Surface Syntax and Formal Semantics for DAML-S-PAI
* DRAFT 0.1 **
Drew McDermott [[and the DAML-S coalition, if they don't expel me]]
September 7, 2003

1 Goals

This is a proposal for a surface syntax and formal semantics of a "processes as instances" (PAI) notation for DAML-S. The idea is to describe a process as an object, typically consisting of pieces which are themselves processes.

The goals of this exercise are to

1. Provide readable surface syntax ...
2. ... That is easily translatable to RDF;
3. Provide precise answers to questions such as "Does a process step proceed if some of its inputs are unavailable?"

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 an abstract description of something with more infix operators if you like.

2.1 Informal Syntax

The key concept in DAML-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 ...)
```

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))
```

However, once we get into the formal part of this paper, we will ignore type declarations in the interest of clarity.

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

[[In this paper we neglect conditional outputs and effects, although we believe they present no special problems.]]

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

```
(sequence
   (toggle_the_switch)
   (toggle_the_switch))
```

which contains two occurrences of the process `toggle_the_switch`. In keeping with RDF style, the description of a process may accompany one of its occurrences, 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 :name toggle_the_switch)
   (toggle_the_switch))
```

Processes don't have to have names.

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

```
(tag-scope (tag1 tag2)
   (sequence
      (tag tog1 (simple_process :name toggle_the_switch))
      (tag tog2 (toggle_the_switch))))
```

The `tag-scope` construct is necessarily to indicate the scope of the names. One can imagine other ways of doing this (such as having the scope be the innermost iteration).

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:

```
(def_type CC_check_res (Constant "authorized"
                                  "not-authorized"
                                  "timeout"))

(def_type CC_acc_status (Constant "accepted" "not-accepted"))

(simple_process :name check_auth
                :args (cc - Credit_card_data)
                :results (res - CC_check_res))

(sequence :results (final_res - CC_acc_status)
    (check_auth cc <= cc
                 res => (ch1res(↓ check1)))
    (tag check1
       (if_then_else :args (ch1res - CC_check_res)
          :cond (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)
                      :cond (ch2res = "not-authorized")
                      :then (value "not-accepted" => final_res)
                      :else (value "accepted" => final_res))))
          :else
             (if_then_else
                :cond (ch1res = "authorized")
```

```
                    :then (value "accepted" => final_res)
                    :else (value "not-accepted"
                                   => final_res)))))
```

2.2 Formal Syntax

In this section we introduce a somewhat simpler syntax that is easier to explain the semantics of. In the next section we explain how to derive the simple syntax from the user-friendly syntax.

```
<defn> ::= (atomic_process <identifier>)
         | (process <interface-decl> <statement>)

<statement> ::= (call <identifier>)
              | (sequence <statement>*)
              | (parallel <statement>*)
              | (choice <statement>*)
              | (if_then_else
                   :cond <proposition>
                   :then <statement>
                   [:else <statement>])
              | (with_precondition <proposition> <statement>)
              | (with_params [:args (<identifier>*)]
                             [:locals (<identifier>*)]
                             [:results (<identifer>*)]
                   <statement>)
              | (tag_bind (<tag-binding>) <statement>)
              | (with_dataflow <statement> <term> => <channel term>)
```

In a `with_params` statement we will allow the parameter declarations to be moved around, often putting the `:results` after the `<statement>`.

```
<tag-binding> ::= (<identifier> <tag-id>+)

<in-or-out> ::= ↓ | ↑

<term> ::= <channel term> | insert term-language grammar here

<channel term> ::= <channel-spec> | <identifier>

<channel-spec> ::= (<identifier> (<in-or-out> <tag-id>+))
```

```
<tag-id> ::= <identifier> | <natural number>
```

Most of the changes modularize mechanisms such as input/output declarations, input-output flows, and tag bindings.

The first main change is to do away with the `tag` and `tag_scope` constructs and replace them with `tag_bind`. What was written informally as

```
(tag_scope (foo)
   (sequence s_1 (tag foo s_2) s_3))
```

is now considered syntactic sugar for

```
(tag_bind ((foo 2))
   (sequence s_1 s_2 s_3)))
```

The idea is that sequence steps are normally tagged 1, 2, ... Symbolic tags can be introduced by explaining the mapping of the symbols to the built-in tags. For conditionals the built-in tags are `:then` and `:else`. For loops (which we are neglecting complely), the built-in tags are natural numbers, corresponding to iterations.

The `tag_bind` construct can be used to refer to steps that are more deeply nested than in the previous example. For instance, in

```
(tag_bind ((foo :else 2))
   (if_then_else
      :cond C
      :then A
      :else (sequence D E F)))
```

`foo` tags step `E`, the second step of the `:else` step of the construct.

The sugared syntax allowed `:args`, `:locals`, and `:results` to be declared for all constructs. We now provide an explicit `with_params` statement. Similarly for preconditions.

The last major change is that dataflows are specified in only one place, using a `with_dataflow` statement. The destination for the flow can be specified in one of two ways: as an identifier (denoting an input or output channel) bound in the scope of the `with_dataflow` statement, or as a designator for a flow into or out of one of steps of the `<statement>` the `with_dataflow` modifies. The identifier `@` is reserved for this process itself.

For example, the flow from the second occurrence of `check_auth` to statement `check2` would be moved up to the sequence level, and written:

```
(with_dataflow
   (tag_bind ((check2 2))
      (with_params :results (ch2res - CC_acc_status)
         (sequence
            (call :name check_auth)
            (with_params :args (ch2res - CC_check_res)
                         :results (res - CC_acc_status)
               (if_then_else
                  ...)))))
   (cc (↑1)) => (ch2res (↓check2)))
```

(The flow from into the `cc` input of the `call` is left as an exercise for the reader.)

Other, lesser, changes are:

1. To require calls of defined processes to be labeled explicitly using `(call` *proc* `...)`.
2. To require parentheses around `<input flow>`s and `<output flow>`s.

One thing that is absent from the formal treatment is any notion of typing of variables. That absence can't be made up by any kind of sugaring. Eliminating types makes the formal semantics simpler, so from now on forget that variables might have types.

2.3 Informal Syntax as Syntactic Sugar for the Formal Kind

Procedure for "desugaring" the concise notation in order to arrive at its equivalent in the formal syntax:

1. Replace every `tag_scope` with the equivalent `tag_bind`, and replace `(tag` g e`)` with e;
2. If a statement contains an `:args`, `:locals`, or `:results` declaration, tranform it into a `with_params` statement, with the parameter declarations pulled out in the obvious way.
3. Every dataflow expression of the form c`<=`v or v`=>`c is deleted. A `with_dataflow` expression is created and wrapped around all the `tag_bind` expressions that bind a tag referred to in c or v.

[[These rules are not quite right. There are some sugared process expressions that apparently can't be desugared. The problem is that the `tag_bind` mechanism is a bit broken. It doesn't really bind anything, in the sense that you can't refer to the new tags *inside* the scope of the `tag_bind`.]]

3 Formal Semantics

3.1 Technicalities

To explain the formal semantics, we need to talk about what the universe looks like. It certainly includes a set of objects U, and in a full treatment U ought to be broken down into classes corresponding to the types of the variables, but, as explained in section 2.2, we are simplifying variable types away.

In addition to objects, we also need a set of *situations*. Classically these are taken to be snapshots of the world, that is, assignments of truth values to all (timeless) propositions. We adopt that position, with the twist that situations are not defined by their truth assignments; two distinct situations can assign exactly the same values to all propositions. We assume there is a partial order on situations. $\sigma_1 \preceq \sigma_2$ means that σ_1 is "in σ_2's past." Whether there is a unique future for any situation, or time *branches* into the future, is an issue that needn't concern us here.[1]

To handle dataflow, we need a few extra object types. An input or output will be bound to an object of type *channel*. A channel is essentially a storage location. A *channel map* is a set of ordered pairs $\{\langle ch, val\rangle\}$, where $val \in U$. A channel map specifies the current contents of a set of channels. A channel with no value does not appear in the channel map, so *channel map*$(c) = \perp$, the "undefined value." We assume the existence of an infinite number of channels, so we can always find one we haven't used yet.

An *environment* is a mapping from identifiers to values. Values are from the set $U \cup channels$. A special case is an environment that maps arg/result parameters to channels, which we will call a *parameter map*.

The denotation of a DAML-S process expression is (modulo technicalities) a set of pairs $\langle x_0, x_1\rangle$, where x_0 is a "starting point" and x_1 is a "finishing point," the intuition being that one way for the process to proceed starting in x_0 is to wind up in x_1. x_0 and x_1 are *valuated situations*, triples of the form $\langle p, c, s\rangle$, where s is a situation, c is a channel map, and p is a parameter map. s represents the state of the outside world, c represents the contents of channels, and p associates arg/results parameters of the process with channels. Note that c is a "global" channel map, and can mention channels other than those mentioned in p. To refer to the pieces of a valuated situation, we use pm, cm, sit, so that valuated situated

[1] The Reiter School treats situations as sequences of actions (i.e., choices made by an agent). That approach could probably be made to work for us, but for now we'll just be neutral about exactly what situations are metaphysically.

$s = \langle pm(s), cm(s), sit(s) \rangle$.

A pair of valuated situations vs_1, vs_2, where $sit(vs_1) \preceq sit(vs_2)$, is called a *valuated situation interval*, or VSI. So the denotation of a process is (almost) a set of valuated situation intervals, or, equivalently, a relation on valuated situations. $\langle vs_1, vs_2 \rangle$ is in the set if vs_2 is one place the process can finish given that it starts in vs_1. We can refer to the pieces vs_1, vs_2 of a VSI vsi as $begin(vsi), end(vsi)$.[2]

The presence of tags compels us to impose some structure on the VSI a process term denotes. A tag refers to a piece of a process execution trace. So we introduce the notion of a *tag tree*, which records the bindings of tags to valuated situation intervals. Each node of the tree is labeled with a VSI, and has a finite number of subtrees, the links to which are labeled with tags. If m is such a tree, then $piece(m, t_1\, t_2 \ldots t_k)$, where each t_i is a `<tag-id>` is the VSI obtained by getting piece t_1 of m, then piece t_2 of that subtree, and so forth. $piece(m)$ is the VSI associated with the whole tag tree. For clarity, one can also write this as $\texttt{@}m$. The labels below the top node of a tag tree m are denoted by $labels(m)$. If the sequence of tag-ids contains a tag corresponding to no subtree, the value of the *piece* expression is $\perp$. The symbol $\emptyset$ is used for a null tree, associated with no VSIs. The notation $m + t \mapsto m'$ means the tree m augmented with a new branch labeled t that takes you to tag tree m', which thereby becomes a subtree of the new tree. A leaf tag tree whose sole node is labeled v is written $\emptyset + \texttt{@} \mapsto v$. If there is already a subtree or VSI associated with a label l in m, then $m + l \mapsto x$ is m, but with $piece(m, l)$ reset to x.

A key requirement for a useful tag tree is that the VSIs labeling its nodes must be *comparable.* That is, if $\langle s_1, s_2 \rangle$ is a VSI from somewhere in a tag tree and $\langle s_3, s_4 \rangle$ is a VSI from somewhere else, then the four points are ordered by $\preceq$ somehow; that is, there is a permutation i_1, i_2, i_3, i_4 of $1, 2, 3, 4$ so that $sit(s_{i_1}) \preceq sit(s_{i_2}) \preceq sit(s_{i_3}) \preceq sit(s_{i_4})$. A set of tag trees may describe alternative timelines (or time branches, or possible worlds, $\ldots$), but each tag tree is drawn from a single timeline.

Another important requirement is that a tag tree be *inertial*, which means "internally" and "externally" inertial.

A set of valuated situation intervals is *internally inertial* if it has the following two properties:

1. *Parsimony:* No channel appears in a channel map unless some param-

[2] One way to think about nondeterminism would be to allow a process to end in more than one valuated situation. But there other ways to think about branching time, and we don't want to commit to a particular scheme.

eter is or was bound to it at some point.

2. *Conservation:* Nothing changes the contents of channels except dataflows to the input parameters denoting them.

Both properties are expressed as constraints on consecutive VSIs, defined thus: vsi_1 and vsi_2 are *consecutive* in a set of VSIs S if and only if

$vsi_1 \in S$ and $vsi_2 \in S$
and $sit(end(vsi_1)) \prec sit(begin(vsi_2))$
and there is no $vsi' \in S$ such that
$$sit(end(vsi_1)) \prec sit(end(vsi')) \prec sit(begin(vsi_2))$$

One might wonder about the possibility of there being an infinite sequence of situations between vsi_1 and vsi_2. In this document, we consider only finite tag trees, so the issue won't come up, but when the formalism is extended to iterations, we will certainly need an "anti-Zeno" rule to prevent such pathological cases from arising.

Parsimony is then formalized as follows: If c is a channel in $cm(begin(v))$ for some VSI v in our set, then either c appears in $cm(end(v'))$ for some v' such that v' and v are consecutive, or there is a parameter p such that $(pm(begin(v)))(p) = c$.

Conservation is expressed thus: If a set of VSIs contains two consecutive elements vsi_1 and vsi_2, then $cm(begin(vsi_2))$ differs from $cm(end(vsi_1))$ only on channels denoted by input parameters to vsi_2. More technically,

If vsi_1 and vsi_2 are consecutive
then if there is a channel c such that
$$(cm(end(vsi_1))(c) \neq (cm(begin(vsi_2))(c)$$
then there is a parameter p such that $(pm(begin(vsi_2)))(p) = c$

A set of VSIs is *externally inertial* if, whenever two of its elements, vsi_1 and vsi_2, are consecutive, then the only changes that occur between $sit(end(vsi_1))$ and $sit(begin(vsi_2))$ are those allowed by the "physics" of the world. In the present paper we will say nothing about physics, except to note that if the world obeyed the laws of "classical planning" [¡bibref¿] then *no* changes would occur between the end of vsi_1 and the beginning of vsi_2, and the two situations would assign the same truth values to all propositions. Of course, the world of web services is definitely nonclassical, and the physics could get very intricate, what with autonomous processes, randomness, and the presence of other agents, some more cooperative than others.

Having defined all these concepts in terms of sets of VSIs, we extend them to a tag tree by applying them to the set of all VSIs labeling nodes of the tree.

The formal semantics makes use in a few places of sets of ordered pairs, a.k.a functions, so we introduce some common terminology to talk about them. Typically our functions are finite, hence partial. So $F(x)$ is either the y corresponding to x in the list of ordered pairs, or $\perp$ if there's no such pair. We often want to avoid $\perp$, because it indicates that something meaningless or impossible has occurred; $F(\perp) = \perp$, so $\perp$ will propagate out to make a given syntactically correct DAML-S expression meaningless. If F is a function, then $F \setminus\!+ F'$ means $F \setminus \mathit{overlap}(F, F') \cup F'$, where $\mathit{overlap}(F, F')$ is the set of elements of F whose arguments are in the domain of F'. In other words, $F \setminus\!+ F'$ is $F \cup F'$ with bindings from F' replacing those in F that give different results for a given argument: $(F \setminus\!+ F')(x) = \; \mathit{if} \; x \in \mathit{dom}(F') \; \mathit{then} \; F'(x) \; \mathit{else} \; F(x)$.

There one more gimmick we need here for technical reasons. We use the symbol "$\diamond$" to refer to the "nowhere channel," a mathematical `/dev/null`. It's not an error to write to this channel, but it is an error to try to read from it.

3.2 The Semantics

We are finally ready to proceed. We will define the function $\mathcal{M}(e, m, \mathit{env})$, which gives the meaning of expression e with respect to tag tree m and environment *env*. Because some expressions denote channels, every expression can be thought of as having a c-value and a d-value, which correspond to the "l-value" and "r-value" from programming-language semantics. So $\mathcal{M}(e, m, \mathit{env}) = \langle \mathcal{C}(e, m, \mathit{env}), \mathcal{D}(e, m, \mathit{env}) \rangle$. $\mathcal{D}$ gives the *d-value* (denotation) of an expression, and $\mathcal{C}$ gives its *c-value*, the channel, if any, that the d-value came from. For many expressions, such as `5` and `x+y`, the c-value is undefined, so it will be $\perp$ or $\diamond$, as appropriate. In cases where it is $\perp$, we will give the value of $\mathcal{D}$ for an expression and not bother with spelling out $\mathcal{C}$ explicitly. Our main focus in this paper is the denotations of process expressions *exp*, so what we are ultimately interested in is the set of valuated situation intervals

$$\mathit{image}(R, \mathcal{D}(\mathit{exp}, \emptyset, E_0))$$

where E_0 is the built-in or "basis" environment; $\mathit{image}(f, S)$ means $\{f(x) | x \in S\}$; and $R(m) = \texttt{@}m$, the VSI labeling the root node of m.

A full treatment of the semantics would specify the meanings of all expressions, including the long-awaited formalism for propositions (conditions and effects). But we're going to focus here on the process sublanguage, so the long-awaited formalism will have to be awaited a little longer. [[Actually, I expect no substantive problems to arise in extending $\mathcal{M}$ to a logical language. In particular, it should be possible to refer to the entities, such as channels, bound in a process, simply by using their names.]]

We start by giving the semantics of channel terms.

$$
\begin{aligned}
&\mathcal{M}(id(\downarrow t_1 \dots t_k), m, env) = H(begin(piece(m, t_1 \dots t_k)), id)\\
&\mathcal{M}(id(\uparrow t_1 \dots t_k), m, env) = H(end(piece(m, t_1 \dots t_k)), id)\\
&\text{where } H(vs, id) = \text{ If } vs = \perp \text{ then } \langle \diamond, \perp \rangle\\
&\qquad\qquad \text{else Let } c = (pm(vs))(id)\\
&\qquad\qquad\qquad \text{in if } c \text{ is a channel then } \langle c, (cm(vs))(c) \rangle\\
&\qquad\qquad\qquad\qquad \text{else } \langle \diamond, \perp \rangle
\end{aligned}
$$

If $env(id)$ is a channel c, $\mathcal{M}(id, m, env) = \langle c, (cm(end(\texttt{@}m)))(c) \rangle$

For all other e,

$$\mathcal{C}(e, m, env) = \perp$$

so that we can define $\mathcal{M}(e, m, env)$ just by specifying $\mathcal{D}(e, m, env)$.

We construct an interpretation of a set of composite processes in the usual way. We start with an assignment of a set of VSIs to each atomic-process name. We'll just assume that E_0, the initial environment, includes these assignments.

The definition of $\mathcal{D}$ is a pile of special cases, one for each sort of composite process.

If a is a named process (especially an atomic one), then
$\mathcal{D}(\texttt{(call } a\texttt{)}, m_0, env) = env(a) = \{\emptyset + \texttt{@} \mapsto v | v \in env(a)\}$

$$
\begin{aligned}
&\mathcal{D}(\texttt{(sequence } p_1\ p_2\ \dots\ p_n\texttt{)},\ m_0,\ env)\\
&\quad = \{\text{tag tree } m | \{1, \dots, n\} \subseteq labels(m) \text{ and}\\
&\qquad \text{for some } v_1, \dots, v_n, \text{ where } v_i \in \mathcal{D}(p_i, m_0, env)\\
&\qquad\quad \text{for all } j, 1 \leq j \leq n, piece(m, j) = v_j\\
&\qquad\quad \text{and for all } j, 1 \leq j < n, sit(end(v_j)) \preceq sit(begin(v_{j+1}))\\
&\qquad\quad \text{and } v_1, \dots, v_n \text{ are an inertial sequence}\\
&\qquad\quad \text{and } \texttt{@}m = \langle begin(v_1), end(v_n) \rangle\}
\end{aligned}
$$

Note that we don't have to mention comparability of the v_i because it's implied by the requirement that they are linearly ordered.

$$
\begin{aligned}
&\mathcal{D}(\texttt{(parallel } s_1 \ldots s_k\texttt{)}, m_0, \mathit{env}) \\
&\quad = \text{Let } M = \{\emptyset + (1 \mapsto v_1) + \ldots + (k \mapsto v_k) \\
&\qquad\qquad \mid \text{some } v_1 \in \mathcal{D}(s_1, m_0, \mathit{env}), \ldots, v_k \in \mathcal{D}(s_k, m_0, \mathit{env})\} \\
&\qquad\qquad\quad \text{such that the } v_i \text{ are comparable and inertial}\} \\
&\qquad \text{in } \mathit{image}(\mathit{combine}, M) \\
&\qquad \text{where } \mathit{combine}(m) \\
&\qquad\qquad = \{m + \texttt{@} \mapsto \langle v_b, v_e\rangle \\
&\qquad\qquad\quad \mid \text{Let } S_b = \{v \mid v = \mathit{begin}(\mathit{piece}(m, i)) \\
&\qquad\qquad\qquad\qquad \text{for some } i \in [1, k]\} \\
&\qquad\qquad\quad \text{and } S_e = \{v \mid v = \mathit{end}(\mathit{piece}(m, i)) \\
&\qquad\qquad\qquad\qquad \text{for some } i \in [1, k]\} \\
&\qquad\qquad\quad \text{in } v_b = \mathit{earliest}(S_b) \text{ and } v_e = \mathit{latest}(S_e)\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{D}(\texttt{(choice } c_1, \ldots, c_k\texttt{)}, m, \mathit{env}) \\
&\quad = \cup_{j=1}^{k} \mathcal{D}(c_j, m, \mathit{env})
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{D}(\texttt{(if\_then\_else :cond } t \texttt{ :then } Y \texttt{ :else } N\texttt{)}, m_0, \mathit{env}) \\
&\quad = C(\texttt{true}, Y, \texttt{iftrue}) \cup C(\texttt{false}, N, \texttt{iffalse}) \\
&\qquad \text{where } C(\tau, e, l) = \{\emptyset + (\texttt{@} \mapsto m) + (l \mapsto m) \\
&\qquad\qquad \mid \text{for some } m \in \mathcal{D}(e, m_0, \mathit{env}) \\
&\qquad\qquad\quad \text{such that } \mathcal{D}(t, m, \mathit{env}) = \tau\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{D}(\texttt{(with\_precondition } p \; s\texttt{)}, m_0, \mathit{env}) \\
&\quad = \{m \in \mathcal{D}(p, m_0, \mathit{env}) \mid p \text{ is true in } \mathit{sit}(\mathit{begin}(\texttt{@}m))\}
\end{aligned}
$$

That concludes our definitions of the basic control constructs. Now we handle the various notations for binding parameters and passing values to them.

$$
\begin{aligned}
&\mathcal{D}(\texttt{(with\_params } q \texttt{ :args } A \texttt{ :locals } V \texttt{ :results } R\texttt{)}, m_0, \mathit{env}) \\
&\quad = \text{Let } E_A = \{\langle a, c\rangle \mid a \in A \textit{ and } c \textit{ a new channel}\} \\
&\qquad\quad E_V = \{\langle v, c\rangle \mid v \in V \textit{ and } c \textit{ a new channel}\} \\
&\qquad\quad E_R = \{\langle r, c\rangle \mid r \in R \textit{ and } c \textit{ a new channel}\} \\
&\quad \text{then let } M = \mathcal{D}(p, m_0, \mathit{env} \mathbin{\backslash\!+} E_P \mathbin{\backslash\!+} E_V \mathbin{\backslash\!+} E_R) \\
&\qquad \text{in } \mathit{image}(A, M) \\
&\qquad\quad \text{where for all } m \in M \\
&\qquad\qquad A(m) = m + \texttt{@} \mapsto \langle vs_b, vs_e\rangle \\
&\qquad\qquad\quad \text{where let } \langle vs'_b, vs'_e\rangle = \texttt{@}m \\
&\qquad\qquad\qquad \text{in } vs_b = \langle pm(vs'_b) \mathbin{\backslash\!+} E_P, cm(vs'_b), \mathit{sit}(vs'_b)\rangle
\end{aligned}
$$

and $vs_e = \langle pm(vs'_e) \setminus\!\!+ E_R, cm(vs'_e), sit(vs'_e)\rangle$

$$
\begin{aligned}
&\mathcal{D}(\texttt{(with\_dataflow}\ p\ r\texttt{=>}c\texttt{)}, m_0, env)\\
&\qquad = \{m \in \mathcal{D}(p, m_0, env)\\
&\qquad\quad \mid \mathcal{M}(c, m, env) = \langle x, v_c\rangle, x \neq \perp,\ \text{and}\ \mathcal{D}(r, m, env) = v_r \neq \perp\\
&\qquad\quad\ \ \text{and}\ v_r = v_c\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{D}(\texttt{(tag\_bind ((}id\ tag\texttt{))}\ \ P\texttt{)}, m_0, env)\\
&\qquad = image(f, \mathcal{D}(P, m_0, env))\\
&\qquad \text{where}\ f(m) = m + id \mapsto piece(m, tag)
\end{aligned}
$$

3.3 Some clarifying remarks

In case the meaning of all this isn't intuitively obvious, let us make a few remarks.

A key feature of the the `with_params` construct is that it binds channel identifiers in two places: the environment, and the parameter maps for the process. That allows us to refer to them in two different ways. Within the process, we can refer to them by name, because they're bound in the environment. After leaving the process, but within the scope of a `with_dataflow` wrapping it, we can refer to the same channels by their associations with the parameter maps for the process's steps. E.g., in

```
(with_params
   :args (w)
   (with_dataflow
       (with_dataflow
           (sequence (call foo)
                     (with_params
                         :args (x)
                         (call baz)
                         :results (y))
                     (with_params
                         :args (z)
                         (call shazam)))
           w => (x(⇂2)))
       (y(↿2)) => (z(⇂3))))
```

we send the d-value of `w` to the channel `x` of (i.e., bound around the call to) `baz`, and we send the contents of the channel `y` of `baz` to the input channel `z` of `shazam`. The d-value of `y(↿2)` is obtained in two stages: letting *evs* be *end* valuated situation of step 2, we first retrieve the channel for `y` from the parameter map of *evs*, then we retrieve the contents of that channel from the channel map of *evs*. Similarly, letting *bvs* be the *begin* valuated situation of step 3, we get the `z` channel from the parameter map of *bvs* and we store the new association in the channel map of *bvs*

Actually, it may not be clear how anything gets "stored." The definition of *with_dataflow* simply requires that the d-value of one expression be found in the c-value of another, as if by magic. What actually happens is that the semantics assigns a large number of possible denotations to a process term. In some of them, the c-value (a channel) happens to contain the correct value, and the others are weeded out.

Nothing in the example tells us how the channel for `y` gets set or what to; that presumably occurs inside step 2. If it never does get set, then the channel map will give us the value $\perp$ for the channel. According to the definition of `with_dataflow`, a tag map with this feature will not pass muster.

Another tricky bit is that we have to make sure we always use the right channel map to retrieve parameter values from. Typically it's the latest one available. Internal inertiality guarantees that channel contents accumulate as time passes, except for dataflows.

4 Relationship to "Deep" Syntax (RDF)

The surface syntax we have been working with was designed with an eye toward transformation into RDF. A process would be described as an object with various properties. E.g. `(if_then_else :cond` c `:then` a `:else`b`)` would get turned into

```
<if_then_else>
   <cond>
     Representation of c
   </cond>
   <then>
      Representation of a
   </then>
   <else>
      Representation of b
   </else>
</if_then_else>
```

`:name` declarations would get turned into `ID` attributes. Parameter and precondition declarations would be handled as subelements of the process elements they belong to.

The representation of dataflows is closer to the formal syntax than the informal version. That is, the dataflow is itself an object with properties (`from-expression`, `to-param`, `to-step`), which would be a subelement of the appropriate process element. That is, `(with_dataflow (sequence...)` e`=> (`p`(` $\downarrow$ s`)))` would come out as

```
<sequence>
   ... steps ...
   <dataflow>
      <value>
          Representation of e
      </value>
      <to-step direction="arg" param="p">
          Representation of s
      </to-step>
   </dataflow>
</sequence>
```

5 Comments, conclusions, future directions

I have a nagging feeling that I've produced something that's too close to an ordinary programming language, and lost some of the original DAML-S aim of making simple/atomic processes intrinsically refer to communication with outside agents. In this version, if you want to send or receive a message, you just call a primitive action that sends it or receives it. The notion of "channel" involves purely internal communication channels. To handle true interactions with other agents, we need to introduce a type *port* that sends and receives values to the outside world.

Part of the problem is that DAML-S has never been quite clear on the distinction between ports and channels. That's why there have been so many debates on questions like: Who is responsible for supplying inputs to process steps?

A related perennial question is, Who is responsible for making sure a precondition of a step is true? In this version, it should be obvious that the process interpreter, or some more sophisticated planner, is responsible.

We said in section 1 that one of our goals was to produce crisp answers to questions such as "Does a process step proceed if some of its inputs are unavailable?" The answer is "Yes, but No." Theoretically a process interpreter is allowed to be clairvoyant. In executing a statement (`with_dataflow` p v`=>`c), if it guesses the value of v correctly then it can put it in c even before it finds out the actual value. Here's an example:

```
(tag_scope (produce consume)
   (parallel
      (sequence (gen1)
                (tag produce (gen2 :results (yummie))))
      (sequence (tag consume
                     (eat1 goodie <= (yummie(↑produce))))
                (eat2))))
```

If the interpreter gets to the step `consume` before the step `produce`, it is allowed to predict the value of `yummie` and proceed. If this value is a boolean, and in most situations it's `true` 999 times out of a 1000, the interpreter might just pass `true` as the `goodie` argument to `eat1`.

However, if an interpreter never resorts to such sophisticated maneuvers, then what it will do when it encounters a step with an unavailable argument is wait for it to become available. Deadlocks can arise easily, simply by making two steps produce inputs for each other.

Another interesting question: "Can a channel be set more than once?" The answer is No. The reason, as explained in section 3.3, is that channels never actually get set. Situation sequences in which a channel has the wrong value just get weeded out. If you discard all the scenarios in which the channel doesn't have value x_1, and all those in which it doesn't have value $x_2 \neq x_1$, you will have discarded all scenarios, and be left with a meaningless process.

Here's a little exercise for your (by now well honed) intuitions: What does this expression denote:

```
(tag_scope cycle
   (sequence
      (tag cycle
         (with_params :args (x) :results (y)
            (no-op)
            x+1 => y
            x <= y))))
```

Answer: Nothing, for a variety of reasons. One thing it does *not* denote is an infinite loop that keeps making `x` and `y` bigger.

There is a lot left to do. We haven't discussed conditional effects and values. They should not be hard to incorporate.

DAML-S actually has no `parallel` construct; instead it has `split`, `unordered`, and `split+join`. The differences between these are obscure, so a vanilla `parallel` was put in their place. Someone who understands the semantics of the original constructs should have a go at defining them formally.

After section 2.1 we neglect defining and naming new processes. One reason is that we haven't given any thought to the semantics of recursive processes. Presumably we just need to wrap a big fixed-point operator around everything, but the matter needs a bit of thought.

Iteration was left out of consideration. The only reason for this omission is that it would require generalizing channels a bit. A loop requires the idea of an *accumulator*. 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 channels that contain a history list of the values accumulated to date. That way, we can use the same tricks as before to avoid side effects on channels.

If we actually want side effects on channels, the best way to do it is probably to introduce an artificial class of "assignment events" that bring the side effect about. We have to decide exactly at what time such an event occurs, and the answer is probably "Right before each step that consumes a channel's contents."