Concurrent Semantics for the Web Services Specification Language DAML-S

Anupriya Ankolekar[1], Frank Huch[2], and Katia Sycara[1]

[1] Carnegie Mellon University, Pittsburgh PA 15213, USA
{anupriya,katia}@cs.cmu.edu,
[2] Christian-Albrechts-University of Kiel, 24118 Kiel, Germany
fhu@informatik.uni-kiel.de

Abstract. The DARPA Agent Markup Language ontology for Services (DAML-S) enables the description of Web-based services, such that they can be discovered, accessed and composed dynamically by intelligent software agents and other Web services, thereby facilitating the coordination between distributed, heterogeneous systems on the Web. We propose a formalised syntax and an initial reference semantics for DAML-S.

Keywords: DAML-S, Web services, concurrent semantics, agents

1 Introduction

The DARPA Agent Markup Language Services ontology (DAML-S) is being developed for the specification of Web services, such that they can be dynamically discovered, invoked and composed with the help of existing Web services. DAML-S, defined through DAML+OIL, an ontology definition language with additional semantic inferencing capabilities, provides a number of constructs or DAML+OIL classes to describe the properties and capabilities of Web services. DAML-S will be used by Web service providers to markup their offerings, by service requester agents to describe the desired services, as well as by planning agents to compose complex new services from existing simpler services. Other approaches to the specification of Web services from the industry are UDDI, WSDL, WSFL and XLANG, which address different aspects of Web service description provided by DAML-S. Furthermore, DAML-S is unique in that, due to its foundations in DAML+OIL, it provides markup that can be semantically meaningful for intelligent agents.

Although DAML-S is intended primarily for Web-based services, the basic framework can be extended to facilitate the coordination and interoperability, more generally, of systems in heterogeneous, dynamic environments. In this paper, we propose an interleaving, strict operational semantics for DAML-S[1] informally described in [1].

[1] DAML-S is currently under development and the language described here is the DAML-S Draft Release 0.5 (May 2001).

F. Arbab and C. Talcott (Eds.): COORDINATION 2002, LNCS 2315, pp. 14–21, 2002.

The development of a reference semantics for DAML-S brings any ambiguities about the language specification to the fore so that they can be addressed and resolved. It can also provide the basis for the future DAML-S execution model. Furthermore, having a formal semantics is the first step towards developing techniques for automated verification of functional and non-functional properties of the DAML-S execution model. The formalisation of other Web standards would make it easier to compare and contrast their capabilities and better understand the strengths and weaknesses of various approaches.

In this paper, we model a core subset of DAML-S, referred to as *DAML-S Core*. Every service defined in DAML-S can be transformed into a functionally equivalent service definition in DAML-S Core stripped of additional attributes that aid in service discovery or any quality-of-service parameters. The next section, Section 2, presents the DAML-S ontologies and the process model of a service. Section 3 discusses some of the issues involved in developing a formal model for DAML-S and presents the syntax of DAML-S Core. Finally, a formal semantics for DAML-S Core is given in Section 4.

2 The DAML-S Ontology

The DAML-S ontology consists of three parts: a *service profile*, a *process model* and a *service grounding*. The service profile of a particular Web service would enable a service-requesting agent to determine whether the service meets its requirements. The profile is essentially a summary of the service, specifying the input expected, the output returned, the precondition to and the effect of its successful execution. The process model of a service describes the internal structure of a service in terms of its subprocesses and their execution flow. It provides a detailed specification of how another agent can interact with the service. Each process within the process model could itself be a service, in which case, the enclosing service is referred to as a *complex* service, built up from simpler, atomic services. The service grounding describes how the service can be accessed, in particular which communication protocols the service understands, which ports can receive which messages and so forth.

In this paper, we will only be considering the service process model, since it primarily determines the semantics of the service's execution. The formalisation proposed here will however form the basis for an execution model and provide inputs for the definition of the service grounding. The inputs, outputs and effects of a process can be instances of any class in DAML+OIL. The preconditions are instances of class `Condition`. There are a number of additional constructs to specify the control flow within a process model: `Sequence`, `Split`, `Split+Join`, `If-Then-Else`, `Repeat-While`, `Repeat-Until`. The execution of a service requires communication, interaction between the participants in a service transaction. The DAML-S constructs for communication will be described in the future service grounding. Since modelling the communication within a service transaction is essential to describing the execution semantics of a service described in DAML-S, we define a set of what we consider basic communication primitives, for example, for the sending and receiving of messages.

3 Modelling DAML-S Core

The DAML-S class `Process` and its subclasses, representing agents, are modelled as functions. DAML-S agents essentially take in inputs and return outputs, exhibiting simple function-like behaviour. A Web page, for example, is an extremely simple agent which has no input and as output, merely some HTML content. The input to a `Process` is not restricted and could be a `Process` itself, resulting in a '*higher-order*' agent, offering meta-level functionality. A simple example of a higher-order service is an agent that, when given a task and an environment of existing services, locates a service to perform the task, invokes the service and returns the result. The functionality of the agent thus depends on the set of services in the world that it takes as input.

Furthermore, agents can be composed together. This composition itself represents an agent with its own inputs and outputs. The composition could be sequential, dependent on a conditional or defined as a loop. The composition could also be concurrent, where the agents can interact with each other, representing relatively complex, distributed applications, such as chat systems.

DAML-S classes are defined through DAML+OIL, an ontology definition language. DAML+OIL, owing to its foundations in RDF Schema, provides a typing mechanism for Web resources [8] [9], such as Web pages, people, document types and abstract concepts. The difference between a DAML+OIL class and a class in a typical object-oriented programming language is that DAML+OIL classes are meant primarily for data modelling and contain no methods. Subclass relationships are defined through the property `rdfs:subClassOf`. We model classes in DAML-S as type expressions and subclasses as subtypes. Modelling other properties with arbitrary semantics does not significantly affect the type system or the functional behaviour of DAML-S agents and are therefore not considered further. More formally,

Definition 1 (Type Expressions). *A* type expression $\tau \in \mathcal{T}$ *is either a type variable* $\alpha \in \mathcal{V}$ *or the application,* $(T\tau_1 \cdots \tau_n)$*, of an n-ary type constructor* $T \in \mathcal{F}$ *to the type expressions* $\tau_1, \ldots, \tau_n$*.*

Type constructors in $\mathcal{F}$ are determined by DAML-S Core classes, such as `List`, `Book` and `Process`. In addition to these, DAML-S Core has a predefined functional type constructor $\rightarrow$, for which, following convention, we will use the infix notation. All type constructors bind to the right, i.e. $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ is read as $(\tau_1 \rightarrow (\tau_2 -> \tau_3))$.

Type expressions build the term algebra $T_{\mathcal{F}}(\mathcal{V})$. DAML-S agents can be polymorphic with respect to their input and output, for example an agent which duplicates input of arbitrary type. Polymorphic types are type expressions containing type variables. The expression `a` $\rightarrow$ `b`, for instance, is a polymorphic type with type variables `a` and `b`, which can be instantiated with concrete types. The substitution [`a`/`integer`, `b`/`boolean`] applied to `a` $\rightarrow$ `b` results in the type `integer` $\rightarrow$ `boolean`. Identical type variables in a type expression indicate identical types. For the formalisation of polymorphism, we use *type schemas*, in which all free type variables are bound: $\forall\alpha_1, \ldots, \alpha_n.\tau$, where τ is a type and $\alpha_1, \ldots, \alpha_n$ are the generic variables in τ.

Although DAML-S Core agents can be functionally simple, they derive much of their useful behaviour from their ability to execute concurrently and interact with one another. The communication an agent is engaged in is a side-effect of its functional execution. Communication side-effects can be incorporated into the functional description of agents with the help of the `IO` monad. Monads were introduced from category theory to describe programming language computations, actions with side-effects, as opposed to purely functional evaluations. The `IO` monad, introduced in Concurrent Haskell [7], describes actions with communication side-effects.

The `IO` monad is essentially a triple, consisting of a unary type constructor `IO` and two functions, `return` and `(>>=)`. A value of type `IO a` is an I/O action, that when performed, can engage in some communication before resulting in a value of type `a`. The application `return` v represents an agent that performs no IO and simply returns the value v. The function `(>>=)` represents the sequential composition of two agents. Thus, `action1 >>= action2` represents an agent that first performs `action1` and then `action2`. Consider the type of `(>>=)`: $\forall$`a,b.IO a` $\rightarrow$ `(a` $\rightarrow$ `IO b)` $\rightarrow$ `IO b`. First, an action of type `IO a` is performed. The result of this becomes input for the second action of type `a` $\rightarrow$ `IO b`. The subsequent execution of this action results in a final value of type `IO b`. The expression on the right-hand side of `(>>=)` must necessarily be a unary function that takes an argument of type `a` and returns an action of type `IO b`.

Although the communication an agent is engaged in can be expressed with the `IO` monad, we still need to describe the means through which communication between multiple agents takes place. We model communication between agents with *ports* [6], a buffer in which messages can be inserted at one end and retrieved sequentially at the other. In contrast to the *channel* mechanism of Concurrent Haskell, only one agent can read from a port, although several agents can write to it. The agent that can read from a port is considered to own the port. Since we need to be able to type messages that are passed through ports, each agent is modelled as having multiple ports of several different types. This conceptualisation of ports is also close to the UNIX port concept and is therefore a natural model for communication between distributed Web applications. Agents and services are modelled as communicating asynchronously. Due to the unreliable nature of the Web, distributed applications for the Web are often designed to communicate asynchronously.

Definition 2 (DAML-S Core Expressions). *Let Var^τ denote the set of variables of type τ. The set of* DAML-S Core expressions *over Σ, $Exp(\Sigma)$, is defined in Table 1. The set of expressions of type τ is denoted by $Exp(\Sigma)^\tau$.*

Definition 3 (DAML-S Core Agents). *Let $x_i \in Var^{\tau_i}, x_i$ pairwise different and $e \in Exp(\Sigma)^\tau$. A* DAML-S service definition *then has the following form*

$$s \;\; x_1 \cdots x_n \texttt{:=} \;\; e$$

$s \in \mathcal{S}$ is said to have type $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$. $\mathcal{S}$ denotes the set of services.

In the definition of $Exp(\Sigma)$ in Table 1, we use partial application and the curried form of function application. For a function that takes two arguments, we use the curried type $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ instead of $(\tau_1, \tau_2) \rightarrow \tau_3$.

Table 1. DAML-S Core Expressions

Σ	$\Sigma \subseteq Exp(\Sigma)$	
var	$Var^\tau \subseteq Exp(\Sigma)^\tau$	
abs	$\backslash x \texttt{ -> } e \in Exp(\Sigma)^{\tau_1 \to \tau_2}$	for $x \in Var^{\tau_1}$, $e \in Exp(\Sigma)^{\tau_2}$
appl	$(e_1\ e_2) \in Exp(\Sigma)^{\tau_2}$	for $e_1 \in Exp(\Sigma)^{\tau_1 \to \tau_2}$, $e_2 \in Exp(\Sigma)^{\tau_1}$
cond	$\texttt{cond}\ e\ e_1\ e_2 \in Exp(\Sigma)^{\texttt{IO}\ \tau}$	for $e \in Exp(\Sigma)^{\texttt{boolean}}$, $e_1, e_2 \in Exp(\Sigma)^{\texttt{IO}\ \tau}$
return	$\texttt{return}\ e \in Exp(\Sigma)^{\texttt{IO}\ \tau}$	for $e \in Exp(\Sigma)^\tau$
seq	$e_1 \texttt{ >>= } e_2 \in Exp(\Sigma)^{\texttt{IO}\ \tau_2}$	for $e_1 \in Exp(\Sigma)^{\texttt{IO}\ \tau_1}$, $e_2 \in Exp(\Sigma)^{\tau_1 \to \texttt{IO}\ \tau_2}$
send	$e_1 \texttt{!} e_2 \in Exp(\Sigma)^{\texttt{IO ()}}$	for $e_1 \in Exp(\Sigma)^{\texttt{Port}\ \tau}$, $e_2 \in Exp(\Sigma)^\tau$
rec	$e\texttt{?} \in Exp(\Sigma)^{\texttt{IO}\ \tau}$	for $e \in Exp(\Sigma)^{\texttt{Port}\ \tau}$
port	$\texttt{newPort}\tau \in Exp(\Sigma)^{\texttt{IO Port}\ \tau}$	for $\tau \in \mathcal{T}$
spawn	$\texttt{spawn}\ e \in Exp(\Sigma)^{\texttt{IO}\ \tau \to \texttt{IO ()}}$	for $e \in Exp(\Sigma)^{\texttt{IO}\ \tau}$
choice	$\texttt{choice}\ e_1\ e_2 \in Exp(\Sigma)^{\texttt{IO}\ \tau}$	for $e_1, e_2 \in Exp(\Sigma)^{\texttt{IO}\ \tau}$
serv	$s\ e_1 \cdots e_n \in Exp(\Sigma)^\tau$	for $e_i \in Exp(\Sigma)^{\tau_i}$, $s \in \mathcal{S}^{\tau_1 \to \cdots \to \tau_n \to \tau}$

Port references are constructed with a unary type constructor $\texttt{Port} \in \mathcal{F}$. A send operation takes as argument a destination port and a message and sends the message to the port, resulting in an I/O action that returns no value. Similarly, a receive operation takes as argument a port on which it is expecting a message and returns the first message received on the port. It thus performs an I/O action and returns a message. To be well-typed, the type of the message and the port must match. The `spawn` service takes an expression, an I/O action, as argument and spawns a new agent to evaluate the expression, which may not contain any free variables. The `choice` operator takes two I/O actions as arguments, makes a non-deterministic choice between the two and returns it as the result. For the application of choice to be well-typed, both its arguments must have the same type, since either one of them could be returned as the result.

4 Semantics of DAML-S

The semantics of DAML-S has been informally described in [1]. In this section, we describe a formal operational semantics of Core DAML-S. Our semantics is based on the operational semantics for Erlang [2] and Concurrent Haskell [7] programs, inspired by the structural operational semantics of CCS and the π-calculus.

In a Σ-Interpretation $\mathcal{A} = (A, \alpha)$, A is a T-sorted set of concrete values and α an interpretation function that maps each symbol in Ω, the set of all

constructors defined through DAML+OIL, to a function over A. In particular, A includes functional values, i.e. functions.

Definition 4 (State). *A* state *of execution within DAML-S Core is defined as a finite set of agents: State* $:= \mathcal{P}_{fin}(Agent)$

An agent *is a pair* (e, φ)*, where* $e \in Exp(\Sigma)$ *is the DAML-S Core expression being evaluated and* φ *is a partial function, mapping port references onto actual ports:*

$$Agent := Exp(\Sigma) \times \{\varphi \mid \varphi : \texttt{PortRef} \rightharpoonup \texttt{Port}_\tau^{\mathfrak{A}}\}$$

for all τ*, where* $\texttt{Port}_\tau^{\mathfrak{A}} := (A^\tau)^*$ *and* `PortRef` *is an infinite set of globally known unique port references, disjoint with* A*. Since no two agents can have a common port, the domains of their port functions* φ *are also disjoint.*

Definition 5 (Evaluation Context). *The set of* evaluation contexts $\mathcal{EC}$ *[10] for DAML-S Core is defined by the context-free grammar*

$$E := [\,] \mid \phi(v_1, \ldots, v_i, E, e_{i+2}, e_n) \mid (E\ e) \mid (v\ E) \mid E \texttt{>>=} e$$

for $v \in A$*,* $e, e_1, e_2 \in Exp(\Sigma)$*,* $\phi \in \Omega \cup \mathcal{S} \backslash \{\texttt{spawn}, \texttt{choice}\}$*.*

Definition 6 (Operational Semantics). *The* operational semantics *of DAML-S is* $\longrightarrow \subset \text{State} \times \text{State}$ *is defined in Tables 2 and 3. For* $(s, s') \in \longrightarrow$*, we write* $s \longrightarrow s'$*, denoting that state* s *can transition into state* s'*.*

The application of a defined service is essentially the same as the application rule, except that the arguments to s must be evaluated to values, before they can be substituted into e. In a [SEQ], if the left-hand side of `>>=` returns a value v, then v is fed as argument to the expression e on the right-hand side. That is, the output of the left-hand side of `>>=` is input to e.

Evaluating `spawn` e results in a new parallel agent being created, which evaluates e and has no ports, thus φ is empty. Creating a new port with port descriptor p involves extending the domain of φ with p and setting its initial value to be the empty word ϵ. The port descriptor p is returned to the creating agent.

Table 2. Semantics of DAML-S Core - I

(FUNC)	$\dfrac{\phi \in \Omega}{\Pi, (E[\phi v_1 \cdots v_n], \varphi) \longrightarrow \Pi, (E[\phi_{\mathfrak{A}} v_1 \cdots v_n], \varphi)}$
(APPL)	$\dfrac{\text{free}(u) \cap \text{bound}(e) = \emptyset}{\Pi, (E[(\backslash x \texttt{ -> } e)\ u)], \varphi) \longrightarrow \Pi, (E[e[x/u]], \varphi)}$
(CONV)	$\dfrac{y \text{ is a fresh free variable}}{\Pi, (E[\backslash x \texttt{ -> } e], \varphi) \longrightarrow \Pi, (E[\backslash y \texttt{ -> } e[x/y]], \varphi)}$
(SERV)	$\dfrac{s x_1 \cdots x_n := e \in \mathcal{S}}{\Pi, (E[s v_1 \cdots v_n], \varphi) \longrightarrow \Pi, (E[e'[x_1/v_1, \ldots, x_n/v_n]], \varphi)}$

The evaluation of a receive expression $p?$ retrieves and returns the first value of p. The port descriptor mapping φ is modified to reflect the fact that the first message of p has been extracted. Similarly, the evaluation of a send expression, $p!v$, results in v being appended to the word at p. Since port descriptors are globally unique, there will only be one such p in the system.

The rules for (COND-FALSE) and (CHOICE-RIGHT) are similar to the rules for (COND-TRUE) and (CHOICE-LEFT) given in Table 3. If the condition b evaluates to `True`, then the second argument e_1 is evaluated next, else if the condition b evaluates to `False`, the third argument e_2 is evaluated next. For a choice expression $e_1 + e_2$, if the expression on the left e_1 can be evaluated, then it is evaluated. Similarly, the right-hand side e_2 is evaluated, if it can be evaluated. However, the choice of which one is evaluated is made non-deterministically.

Table 3. Semantics of DAML-S Core - II

$$\text{(SEQ)} \quad \Pi, (E[\texttt{return}\ v\ \texttt{>>=}\ e], \varphi) \longrightarrow \Pi, (E[(e\ v)], \varphi)$$

$$\text{(SPAWN)} \quad \Pi, (E[\texttt{spawn}\ e], \varphi) \longrightarrow \Pi, (E[\texttt{return}\ ()], \varphi), (e, \emptyset)$$

$$\text{(PORT)} \quad \frac{p \text{ new } \texttt{PortRef} \qquad \varphi'(x) = \begin{cases} \epsilon & \text{if } x = p; \\ \varphi(x) & \text{otherwise.} \end{cases}}{\Pi, (E[\texttt{newPort}\ \tau], \varphi) \longrightarrow \Pi, (E[\texttt{return}\ p], \varphi')}$$

$$\text{(REC)} \quad \frac{p \in Dom(\varphi) \qquad \varphi(p) = v \cdot w \qquad \varphi'(x) = \begin{cases} w & \text{if } x = p; \\ \varphi(x) & \text{otherwise.} \end{cases}}{\Pi, (E[p?], \varphi) \longrightarrow \Pi, (E[\texttt{return}\ v], \varphi')}$$

$$\text{(SEND)} \quad \frac{p \in Dom(\varphi_2) \qquad \varphi_2(p) = w \qquad \varphi_2'(x) = \begin{cases} w \cdot v & \text{if } x = p; \\ \varphi_2(x) & \text{otherwise.} \end{cases}}{\Pi, (E[p!\mathrm{v}], \varphi_1), (e, \varphi_2) \longrightarrow \Pi, (E[\texttt{return}\ ()], \varphi_1), (e, \varphi_2')}$$

$$\text{(COND-TRUE)} \quad \Pi, (E[\texttt{cond True}\ e_1\ e_2], \varphi) \longrightarrow \Pi, (E[e_1], \varphi)$$

$$\text{(CHOICE-LEFT)} \quad \frac{\Pi, (E[e_1], \varphi) \longrightarrow \Pi', (E[e_1'], \varphi')}{\Pi, (E[\texttt{choice}\ e_1\ e_2], \varphi) \longrightarrow \Pi', (E[e_1'], \varphi')}$$

5 Conclusions

We have presented a formal syntax and semantics for the Web services specification language DAML-S. Having a reference semantics for DAML-S during its design phase helps inform its further development, bringing out ambiguities and clarification issues. It can also form a basis for the future DAML-S execution model. With a formal semantics facilitates the construction of automatic tools to assist in the specification of Web services. Techniques to automatically verify properties of Web service specifications can also be explored with the foundation of a formal semantics. Since DAML-S is still evolving, the semantics needs to be constantly updated to keep up with current specifications of the language.

References

1. The DAML Services Coalition. DAML-S: Semantic Markup For Web Services. In Proceedings of the International Semantic Web Workshop, 2001
2. Frank Huch. Verification of Erlang Programs using Abstract Interpretation and Model Checking. ACM International Conference of Functional Programming 1999.
3. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign calls in Haskell. November 2000. `http://research.microsoft.com/Users/simonpj/papers/marktoberdorf.htm`
4. Simon Peyton Jones and John Hughes, editors. Haskell 98: A Non-strict, Purely Functional Language `http://www.haskell.org/onlinereport/`
5. Philip Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, Springer Verlag, LNCS 925, 1995.
6. Frank Huch and Ulrich Norbisrath. Distributed Programming in Haskell with Ports. Lecture Notes in Computer Science, Vol. 2011, 2000.
7. Simon Peyton Jones and Andrew Gordon and Sigbjorn Finne. Concurrent Haskell. POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. St. Petersburg Beach, Florida, pg. 295–308, 1996.
8. Annotated DAML+OIL Ontology Markup `http://www.daml.org/2001/03/daml+oil-walkthru.html`
9. Dan Brickley and R. V. Guha. *Resource Description Framework (RDF) Schema Specification 1.0*, W3C Candidate Recommendation 27 March 2000. `http://www.w3c.org/TR/rdf-schema/`
10. Matthias Felleisen and Daniel P. Friedman and Eugene E. Kohlbecker and Bruce Duba. A syntactic theory of sequential control. Theoretical Computer Science, Vol. 52, No. 3, pg. 205–237, 1987.