Note on Conditions in DAML-S:

Author: Drew McDermott *This note is "non-normative" as they say in W3C-land.* It should be thought of as a proposal for representing conditions and other terms with recursive syntax, not an official part of DAML-S.

1 The DRS Notation

RDF, and notations based on it such as DAML+OIL and OWL, are not very good at representing arbitrary propositions, which are necessary at various points in the representation of processes and plans. There are various official ("DAML rules") and semiofficial ("RuleML") efforts to solve this problem. The approach advocated here is described in more detail in
Drew McDermott and Dejing Dou 2002 Representing Disjunction and Quantifiers in RDF Embedding logic in DAML/RDF. *Proc. Int'l Semantic Web Conference*. Available at `http://www.cs.yale.edu/~dvm`. That paper acknowledges contributions from many other people to the devices I describe below.

Recent work on making RuleML compatible with RDF seems to be going in a similar direction (`http://www.dfki.uni-kl.de/ruleml/indtd0.8.html`).

The committee designing DAML Rules may end up extending DAML/RDF syntax in various ways, but the rest of us must live with it as it is. That means that complex formulas must be described as objects whose pieces are subformulas. A rule processor must then take such a description as an assertion that the rule is true, if it occurs in an appropriate context. For DAML-S, an appropriate context would be the precondition or effect of a step in a process.

The building blocks of a formula are terms and atomic formulas. An atomic formula is represented in the XML serialization of RDF thus:

```
<drs:Atomic_formula>
   <rdf:subject>Ts</rdf:subject>
   <rdf:predicate>P</rdf:predicate>
   <rdf:object>Tb</rdf:object>
</drs:Atomic_formula>
```

where P is a predicate, T_s is a term,and T_b is a term if P is binary. If P is not binary, we have to be more devious about T_b, as we will describe shortly. We assume that the namespace `rdf` is the usual one for RDF (URI: `http://www.w3.org/1999/02/22-rdf-syntax-ns`). We are borrowing the standard tags for "reifying" RDF expressions, because, at the atomic-formula level, that's exactly what we're doing. The namespace `drs` is for vocabulary items in our formula-representation system (URI: `http://www.cs.yale.edu/~dvm/daml/drsonto.daml#`). "DRS" stands for "Declarations in RDF made Simple."

In the simple cases, the objects P, T_s, and T_b of an atomic formula are resources, just as one would expect. For example, to assert that that Tony Blair's address is 10 Downing Street, one might write

```
<drs:Atomic_formula>
   <rdf:subject resource="...URI for Blair"/>
   <rdf:predicate resource="...URI for address"/>
   <rdf:object>
      <adr:Address>
         <adr:number>10</adr:number>
         <adr:street>Downing Street</adr:street>
         <adr:city resource="...URI for London..."/>
         ...
      </adr:Address>
   </rdf:object>
</drs:Atomic_formula>
```

Note that as far as RDF is concerned, `Atomic_formula` is just another class. Its members are formulas. It is described by a DAML+OIL ontology (located at the same URI as the one we use for the `drs` namespace: `http://www.cs.yale.edu/~dvm/daml/drsonto.daml#`). So technically all we are doing here is *describing* formulas, with the unstated convention that any formula whose description occurs at the top level of a document is considered asserted by that document. A human or computational reader of the notation that doesn't know the convention will treat the descriptions as mere descriptions (and pretty useless ones at that).

If a predicate has arity > 2, its `rdf:object` slot is filled by a a *term sequence*, represented as a DAML+OIL list of all arguments after the first. We use the same idea for representing nonatomic terms. For example, suppose we have a function `person-name` that takes three arguments, the family name, given name, and middle initial(s). In predicate calculus, this might yield the term `person-name("Musharraf", "Shlomo", "")`. In our RDF serialization, this term would be written

```
<drs:Functional_term>
   <drs:term_function resource="....#person-name"/>
   <drs:term_args>
      <drs:Term_seq>
         <rdf:_1 xsd:type="xsd:string">Musharraf</rdf:_1>
         <rdf:_2 xsd:type="xsd:string">Shlomo</rdf:_2>
         <rdf:_3 xsd:type="xsd:string"></rdf:_3>
      </drs:Term_seq>
   </drs:term_args>
</drs:Functional_term>
```

(Brief and somewhat flaming digression: In case you're wondering why we represented one object as an `Address` *description* and another as a `person-name` *term,* the answer is that you can do whichever seems more natural. The only caveat is that any properties of an object described using RDF are actually asserted, so if you describe a "hypothetical" object of some kind, you have to realize that RDF treats it as a description of an actual object. In other words, if you give Santa Claus's address as 0000 North Pole Drive, then as far as RDF is concerned you have asserted that there is such a roadway and a 0th location along its length. This "feature" is why we have to resort to reification in the first place.)

If an atomic formula has exactly one argument, the `object` slot is occupied by a special object called `null` from the DRS namespace. If it has zero arguments, both the `subject` and `object` slots are both `null`.

Above the level of atomic formulas we have connectives, which are just classes of formulas with names `Implies`, `And`, `Or`, and `Not`. Formulas of classes `And` and `Or` have a single property `conn_args` which is a `Term_seq` of the arguments. Formulas of class `Implies` have two properties, `antecedent` and `consequent`. Formulas of class `Not` have a single property `negated`. One can easily more connectives if they are needed.

Variables in this system are treated as belonging to a class called `drs:Var`. Variables have unique `ID`s, as other objects do. By "unique" I mean "unduplicated elsewhere in the document," as usual with XML/RDF ids. Variables may also have `name`s, which link them back to their source form. The unique ID is necessary to distinguish different variables with the same name.

Variables are bound with quantifiers, which are objects with two properties: the `bound_vars` (a DAML+OIL list of variables) and the `body` (a formula). There are two quantifier classes, `Forall` and `Exists`, both subclasses of `Binder`. You can add new quantifiers-like entities, such as λ, by subclassing `Binder`. You can even do without quantifiers, and use Prolog-style conventions for free variables. Example: $\forall xy(\texttt{ loves}(x, y) \supset \texttt{ loves}(y, x))$ would be represented as

```
<drs:Forall>
   <drs:bound_vars>
      <drs:Var_bag>
         <rdf:_1>
            <drs:Var ID="v1" name="x"/>
         </rdf:_1>
         <rdf:_2>
            <drs:Var ID="v2" name="y"/>
         </rdf:_2>
      </drs:Var_bag>
   </drs:bound_vars>
   <drs:body>
      <drs:Implies>
```

```
        <drs:antecedent>
           <drs:Atomic_formula>
              <rdf:predicate
                 resource="http://human.rels.org/term#loves"/>
              <rdf:subject resource="#v1"/>
              <rdf:object resource="#v2"/>
           </drs:Atomic_formula>
        </drs:antecedent>
        <drs:consequent>
           <drs:Atomic_formula>
              <rdf:predicate
                 resource="http://human.rels.org/term#loves"/>
              <rdf:subject resource="#v2"/>
              <rdf:object resource="#v1"/>
           </drs:Atomic_formula>
        </drs:consequent>
     </drs:Implies>
  </drs:body>
</drs:Forall>
```

As formulas get larger and more deeply, they tend to wander annoyingly far to the right. Fortunately, we can fix that by moving subformulas out and making reference to them. I will make use of that device below.

2 Application to DAML-S

At several places DAML-S has slots for conditions or effects of actions, but these are of class `Condition` or `ConditionalEffect`, which have no interesting properties. The notation sketched above has the potential to flesh these classes out.

For example, in the "Congo" service, the atomic process `SpecifyPayment-Method` might have a conditional effect "paymentOkay" with condition

```
(forall (cc - Credit-card sl bal pb dif - Dollars)
   (and (credit-card cc
            creditCardType creditCardNum creditCardExpDate)
        (spending-limit cc sl)
        (balance cc bal)
        (difference sl bal dif)
        (price book pb)
        (>= dif pb)))
```

where `creditCardType`, `creditCardNum`, `creditCardExpDate`, and `cust` are inputs to the process derived from previous steps in the Congo process. As described in section 5.2 of the Technical Overview, we should consider these to be names of *value links,* which tie together the input/outputs of one step and the input/outputs of other steps which should be the same. Two cases are where one step's output is the input of a later step, and where a subprocess's output is also an output of its superprocess.

The first case is exemplified by this RDF:

```
<rdf:Description rdf:about="&congoProc;FullCongoBuy">
   <process:sameValues ID="isbnLink"
                        rdf:parsetype="daml:collection">
      <process:ValueOf
         process:atClass="&congoProc:LocatedBookOutput"
         process:theProperty="&congoProc:outInCatalogueBookISBN"/>
      <process:ValueOf
         process:atClass="&congoProc:PutInCart"
         process:theProperty="&congoProc:putInCartISBN"/>
   </process:sameValues>
</rdf:Description>
```

which states that the conditional output `outInCatalogueBookISBN` of the step `LocateBook` in the case where the search is successful is (i.e., has the same value as) the input to the step `PutInCart`. We can refer to this value using the ID `isbnLink` of the `sameValues` declaration. I won't show the details here, but will use the convention that any identifier ending in " `Link`," such as `cardTypeLink` or `custLink`, is a reference to a `sameValues` object.

Now in `Process.daml` we replace the declaration of `Condition` with this:

```
<daml:Class rdf:ID="Condition">
   <rdfs:sameClassAs rdf:resource="&drs:Formula"/>
</daml:Class>
```

In `CongoProcess.daml`, we introduce a conditional output of `Specify-PaymentMethod`:

```
<rdf:Class rdf:ID="PaymentOkay">
  <rdfs:subClassOf rdf:resource="&process;#ConditionalOutput"/>
  <rdfs:subClassOf>
     <daml:Restriction>
         <daml:onProperty rdf:resource="&process;#coCondition"/>
         <daml:toClass rdf:resource="#PaymentOkayCondition"/>
     </daml:Restriction>
  </rdfs:subClassOf>
</rdf:Class>
```

We amend the description of Congo class `SpecifyPaymentMethod`:

```
<daml:Class rdf:ID="SpecifyPaymentMethod">
  <rdfs:subClassOf rdf:resource="&process;#AtomicProcess"/>
  The same stuff as before
```

```
  <daml:subClassOf>
     <daml:Restriction>
        <daml:onProperty rdf:resource="&process;output"/>
        <daml:toClass rdf:resource="#PaymentOkayCondition"/>
     </daml:Restriction>
  </daml:subClassOf>
</daml:Class>
```

and finally declare what the condition is:

```
<drs:Forall rdf:ID="PaymentOkayCondition">
   <drs:bound_vars>
      <drs:Var_bag>
         <rdf:_1 ID="cc1" name="cc">
            <drs:type rdf:resource="&cc;CreditCard"/>
         </rdf:_1>
         <rdf:_2 ID="sl2" name="sl">
            <drs:type rdf:resource="&cc;Dollars"/>
         </rdf:_2>
         <rdf:_3 ID="bal3" name="bal">
            <drs:type rdf:resource="&cc;Dollars"/>
         </rdf:_3>
         <rdf:_4 ID="pb4" name="pb">
            <drs:type rdf:resource="&cc;Dollars"/>
         </rdf:_4>
         <rdf:_5 ID="dif" name="dif">
            <drs:type rdf:resource="&cc;Dollars"/>
         </rdf:_5>
      </drs:Var_bag>
   </drs:bound_vars>
   <drs:body rdf:resource="#body1"/>
</drs:Forall>
```

Here I'm assuming `&cc;` is an entity equal to the URL for some credit-card namespace.

The body of the formula is described thus:

```
<drs:And rdf:ID="body1">
   <drs:conn_args>
      <drs:Term_seq>
         <rdf:_1>
            <drs:Atomic_formula>
               <rdf:subject rdf:resource="#cc1"/>
               <rdf:predicate rdf:resource="&cc;creditCard"/>
               <rdf:object>
                  <drs:Term_seq>
                     <rdf:_1 rdf:resource="#cardTypeLink"/>
                     <rdf:_2 rdf:resource="#cardNumLink"/>
```

```
                <rdf:_3 rdf:resource="#cardExpDateLink"/>
             </drs:Term_seq>
          </rdf:object>
       </drs:Atomic_formula>
    </rdf:_1>
    <rdf:_2>
        <drs:Atomic_formula>
           <rdf:subject rdf:resource="#cc1"/>
           <rdf:predicate rdf:resource="&cc;spendingLimit"/>
           <rdf:object rdf:resource="#sl2"/>
        </drs:Atomic_formula>
    </rdf:_2>
    <rdf:_3>
        <drs:Atomic_formula>
           <rdf:subject rdf:resource="#cc1"/>
           <rdf:predicate rdf:resource="&cc;balance"/>
           <rdf:object rdf:resource="#bal3"/>
        </drs:Atomic_formula>
    </rdf:_3>
    <rdf:_4>
       <drs:Atomic_formula>
          <rdf:subject rdf:resource="#sl2"/>
          <rdf:predicate rdf:resource="&cc;difference"/>
          <rdf:object>
             <drs:Term_seq>
                <rdf:_1 rdf:resource="#bal3"/>
                <rdf:_2 rdf:resource="#dif5"/>
             </drs:Term_seq>
          </rdf:object>
       </drs:Atomic_formula>
    </rdf:_4>
    <rdf:_5>
        <drs:Atomic_formula>
           <rdf:subject rdf:resource="#dif5"/>
           <rdf:predicate rdf:resource="&cc;price"/>
           <rdf:object rdf:resource="#isbnLink"/>
        </drs:Atomic_formula>
    </rdf:_5>
    <rdf:_6>
       <Atomic_formula>
          <rdf:subject rdf:resource="#dif5"/>
          <rdf:predicate rdf:resource="&cc;greaterOrEqual"/>
          <rdf:object rdf:resource="#pb4/>
       </Atomic_formula>
    </rdf:_6>
</drs:And>
```