Surface Syntax for OWL-S-PAI
* DRAFT 0.5 **
The OWL-S coalition
Edited by Drew McDermott
October 13, 2003

1 Goals

This is a proposal for a surface syntax for the emerging "processes as instances" (PAI) notation for OWL-S. The formal semantics that was formerly conjoined has been split off, and will be dealt with in a separate document.

The goals of this exercise are to

1. Provide readable surface syntax
2. Explain how it relates to the usual RDF syntax

Change history:

V. 0.5, 2003-10-13:	Change 'parameterSpec' to 'parameterInst' to better match the notion of process instance. Replaced parameter IDs with attribute `paramName`. Streamlined tag-binding notation in RDF syntax. Specified convention for referring to parameters in logical formulas. Provided practical alternatives to $\downarrow\vdash\uparrow$ characters.

2 Syntax and Informal Semantics

The syntax we propose is somewhat Lisp-based, but not entirely. The main reason to go this route is that Lisp's concrete syntax is essentially isomorphic to its abstract syntax, so you can view this as a placeholder for a syntax with more infix operators and fewer parentheses.

The key concept in OWL-S is the *process*, which is an activity carried out by an agent, typically a web service or a client.

A *process definition* is a description of a process. Processes come in several flavors, atomic, simple, and various sorts of composite, distinguished by their control constructs (conditional, choice, parallel, loop, etc.).

We will assume that a process starts with its control construct, or the reserved words `Atomic` or `Simple`. So an if-then-else might look like

```
(If-Then-Else ...)
```

By convention, reserved control constructs have names starting with a capital letter.

2.1 Args, Results, Preconditions, Effects

Every process can have input and output parameters, described using fields `:args` and `:results`. Input and output parameters may have optional types, the OWL classes they belong to. So a simple sequence might be described thus:

```
(Sequence :args (a - Integer)
   ---steps---
   :results (b - String))
```

We also capitalize the name of classes used in type declarations. Multiple args and results can be expressed by putting more pieces into a single `:args` or `:results`, or by having multiple `:args` and `:results` specs, or by any convenient combination. A variant of `:results` is `:conditional-results`, as in

```
(Sequence ...
   :conditionalResults
      (:coCondition (or (expired card1)
                        ((balance card1) > (limit card1)))
       fail-message - String))
```

The general format of a conditional result is
`(:coCondition` P —*params*—`)`.

Besides `:args` and `:results`, there can be a `:locals` declaration.

Processes can also have preconditions, which must be true before the process can be started:

```
(Sequence :args (customer - Person)
          :precondition (exists (x - Credit-card)
                                (and (credit-card-of x customer)
                                     (not (maxed-out x))))
   ...)
```

If the precondition is not true when the process begins, then some sort of failure should occur. [[This is an area that still needs elaboration in the OWL-S context.]] An outside observer looking at this process description can assume that the process executor ensures that the precondition is true at the appropriate time. The process executor itself might employ a planner of some kind to elaborate the process with steps that make the precondition true.

Processes can also have effects, which are represented using `Effect` and `ConditionalEffect` expressions:

```
(ConditionaEffect
    :ceCondition P
    :ceEffect E)
```

Example:

```
(Atomic ...
    :args (cd - Credit-card newcharge - Number)
    :effect (ConditionalEffect
                :ceCondition (not (maxed-out cd))
                :ceCondition (not (stolen cd))
                :ceEffect (add (bal cd) newcharge)))
```

Note that a `ConditionalEffect` can have multiple conditions. The intent is that if all are true, the effect will be "imposed." (Please note that our examples, which all seem to be talking about credit cards, do *not* reflect any coherent theory of credit-card transactions! Think of them as a nonexhaustive sampler of good and bad ideas for representing actions involving credit cards.)

An unconditional effect is one that has no `ceConditions`. These can be written using the same notation as for `ConditionalEffect`, but with `UnConditionalEffect` substituted. Alternatively, one can just specify the effect. So the following are equivalent:

```
(Atomic ...
    :results (x - Number)
    :effect (UnConditionalEffect
                :ceEffect (know ((bal card) = x))))
```

and

```
(Atomic ...
    :results (x - Number)
    :effect (know ((bal card) = x)))
```

2.2 Process Instances, Tags, and Dataflow

There is a crucial distinction between a process and a *process instance*. The distinction is obvious in a case like this:

```
(Sequence
   (toggle-the-switch)
   (toggle-the-switch))
```

which contains two instances of the process `toggle-the-switch`. In keeping with RDF style, the description of a process may accompany one of its instances, or may be placed elsewhere. In the example above, `toggle-the-switch` must obviously be defined somewhere else. Instead, we could have written this:

```
(Sequence
   (Atomic-process :ID toggle-the-switch)
   (toggle-the-switch))
```

If we want to give a name to a process instance, we use the `tag` construct:

```
(tag-scope (tog1 tog2)
   (sequence
      (tag tog1 (simple-process :ID toggle-the-switch))
      (tag tog2 (toggle-the-switch))))
```

The `tag-scope` construct is necessarily to indicate the scope of the names. However, there is an obvious rule for filling in the scope if left implicit: The scope of a tag is as wide as possible but no wider than the innermost iteration or process definition (that is, with an `:ID` attribute). OWL-S syntax checkers should use this rule to fill in the scope of tags when left implicit.

We can use `tag` names to describe dataflows between steps. Suppose we have a process for authorizing uses of credit card. Because it must communicate with some computer in a central location, it sometimes times out if traffic to that computer is heavy. So the process has three possible outputs: `authorized`, `not-authorized`, and `timeout`. The process used by a retailer might be to try the subprocess one or two times and then give the customer the benefit of the doubt. First, some definitions:

```
(owl:Class CC-check-res
   (owl:oneOf (owl:Thing :ID authorized)
              (owl:Thing :ID not-authorized)
              (owl:Thing :ID timeout)))

(owl:Class CC-acc-status
   (owl:oneOf (owl:Thing accepted)
              (owl:Thing not-accepted)))

(Simple-process :ID check-auth
                :args (cc - Credit-card-data)
                :results (res - CC-check-res))
```

Now, a process using the entities defined:

```
(Sequence :args (cc - Credit-card-data)
          :results (final-res - CC-acc-status)
    (check-auth cc <= cc
                res => (ch1res(↓ check1)))
    (tag check1
       (If-Then-Else :args (ch1res - CC-check-res)
          :ifCondition (ch1res = timeout)
          :then
             (Sequence :results (ch2res - CC-acc-status)
                (check-auth cc <= cc
                            res => (ch2res(↓ check2)))
                (tag check2
                   (If-Then-Else :args (ch2res - CC-check-res)
                                 :results (res - CC-acc-status)
                      :ifCondition (ch2res = not-authorized)
                      :then (Value not-accepted => final-res)
                      :else (Value accepted => final-res))))
          :else
             (If-Then-Else
                :ifCondition (ch1res = authorized)
                :then (Value accepted => final-res)
                :else (Value not-accepted
                             => final-res)))))
```

An expression of the form e_1 `=>` e_2 may be embedded in any process expression. Here e_i is a *tagged parameter expression,* an unambiguous specification of a parameter of a particular step. The meaning of e_1 `=>` e_2 is that the value of parameter e_1, when it becomes available, also becomes the value of e_2. It is called a *dataflow expression.*

The format of the e_i's in a dataflow expression is p`(`$[\downarrow \mid \uparrow]$`[`$s$`]`), where s is an optional step tag. The presence of $\downarrow$ vs. $\uparrow$ tells us whether we are referring to an input or output parameter, and p tells us its name. So `ch2res(↓check2)` means the input parameter `ch2res` of `check2`, the second attempt to check the credit card. If the s part is omitted, it means the innermost process that the "`=>`" expression is found in that has an input or output parameter p.

An expression of the form *param*`(` $\uparrow$ `)` on the left of an "`=>`" may be abbreviated as simply *param*. Similarly, an expression e`=>`*param*`(` $\downarrow$ `)` may be abbreviated as *param*`<=`e.

Because ↓ and ↑ are not available on standard computer keyboards, we allow "`in`" and "`out`," or "`>`" and "`<`" instead.

There is an issue about what an OWL-S execution engine should do if a step has an unfilled input parameter but is otherwise ready to be executed. Our current position is that the engine should pause until the value of the parameter is available. It would probably be wise to avoid making this the *only* determinant of control flow. That is, if data flows from step 1 to step 2, it's a good idea to make sure that step 2 follows step 1 in a `Sequence`. However, this is not always possible; there are control patterns that can be expressed through dataflow and no other way (so far).

Another issue is whether a parameter of a step can get a value more than once. The (current) answer is No. The intent is to allow reasoners to make strong inferences about what exactly is flowing from one step to another without detailed analysis of how the channel between them is set. One consequence of this design decision is that nontrivial dataflow in loops can't really be represented with the tools at hand.

There is a built-in control contruct `Compute` that takes arbitrary inputs (including none) and outputs, `val`. For instance, it could take numerical data from two predecessor steps and sum them, thus:

```
(Compute:args (n1 n2 - Number)
        :results ((val (n1 + n2)) - Number))
```

The form

`(Compute :args (...)  :results ((val` E`) -` t`) (val =>` e`))`

can be abbreviated

`(Value :args (...)` E `=>` e`)`

2.3 Calling Processes

There are two ways to "call" a process: write `([Call]` *process-name* `...)`, or `(Invoke :service` S *process-name*`) ...)`. The former notation (in which `Call` is optional) means that the process with the given name is to be created and run as a subroutine of the current process. The second is more general, and means that a process with the given name is to be found or created, and the arguments are to be passed to it. The process might be run as a subroutine, but it might also be found on another host somewhere, and the arguments might be transmitted to it using (e.g.) SOAP messages. Which of these possibilities (among others) obtains depends on the service argument S, which might be the URL of a service description. Exactly what S consists of, and how the information there interacts with the *grounding* of the current process, are matters outside the scope of this document.

Two constructs exist to make it possible to write web services that may be invoked from another process:

`(Accept :service` *S* `:ID` *process-name* `:followWith` *process*`)`

declares that this process *implements* the service described by *S*. When some `Invoke` from another host finds this implementation, the *process* is executed.

To provide more flexibility, several alternative `Accept`s can be wrapped inside a `Select`:

```
(Select
    (Accept ...)
    (Accept ...)
    ...
    (Accept ...))
```

This construct allows a single host to implement several services.

[[We need to be clear about whether `Invoke` is nonblocking, or can be declared to be nonblocking; and under what circumstances an `Accept` starts a new thread.]]

2.4 Miscellaneous Control Constructs

All that remains is to sketch the various control constructs and their meanings.

* `(Choice` *List-of-processes*`)` chooses an element from the *List-of-processes* and executes it. Which one is chosen is unspecified; it is either chosen by machinery that is not revealed, or is the result of some planning process.

* `(Split` *List-of-processes*`)` spawns execution of all of the processes, in separate threads, as it were. The `Split` finishes immediately.

* `(Split+Join` *List-of-processes*`)` executes all the processes in the txrmitList-of-processes in parallel, then waits until all complete before proceeding.

* `(Repeat-While :whileCondition` *P* `:whileProcess` *Q*`)` executes *Q* until *P* is false, possibly zero times.

* `(Repeat-Until :untilCondition` *P* `:untilProcess` *Q*`)` executes *Q* until *P* is true, possibly zero times.

[[A BNF syntax will go here when the notation is a bit more stable.]]

3 Relationship to "Deep" Syntax (RDF)

The original syntax for OWL-S was based on RDF and OWL, for the good reason that it provides a declarative description of a process as a set of assertions ("triples"). In this section we explain how the new surface syntax relates to the RDF/OWL syntax.

A process specification corresponds to a *description* of a process. So (*Construct* ...) corresponds to the RDF

<*Construct*>
...
</*Construct*>

The class *Construct* we refer to as a `control class`; it is that class of process whose construct is *Construct*. The fields of a control construct then become properties of the object being described. This applies in a straightforward way to fields like `:then` and `:else` whose values are themselves processes. The constructs `Sequence`, `Split`, and `Split+Join` have an indefinite number of subprocesses. In the deep syntax, we use the property `components` to specify a property of the process whose values are bags of processes. So (`Sequence` p_1 p_2 ...p_n) is translated into

<Sequence>
<components rdf:parseType="Collection">
p_1*
p_2*
...
p_n*
</components>
</Sequence>

(where p_i* is the RDF form of p_i). Similarly for `Split` and `Split+Join`.

Processes have zero or more `arg` properties and zero or more `result` properties. [[Formerly known as inputs and outputs.]] The value of each is an object of the class `Parameter`, or, more likely, one of its subclasses, `InParameter` or `OutParameter`. We need a way to declare the type of the values of the parameter, which is not the same as the type of the parameter itself (which is always `InParameter` or `OutParameter`). To avoid having to use OWL-Full, we do this with a property `parameterValue` suitably restricted. Example:

<Atomic>

```
    <arg>
       <InputParameter rdf:ID="cd1">
             <rdf:type>
                <owl:Restriction>
                   <owl:onProperty rdf:resource="&owl-s;parameterValue"/>
                   <owl:allValuesFrom rdf:resource="&cc;CreditCard"/>
                </owl:Restriction>
             </rdf:type>
       </InputParameter>
    </arg>
</Atomic>
```

The property `parameterValue` should be read as "has as possible value." (We can't refer to the *actual* value of a parameter without an ontology of execution traces, which does not yet exist.) So the example above says that the `cd1` input parameter must be a `CreditCard`.

The hard part of describing preconditions and effects in RDF is, as always, the fact these objects are formulas and terms obeying a recursive grammar. Here we take an agnostic view on which gimmick to use in representing such expressions, and just assume there is a class `Condition` and a class `Effect`. (We have put forth proposals for representing these things in the past, so this is not exactly an omission in OWL-S, just a hole among whose unappetizing fillers we are still reluctant to choose.) In some domains, `Effects` are just `Conditions`, but we reserve the right to use expressions like `(add (bal cd1) (cost mercedes-benz-2))`, which says to increase the balance on `cd1` by some (huge) amount of money.

We still need the classes `ConditionalOutput` and `ConditionalEffect`, with properties `coCondition`, `coOutput`, `ceCondition`, and `ceEffect`. The class `UnconditionalEffect` is a subclass of `ConditionalEffect` restricted to having zero `ceConditions`.

Tags must be handled with some care in RDF. The `tag-scope` construct behaves like a variable binder. We can have a `TagBind` control class with two properties: `tagName` and `process`. The latter has cardinality 1, the former has no bounds on cardinality.

The tag is actually declared by giving a process a `tag` property, whose value is a `tagSpec`. Here is an example. The surface process spec

```
(tag-scope (toot foof)
   (If-Then-Else
       :ifCondition ...
       :then (tag (A) ...)
```

```
        :else (tag (B) ...)))
```

would be represented by the RDF

```
<TagScope>
   <tagName xsd:datatype="&xsd;string">toot</tagName>
   <tagName xsd:datatype="&xsd;string">foof</tagName>
   <process>
      <If-Then-Else>
          <ifCondition> ... </ifCondition>
          <then>
             <Call>
                <tag xsd:datatype="&xsd;string">toot</tag>
                <callee ref:resource="#A"/>
             </Call>
          </then>
          <else>
             <Call>
                <tag xsd:datatype="&xsd;string">foof</tag>
                <callee ref:resource="#B"/>
             </Call>
          </else>
      </If-Then-Else>
   </process>
</TagScope>
```

Dataflows are objects of class `DataFlow`, which has two properties `source` and `destination`, each of which is an object of type `ParameterInst`. A `ParameterInst` is defined by its `psParam`, `i-or-o`, and `psStep` properties. The property `flow` connects a process to the dataflows involving it.

So, for instance, the surface example

```
(Sequence
   (tag step1 (A pen => ult(↓step2)))
   (tag step2 (B)))
```

would look thus in RDF:

```
<TagBind>
   <tagName xsd:datatype="&xsd;string">step1</tagName>
   <tagName xsd:datatype="&xsd;string">step2</tagName>
   </tagBound>
```

```
   <process>
      <Sequence>
         <components rdf:parsetype="Collection">
            <Call>
               <tag xsd:datatype="&xsd;string">step1</tag>
               <callee rdf:resource="#A"/>
            </Call>
            <Call>
               <tag xsd:datatype="&xsd;string">step2</tag>
               <callee rdf:resource="#B"/>
            </Call>
         </components>
         <flow>
            <DataFlow>
               <source>
                  <ParameterInst psParam="pen"
                                 i-or-o="&owl-s;out"
                                 psStep="step1"/>
               </source>
               <destination>
                  <ParameterInst psParam="ult"
                                 i-or-o="&owl-s;in"
                                 psStep="step2"/>
               </destination>
            </DataFlow>
         </flow>
      </Sequence>
   </process>
</TagBind>
```

Unfortunately, most of the abbreviating conventions we can exploit in the surface syntax do not apply in the deep syntax. The RDF version is fairly readable, but difficult for humans to write without error.

Within a condition or effect, we adopt the convention that a reference to a `ParameterInst` is taken to mean the value of that parameter at the step in question. So the condition that the balance on a credit card be 0 would be expressed as

```
<Atomic-formula>
    <rdf:predicate rdf:resource="&math;equal"/>
    <ParameterInst psParam="cd1" i-or-o="&owl-s;in" psStep="st33"/>
```

```
    <rdf:subject rdf:resource="#cd1"/>
    <rdf:object xsd:datatype="&xsd;integer">0</rdf:object>
</Atomic-formula>
```

using the DRS formalism, and assuming that `cd1` is declared as a parameter as at the beginning of section 3.

[[Compute is not yet mapped to a deep construct.]]

4 Ontology for Deep Syntax

[[To be released any day now.]]

5 Comments, conclusions, future directions

The wealth of new material we have introduced here may make some users of OWL-S (and DAML-S) uneasy. Just how stable is this language? Actually, almost all the changes we have made are *augmentations* to the notation, not incompatible changes. The decision to represent processes as instances instead of classes has made it much easier to fill in gaps that had stood empty for a long time.

Although we provide the iterative constructs `Repeat-While` and `Repeat--Until`, we provide no way for them to (say) add up the values received from some source. The only reason for this omission is that it would require generalizing channels a bit. A loop requires the idea of an *accumulator*, which changes in a clearly specified way on each iteration. At most once per iteration a value is sent to the accumulator, and combined with the value that's already there. Probably the best way to model accumulators is as parameters that contain a history list of the values accumulated to date.