specify the information that the process requires for its execution. For atomic processes, the information must come from the client. For the pieces of a composite process, some inputs come directly from the client, but others come from previous steps of the process.

  We said above that an atomic process corresponds to a one-step service that expects one message, so it might appear to be contradictory to allow an atomic process to have multiple inputs. The contradiction is resolved by distinguishing between the *inputs* and the *message* sent to a process. There is just one message, but it can bundle as many inputs as required. The bundling is specified by the *grounding* of the process model; see section **??**.

Similarly, the outputs produced by the invocation of an atomic process flow back to the client as a single message, the format of which is specified by the grounding. Within a composite process, some steps' outputs are sent back to the client, and some are used as inputs to later steps. Exactly where inputs come from and where they go is the subject of section 1.5.

The following example shows the definition of `hasParameter`, and its subproperties `hasInput`, `hasOutput`, and `hasLocal`:

```
<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:range rdf:resource="#Input"/>
</owl:ObjectProperty>

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

<owl:ObjectProperty rdf:ID="hasLocal">
  <rdfs:subPropertyOf rdf:resource="#hasParameter"/>
  <rdfs:range rdf:resource="#Local"/>
</owl:ObjectProperty>
```

- **Preconditions and Results**

  If a process has a precondition, it cannot be performed unless the precondition is true.

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

  Please be sure to distinguish between a condition's being true and having various other properties, such as being believed to be true, being known to be true, being represented in a database as true, etc. In OWL-S, if a process's precondition is false, the consequences of performing or initiating the process are undefined.

The performance of a process may result in changes of the state of the world (effects), and the acquisition of information by the agent performing it (outputs). However, we don't link processes directly to effects and outputs, because process modelers often want to model the dependence of these on context. For example, if a process contains a step to buy an item, there are two possible outcomes: either the purchase succeeds or it fails. In the former case, the effect is that ownership is transferred and the output is, say, a confirmation number. In the latter case, there is no effect, and the output is a failure message.

We use the term *result* to refer to a coupled output and effect.

```
<owl:Class rdf:ID="Result">
  <rdfs:label>Result</rdfs:label>
</owl:Class>

<owl:ObjectProperty rdf:ID="hasResult">
  <rdfs:label>hasResult</rdfs:label>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Result"/>
</owl:ObjectProperty>
```

- **Conditioning Outputs and Effects**

  Having a declared a result, a process model can then describe it in terms of four properties

```
<owl:ObjectProperty rdf:ID="inCondition">
  <rdfs:label>inCondition</rdfs:label>
  <rdfs:domain rdf:resource="#Result"/>
  <rdfs:range rdf:resource="&expr;#Condition"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="resultVar">
  <rdfs:label>resultVar</rdfs:label>
  <rdfs:domain rdf:resource="#Result"/>
  <rdfs:range rdf:resource="#ResVar"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="withOutput">
  <rdfs:label>withOutput</rdfs:label>
  <rdfs:domain rdf:resource="#Result"/>
  <rdfs:range rdf:resource="#OutputBinding"/>
</owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:ID="hasEffect">
  <rdfs:label>hasEffect</rdfs:label>
  <rdfs:domain rdf:resource="#Result"/>
  <rdfs:range rdf:resource="&expr;#Expression"/>
</owl:ObjectProperty>
```

The `inCondition` property specifies the condition under which this result (and not another) occurs. The `withOutput` and `hasEffect` properties then state what ensues when the condition is true. The `resultVar` property declares variables that are bound in the `inCondition`. These variables, called `ResVars`, are analogous to `Locals`, and serve an isomorphic purpose, allowing variables to be bound in preconditions and then be used to declare outputs and effects.

```
<owl:Class rdf:about="ResVar">
  <rdfs:subClassOf rdf:resource="#Parameter"/>
  <owl:disjointWith rdf:resource="#Input"/>
  <owl:disjointWith rdf:resource="#Output"/>
  <owl:disjointWith rdf:resource="#Local"/>
</owl:Class>
```

A typical example is a process that charges a credit card. The charge goes through if the card is not overdrawn. If it is overdrawn, the only output is a failure notification. So the description of the process must include the description of two `Results`, possibly in this form:

```
<AtomicProcess>
   <hasInput/>
      <Input rdf:ID="ObjectPurchased"/>
   </hasInput>
   <hasInput/>
      <Input rdf:ID="PurchaseAmt"/>
   </hasInput>
   <hasInput/>
      <Input rdf:ID="CreditCard"/>
   </hasInput>
   <hasOutput/>
      <Output rdf:ID="ConfirmationNum"/>
   </hasOutput>
   <owls:hasResult>
     <owls:Result>
        <owls:resultVar>
           <owls:ResVar rdf:ID="CreditLimH">
              <owls:parameterType rdf:resource="&ecom;#Dollars"/>
           </owls:ResVar>
      </owls:resultVar>
      <owls:inCondition expressionLanguage="&kif;#Kif"
                        rdf:dataType="&xsd;#string">
         (and (current-value (credit-limit ?creditCard)
                             ?CreditLimH)
              (>= ?CreditLimH ?purchaseAmt))
      </owls:inCondition>
      <owls:withOutput>
         <owls:OutputBinding>
            <owls:toParam rdf:resource="#ConfirmationNum"/>
            <owls:resultForm rdf:parseType="Literal">
               <cc:ConfirmationNum xsd:datatype="&xsd;#string"/>
            </owls:resultForm>
         </owls:OutputBinding>
      </owls:withOutput>
      <hasEffect expressionLanguage="&kif;#Kif"
                 rdf:dataType="&xsd;#string">
         (and (confirmed (purchase ?purchaseAmt) ?ConfirmationNum)
              (own ?objectPurchased)
              (decrease (credit-limit ?creditCard)
                        ?purchaseAmt))
      </hasEffect>
   </owls:Result>
   <owls:Result>
      <owls:resultVar>
         <owls:ResVar rdf:ID="CreditLimL">
            <owls:parameterType rdf:resource="&ecom;#Dollars"/>
         </owls:ResVar>
      </owls:resultVar>
      <owls:inCondition expressionLanguage="&kif;#Kif"
                        rdf:dataType="&xsd;#string">
         (and (current-value (credit-limit ?creditCard)
                             ?CreditLimL)
              (< ?CreditLimL ?purchaseAmt))
      </owls:inCondition>
      <withOutput rdf:resource="&ecom;failureNotice"/>
         <owls:OutputBinding>
            <owls:toParam rdf:resource="#ConfirmationNum"/>
            <owls:valueData rdf:parseType="Literal">
               <drs:Literal>
                  <drs:litdefn xsd:datatype="&xsd;#string">00000000</drs:litdefn>
               </drs:Literal>
            </owls:valueData>
         </owls:OutputBinding>
      </withOutput>
```

FIGURE [[file=ProcessModel.eps, width=5in, clip=]] GOES HERE

Figure 1: Top level of the process ontology

```
      </owls:Result>
    </owls:hasResult>
  </AtomicProcess>
```

As a result of the execution of the process, a credit card is charged and the money in the account reduced. Note, once again, that there is a fundamental difference between effects and outputs. Effects describe conditions in the world, while outputs describe information. In a more realistic version of this example, the service may send a notification, or an invoice, that it charged the credit card account. This output is just a datum of one type or another. The effect describes the actual event that the output is part of the description of: that the amount of money in the credit card account has been reduced and that the client now owns the object it intended to purchase.

1.3 Atomic and Simple Processes

We are now ready to formalize the classes of processes: atomic, composite, and, not mentioned before, "simple."

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

See figure 1.

Atomic processes correspond to the actions a service can perform by engaging it in a single interaction; composite processes correspond to actions that require multi-step protocols; finally, simple processes provide an abstraction mechanism to provide multiple views of the same process. We discuss atomics and simples here, reserving composites for the next subsection.

Atomic processes are directly invocable (by passing them the appropriate messages). Atomic processes have no subprocesses and execute in a single step, as far as the service requester is concerned. That is, they take an input message, do something, and then return their output message. For each atomic process, there must be provided a grounding that enables a service requester to construct messages to the process and deconstruct replies, as explained in Section **??**.

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

<owl:DatatypeProperty rdf:ID="invocable">
  <rdfs:comment>
    Invocable is a flag that tells whether a Process bottoms
    out in atomic processes; clearly, every atomic process is
    ipso facto invocable.
  </rdfs:comment>
  <rdfs:domain rdf:resource="#CompositeProcess"/>
  <rdfs:range  rdf:resource="&xsd;#boolean"/>
</owl:DatatypeProperty>
```

Simple processes are not invocable and are not associated with a grounding, but, like atomic processes, they *are* conceived of as having single-step executions. Simple processes are used as elements of abstraction; a simple process may be used either to provide a view of (a specialized way of using) some atomic process, or a simplified representation of some composite process (for purposes of planning and reasoning). In the former case, the simple process is `realizedBy` the atomic process; in the latter case, the simple process `expandsTo` the composite process (see below).

```
<owl:Class rdf:ID="SimpleProcess">
  <rdfs:subClassOf rdf:resource="#Process"/>
  <owl:disjointWith rdf:resource="#AtomicProcess"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="realizedBy">
  <rdfs:domain rdf:resource="#SimpleProcess"/>
  <rdfs:range rdf:resource="#AtomicProcess"/>
  <owl:inverseOf rdf:resource="#realizes"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="realizes">
  <rdfs:domain rdf:resource="#AtomicProcess"/>
  <rdfs:range rdf:resource="#SimpleProcess"/>
  <owl:inverseOf rdf:resource="#realizedBy"/>
</owl:ObjectProperty>
```

1.4 Composite Processes

Composite processes are decomposable into other (non-composite or composite) processes; their decomposition can be specified by using control constructs such as `Sequence` and `If-Then-Else`, which are discussed below. Because many of the control constructs have names reminiscent of control structures in programming languages, it is easy to lose sight of a fundamental difference: a composite process is not a behavior a service *will* do, but a behavior (or set of behaviors) the client *can* do by sending and receiving a series of messages. If the composite process has an overall effect, then the client must perform the entire process in order to achieve that effect. We have not yet given a precise specification of what it means to perform a process, but all we mean is that, e.g., if a composite is a `Sequence`, then the client sends a series of messages that invoke every step in order. However, the world will not end if the client stops half way through. It may not achieve the effects associated with the entire composite process, but it might have a more devious purpose.

One crucial feature of a composite process is its specification of how its inputs are accepted by particular subprocesses, and how its various outputs are produced by particular subprocesses. We discuss this topic in section 1.5.

```
<owl:Class rdf:ID="CompositeProcess">
  <rdfs:subClassOf rdf:resource="#Process"/>
  <owl:disjointWith rdf:resource="#AtomicProcess"/>
  <owl:disjointWith rdf:resource="#SimpleProcess"/>
  <rdfs:comment>
    A CompositeProcess must have exactly 1 composedOf property.
  </rdfs:comment>
  <owl:intersectionOf rdf:parseType="Collection">
      <owl:Class rdf:about="#Process"/>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#composedOf"/>
         <owl:cardinality rdf:datatype="&xsd;#nonNegativeInteger">
                  1</owl:cardinality>
      </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

A process can often be viewed at different levels of granularity, either as a primitive, undecomposable process or as a composite process. These are sometimes referred to as “black box” and “glass box” views, respectively. Either perspective may be the more useful in some given context. When a composite process is viewed as a black box, a simple process can be used to represent this. In this case, the relationship between the simple and composite is represented using the `expandsTo` property, and its inverse, the `collapsesTo` property.

```
<owl:ObjectProperty rdf:ID="expandsTo">
  <rdfs:domain rdf:resource="#SimpleProcess"/>
  <rdfs:range rdf:resource="#CompositeProcess"/>
  <owl:inverseOf rdf:resource="#collapsesTo"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="collapsesTo">
  <rdfs:domain rdf:resource="#CompositeProcess"/>
  <rdfs:range rdf:resource="#SimpleProcess"/>
  <owl:inverseOf rdf:resource="#expandsTo"/>
</owl:ObjectProperty>
```

A `CompositeProcess` must have a `composedOf` property by which is indicated the control structure of the composite, using a `ControlConstruct`.

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

<owl:Class rdf:ID="ControlConstruct">
</owl:Class>
```

Each control construct, in turn, is associated with an additional property called `components` to indicate the ordering and conditional execution of the subprocesses (or control constructs) from which it is composed.

```
<owl:ObjectProperty rdf:ID="components">
 <rdfs:domain rdf:resource="#ControlConstruct"/>
</owl:ObjectProperty>
```

For instance, any instance of the control construct `Sequence` has a `components` property that iranges over a `ControlConstructList` (a list of control constructs). We give a complete table of composite control constructs below.

Any composite process can be considered a tree whose nonterminal nodes are labeled with control constructs, each of which has children specified using `components`. The leaves of the tree are invocations of other processes, indicated as an object of class `Perform`.

```
<owl:Class rdf:ID="Perform">
  <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#process"/>
      <owl:cardinality rdf:datatype="&xsd;#nonNegativeInteger">
                    1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

The `process` property of a perform is the process performed:

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

When a process is performed as a step in a larger process, there must be a description of where the inputs to the performed process come from and where the outputs go. This issue we defer to section 1.5.

We conclude this section with an overview of the OWL-S control constructs: Sequence, Split, Split + Join, Choice, Unordered, Condition, If-Then-Else, Iterate, Repeat-While, and Repeat-Until.

Sequence : A list of control constructs to be done in order.

```
<owl:Class rdf:ID="Sequence">
  <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#components"/>
      <owl:allValuesFrom rdf:resource="#ControlConstructList"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="ControlConstructList">
<rdfs:comment> A list of control constructs </rdfs:comment>
  <rdfs:subClassOf rdf:resource="&shadow-rdf;#List"/>
  <rdfs:subClassOf>
   <owl:Restriction>
    <owl:onProperty rdf:resource="&shadow-rdf;#first"/>
    <owl:allValuesFrom rdf:resource="#ControlConstruct"/>
   </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
   <owl:Restriction>
    <owl:onProperty rdf:resource="&shadow-rdf;#rest"/>
    <owl:allValuesFrom rdf:resource="#ControlConstructList"/>
   </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Split : The components of a `Split` process are a bag of process components to be executed concurrently. No further specification about waiting or synchronization is made at this level.

```
<owl:Class rdf:ID="Split">
```

```
    <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#components"/>
        <owl:allValuesFrom rdf:resource="#ControlConstructBag"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:Class rdf:ID="ControlConstructBag">
  <rdfs:comment> A multiset of control constructs </rdfs:comment>
    <rdfs:subClassOf rdf:resource="&shadow-rdf;#List"/>
    <rdfs:subClassOf>
     <owl:Restriction>
      <owl:onProperty rdf:resource="&shadow-rdf;#first"/>
      <owl:allValuesFrom rdf:resource="#ControlConstruct"/>
     </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
     <owl:Restriction>
      <owl:onProperty rdf:resource="&shadow-rdf;#rest"/>
      <owl:allValuesFrom rdf:resource="#ControlConstructBag"/>
     </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
```

Split+Join : Here the process consists of concurrent execution of a bunch of process components with barrier synchronization. With `Split` and `Split+Join`, we can define processes that have partial synchronization (e.g., split all and join some sub-bag).

```
  <owl:Class rdf:ID="Split-Join">
    <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#components"/>
        <owl:allValuesFrom rdf:resource="#ControlConstructBag"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
```

Unordered : Allows the process components (specified as a bag) to be executed in some unspecified order, or concurrently. All components must be executed. As with Split+Join, completion of all components is required. Note that, while the unordered construct itself gives no constraints on the order of execution, nevertheless, in some cases, there may be constraints associated with subcomponents, which must be respected.

```
<owl:Class rdf:ID="Unordered">
  <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#components"/>
      <owl:allValuesFrom rdf:resource="#ControlConstructBag"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Examples:

1. If all process components are atomic processes, any ordering is permitted. For instance, (Unordered a b) could result in the execution of a followed by b, or b followed by a.
2. Let a, b, c, and d be atomic processes, and X, Y, and Z be composite processes:

   ```
   X = (Sequence a b)
   Y = (Sequence c d)
   Z = (Unordered X Y)
   ```

   Z, then, translates to the following partial ordering:

   ```
   {(a;b), (c;d)}
   ```

   where ';' means "executes before", and the possible execution sequences (total orders) include

   ```
   {(a;b;c;d), (a;c;b;d), (a;c;d;b), (a;c;d;b),
    (c;d;a;b), (c;a;d;b), (c;a;b;d)}
   ```

Choice : `Choice` is a control construct whose key property is `chooseFrom`, whose value is a list of processes the execution of one of which constitutes execution of the `Choice`.

[[This wording is a significant scale-down from the original idea, which involved being able to choose an arbitrary number of the processes specified as `chooseFrom`, and then impose further constraints on the set chosen. We have decided to simplify because the machinery used to implement the original idea was not thought through carefully, and because there doesn't seem to be much demand for the complicated version.]]

```
<owl:Class rdf:ID="Choice">
  <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
```

```
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#components"/>
        <owl:allValuesFrom rdf:resource="#ControlConstructBag"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:ObjectProperty rdf:ID="chooseFrom">
     <rdfs:domain rdf:resource="#Choice"/>
     <rdfs:range rdf:resource ="#ControlConstructBag"/>
  </owl:ObjectProperty>
```

If-Then-Else : The `If-Then-Else` class is a control construct that has properties `ifCondition`, `then` and *else* holding different aspects of the `If-Then-Else`. Its semantics is intended as "Test `If-condition`; if True do *Then*, if False do `Else`." (Note that the class `Condition`, which is a placeholder for further work, will be defined as a class of logical expressions.)

```
  <owl:Class rdf:ID="If-Then-Else">
    <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty rdf:resource="#components"/>
        <owl:allValuesFrom rdf:resource="#ControlConstructBag"/>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>

  <owl:ObjectProperty rdf:ID="ifCondition">
   <rdfs:comment> The if condition of an if-then-else</rdfs:comment>
    <rdfs:domain rdf:resource="#If-Then-Else"/>
    <rdfs:range rdf:resource="&expr;#Condition"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="then">
    <rdfs:domain rdf:resource="#If-Then-Else"/>
    <rdfs:range rdf:resource="#ControlConstruct"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="else">
    <rdfs:domain rdf:resource="#If-Then-Else"/>
    <rdfs:range rdf:resource="#ControlConstruct"/>
  </owl:ObjectProperty>
```

Iterate : `Iterate` is a control construct whose `nextProcessComponent` property has the same value as the current process component. `Repeat` is defined as a synonym of the `Iterate` class. The repeat/iterate process makes no assumption about how many iterations are made or when to initiate, terminate, or resume. The initiation, termination or maintenance condition could be specified with a `whileCondition` or an `untilCondition` as below.[1]

```
<owl:Class rdf:ID="Iterate">
  <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#components"/>
      <owl:allValuesFrom rdf:resource="#ControlConstructBag"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Repeat-while and Repeat-until Both of these iterate until a condition becomes false or true. Whether the test occurs at a fixed place within the iteration or runs asynchronously varies from subclass to subclass of these classes.

```
<owl:ObjectProperty rdf:ID="whileCondition">
  <rdfs:domain rdf:resource="#Repeat-While"/>
  <rdfs:range rdf:resource="&expr;#Condition"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="whileProcess">
  <rdfs:domain rdf:resource="#Repeat-While"/>
  <rdfs:range rdf:resource="#ControlConstruct"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Repeat-While">
 <rdfs:comment> The repeat while construct</rdfs:comment>
 <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="untilCondition">
  <rdfs:domain rdf:resource="#Repeat-Until"/>
  <rdfs:range rdf:resource="&expr;#Condition"/>
</owl:ObjectProperty>
```

[1]Another possible extension is to ability to define counters and use their values as termination conditions. This could be part of an extended process control and execution monitoring ontology.

```
<owl:ObjectProperty rdf:ID="untilProcess">
  <rdfs:domain rdf:resource="#Repeat-Until"/>
  <rdfs:range rdf:resource="#ControlConstruct"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Repeat-Until">
  <rdfs:comment> The repeat until process</rdfs:comment>
  <rdfs:subClassOf rdf:resource="#ControlConstruct"/>
</owl:Class>
```

1.5 Specifying Data Flow and Parameter Bindings

When defining processes using OWL-S, there are many places where the input to one process component is obtained as one of the outputs of a preceding step, short-circuiting the normal transmission of data from service to client and back. This is one type of *data flow* from one step of a process to another. There are other, less familiar patterns; the outputs of a composite process are all derived as outputs of some of its components, and specifying which component's output becomes output X of the composite is also a data-flow specification.

We adopt the convention that the source of a datum is identified when the user of the datum is declared. If step 1 feeds step 3, we specify this fact in the description of step 3 rather than the description of step 1. We call this a *consumer-pull* convention, as opposed to a *producer-push* alternative. We implement this convention by providing a notation for arbitrary terms as the values of input or output parameters of a process step, plus a notation for subterms denoting the output or input parameters of prior process steps.

Consider the following tableau:

`I1` input of:{ Composite Process CP: }: with output `O1`

composed of

Step 1: Perform S1 $\longrightarrow$ Step 2: Perform S2

where S1 has inputs `I11` and `I12`, and output `O11`
and S2 has input `I21` and output `O21`

and suppose that we want a straightforward data flow: Input `I1` of the overall process CP is used as input `I11` of S1, after adding 1. Input `I12` of S1 is a constant, the string `"Academic"`. Output `O11` of S1 is used as input `I21` of S2. The maximum of 0 and output `O21` of S2, times π, is used as output `O1` of CP. Using a consumer-pull convention, we simply declare the parameters `I1`, `O11`, and `O21`, but for parameters `I11`, `I21`, and `O1` we provide, in addition to a declaration, *bindings* that specify that

`I11`(S1) *comes from* `incr(I1`(CP)`)`

$$\begin{array}{rcl} \texttt{I12}(\text{S1}) & = & \texttt{"Academic"} \\ \texttt{I21}(\text{S2}) & \textit{comes from} & \texttt{O11}(\text{S1}) \\ \texttt{O1}(\text{CP}) & \textit{comes from} & \pi \times \max(0, \texttt{O21}(\text{S2})) \end{array}$$

Each of these equalities is represented in OWL-S as a `Binding`, an abstract object with two properties: `toParam`, the name of the parameter (e.g., `I21`(S2)), and `valueSpecifier`, a description of its value. In an effort to provide value specifications in as concise a manner as possible in a variety of situations, we provide three different types: `valueSource`, `valueData`, and `valueFn`. .

We declare the `toParam` property in the usual way:

```
<owl:Class rdf:ID="Binding">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#toParam"/>
      <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">
                     1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

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

The simplest sort of dataflow spec is `valueSource`:

```
<owl:ObjectProperty rdf:ID="valueSource">
  <rdfs:label>valueSource</rdfs:label>
  <rdfs:domain rdf:resource="#Binding"/>
  <rdfs:range rdf:resource="#ValueOf"/>
  <rdfs:subPropertyOf rdf:resource="#valueSpecifier"/>
</owl:ObjectProperty>
```

The range of `valueSource` is a simple object of class `ValueOf`, specified entirely by its properties `theVar` and `fromProcess`. If a binding with `toParam`$= p$ has `valueSource` $= s$ with properties `theVar`$=v$ and `fromProcess`$= R$, that means that parameter p of this process = parameter v of R.

```
<owl:Class rdf:ID="ValueOf">
  <rdfs:label>ValueOf</rdfs:label>
  <rdfs:subClassOf>
    <owl:Restriction>
```

```
      <owl:onProperty rdf:resource="#theVar"/>
      <owl:cardinality rdf:datatype="&xsd;#nonNegativeInteger">
                1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#fromProcess"/>
      <owl:maxCardinality rdf:datatype="&xsd;#nonNegativeInteger">
                  1</owl:maxCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:ObjectProperty rdf:ID="theVar">
  <rdfs:domain rdf:resource="#ValueOf"/>
  <rdfs:range rdf:resource="#Parameter"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="fromProcess">
  <rdfs:domain rdf:resource="#ValueOf"/>
  <rdfs:range rdf:resource="#Perform"/>
</owl:ObjectProperty>
```

For example, our simple tableau has one place where a `ValueSource` expression can be used. Here is how that part of the tableau would be expressed:

```
<Sequence rdf:ID="CP">
   <hasInput>
      ...
   </hasInput>
   <hasOutput>
      ...
   </hasOutput>

   <components rdf:parseType="Collection">
      <Perform rdf:ID="S1">
         <process rdf:resource="&aux;#UsefulProcess"/>
         ...
      </Perform>
      <Perform rdf:ID="S2">
         <process rdf:resource="&aux;#AnotherUsefulProcess"/>
         <hasDataFrom>   <!-- see below -->
            <Binding>
               <theParam rdf:resource="&aux;#I21"/>
               <valueSource>
                  <ValueOf>
                      <theParam rdf:resource="#O11"/>
                      <fromProcess rdf:resource="#S1"/>
                  </ValueOf>
               </valueSource>
            </Binding>
         </hasDataFrom>
      </Perform>
   </components>
   ...
</Sequence>
```

In this example, we used the `hasDataFrom` property, which is used to link inputs of `Performs` to bindings:

```
<owl:ObjectProperty rdf:ID="hasDataFrom">
  <rdfs:domain rdf:resource="#ProcessComponent"/>
  <rdfs:range rdf:resource="#Binding"/>
</owl:ObjectProperty>
```

The complete tableau appears below, after further discussion of bindings.

There is a subtle issue that arises from the fact that the range of `fromProcess` is `Perform`. In our informal tableau above, we used expressions such as `incr(I1(CP))` to mean the the input `I1` of the overall process CP. But we cannot actually refer to a `Binding` with `valueSource` that is a `ValueOf` with `fromProcess` = CP, because CP is not a `Perform`, but a process. If you think about it, it makes no sense to refer to a value extracted from a process, because every time the process is invoked different values will be involved. We need an expression that refers to the "current value" of a parameter, that is, its value during an actual Perform. We want to say: During any Perform P of CP, the value of the input `I1` of step `S1` is the value of the input `I1` of P.

Hence we introduce a standard variable to play the role of P, and give it the name `TheParentPerform`.

```
<swrl:Variable rdf:ID="TheParentPerform">
  <rdfs:comment>
    A special-purpose variable, used to refer, at runtime, to the execution
    instance of the enclosing composite process definition.
  </rdfs:comment>
</swrl:Variable>
```

We will show how this device is used in our example tableau shortly.

The remaining two data-source specifications use the XML Literal trick to encode arbitrary expressions.

```
<owl:DatatypeProperty rdf:ID="valueFn">
  <rdfs:label>valueFn</rdfs:label>
  <rdfs:subPropertyOf rdf:resource="#valueSpecifier"/>
  <rdfs:domain rdf:resource="#Binding"/>
  <rdfs:range rdf:resource="&rdf;#XMLLiteral"/>
</owl:DatatypeProperty>

<owl:DatatypeProperty rdf:ID="valueData">
  <rdfs:label>valueData</rdfs:label>
  <rdfs:domain rdf:resource="#Binding"/>
</owl:DatatypeProperty>
```

The `valueFn` of a `Binding` is an XML literal to be read as a functional term. Some of its subterms may be `ValueOfs` specifying outputs of previous terms. As with conditions and effects, the denotation of the `valueFn` expression cannot be known until variable values are plugged in. The `valueData` of a `Binding` is an XML literal to be interpreted as constant data.

Here is our complete example tableau, with all data flows expressed using one of the three data-source specs.

```
<Sequence rdf:ID="CP">
   <hasInput>
      <Input rdf:ID="I1"/>
   </hasInput>
   <hasOutput>
      <Output rdf:ID="O1"/>
   </hasOutput>

   <components rdf:parseType="Collection">
      <Perform rdf:ID="S1">
         <process rdf:resource="&aux;#UsefulProcess"/>
         <hasDataFrom>
            <InputBinding>
               <theParam rdf:resource="&aux;#I11"/>
               <valueFn expressionLanguage="&drs;" rdf:parseType="Literal">
                  <drs:Functional_term>
                     <drs:term_function rdf:resource="&arith;#incr"/>
                     <drs:term_args rdf:parseType="Collection">
                        <valueOf>
                            <theParam rdf:resource="#I1"/>
                            <fromProcess rdf:resource="#TheParentPerform"/>
                        </valueOf>
                     </drs:term_args>
                  </drs:Functional_term>
               </valueForm>
            </InputBinding>
            <InputBinding>
               <theParam rdf:resource="&aux;#I12"/>
               <valueData xsd:datatype="&xsd;#string">Academic</valueData>
            </InputBinding>
         </hasDataFrom>
      </Perform>
      <Perform rdf:ID="S2">
         <process rdf:resource="&aux;#AnotherUsefulProcess"/>
         <hasDataFrom>
            <Binding>
               <theParam rdf:resource="&aux;#I21"/>
               <valueSource>
                  <ValueOf>
                      <theParam rdf:resource="#O11"/>
                      <fromProcess rdf:resource="#S1"/>
                  </ValueOf>
               </valueSource>
            </Binding>
         </hasDataFrom>
      </Perform>
   </components>

   <hasResult>
      <withOutput>
         <OutputBinding>
             <theParam rdf:resource="#O1"/>
             <valueFn expressionLanguage="&drs;" rdf:parseType="Literal">
                <drs:Functional_term>
                    <drs:term_function rdf:resource="&arith;#times"/>
                    <drs:term_args rdf:parseType="Collection">
                       <xsd:Integer rdf:datatype="&xsd;#Integer">1</xsd:Integer>
                       <drs:Functional_term>
                          <drs:term_function rdf:resource="&arith;#max"/>
                          <drs:term_args rdf:parseType="Collection">
                             <xsd:Integer rdf:datatype="&xsd;#Integer">0</xsd:Integer>
                             <valueOf>
                                <theParam rdf:resource="#O21"/>
                                <fromProcess rdf:resource="#S2"/>
                             </valueOf>
                          </drs:term_args>
                       </drs:Functional_term>
                    </drs:term_args>
                </drs:Functional_term>
               </valueFn>
            </OutputBinding>
         </withOutput>
      </hasResult>
   </Sequence>
```

Finally, there is another output descriptor, called `resultForm`. This is not attached to a variable, but to a Result:

```
<owl:DatatypeProperty rdf:ID="resultForm">
  <rdfs:label>resultForm</rdfs:label>
  <rdfs:domain rdf:resource="#Result"/>
  <rdfs:range rdf:resource="&rdf;#XMLLiteral"/>
</owl:DatatypeProperty>
```

The purpose of `resultForm` is to provide an abstract XML template for outputs sent back to the client. The reason we need such a template are subtle, and do not always apply. Normally the grounding suffices to express how the components of a message are bundled, i.e., how inputs are put together to make a message to a service, and how replies are disassembled into the intended outputs. In essence, all we can or need to do is build up and tear down record structures (in XML Schema terminology, `ComplexTypes`). But in the case of a process with multiple Results, it can be extremely useful to specify other features of an output message that indicate which result actually occurred, sparing us the chore of providing output fields to encode that information, or making the client infer it from the form of the other fields. That's what `valueForm` is for.

In our example of a credit-card transaction, we had two Results, one for the case where there was a sufficient balance to pay the bill, and one for when there wasn't. We could augment each result with a further binding, such as this one for the failure case:

```
     <owls:Result>
        <owls:resultVar>
           <owls:ResVar rdf:ID="CreditLimL">
              <owls:parameterType rdf:resource="&ecom;#Dollars"/>
           </owls:ResVar>
        </owls:resultVar>
        <owls:inCondition expressionLanguage="&kif;#Kif"
                          rdf:dataType="&xsd;#string">
           (and (current-value (credit-limit ?creditCard)
                               ?CreditLimL)
                (< ?CreditLimL ?purchaseAmt))
        </owls:inCondition>
        <owls:resultForm rdf:parseType="Literal">
            <ecom:CreditExceededFailure>
               <ecom:gap expressionLanguage="&kif;#Kif"
```

```
                    rdf:dataType="&xsd;#string">
             (- ?purchaseAmt ?CreditLimL)
          </ecom:gap>
       </ecom:CreditExceededFailure>
   </owls:resultForm>
   <withOutput rdf:resource="&ecom;failureNotice"/>
      ...
   </withOutput>
</owls:Result>
```