Concurrent Execution Semantics for DAML-S with Subtypes

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 describe a formalised syntax and an initial reference semantics for DAML-S, which incorporates subtype polymorphism. We contrast our semantics with an alternate semantics proposed for DAML-S, based on the situation calculus and Petri nets.

Keywords: Agents, Services, Languages and Infrastructure, Ontologies.

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 [8], 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 include UDDI, WSDL, WSFL and XLANG, which address different aspects of Web service description. UDDI (Universal Description, Discovery and Integration) [15], for instance, is primarily a repository technology and concerns itself with the storage and retrieval of Web service descriptions. WSDL (Web Services Description Language) [16] describes a Web service in terms how the interaction with it takes place: the messages it understands; the ports on which it can receive and send messages. WSFL (Web Services Flow Language) [17] and XLANG [18] describe how services can be composed together, and the behaviour/interaction protocol of a Web service. DAML-S is unique in that, due to its foundations

in DAML+OIL, it provides markup that can be semantically meaningful for intelligent agents.

In this paper, we describe an interleaving, strict operational semantics for DAML-S[1] informally described in [1]. The formalised syntax for DAML-S supports subclass polymorphism, which captures the subsumption-based component of DAML inferencing. Subclass polymorphism stems from object-oriented programming, where if an object expects a value of class τ, it can also accept values of any subclass τ' of τ. Similarly, an agent that accepts an input of class C_1 and recognises that C_1 is a subclass of C_2, can also accept instances of C_2, as inputs.

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. A formal semantics for DAML-S Core is given in Section 4. In the following Section 5, we model subclasses in DAML-S with the help of class constraints. Finally, in Section 6, we compare our approach to the definition of the semantics of DAML-S with another approach using situation calculus and Petri nets [11].

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 the 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* or *composite* service, built up from simpler, atomic services. The service grounding describes how the service can be

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

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. 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, i.e. interaction between the participants in a service transaction. The DAML-S grounding uses WSDL service descriptions to specify the communication between participants of a service transaction. Since modelling the communication within a service transaction is essential to describing the execution semantics of a service described in DAML-S, we will define a set of what we consider to be basic communication primitives, for example, for the sending and receiving of messages. These have close counterparts in WSDL.

In the next section, we first map DAML-S Core constructs onto a formal syntax, based on a core concurrent functional language. We then define a semantics for the DAML-S Core constructs in terms of the formal functional syntax.

3 Modelling DAML-S Core

The DAML-S class `Process` and its subclasses, representing services/agents[2], are modelled as functions. DAML-S agents essentially take in inputs and return outputs, exhibiting function-like behaviour. A Web document, for example, is an 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

[2] Services have a description and an execution component and therefore, can be considered as active processes or agents. In the following, we do not distinguish between agents and services. Note the agents described here are simply processes and do not necessarily display any complex, autonomous behaviours.

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 with the help of type constraints, presented in Section 5. At this stage, we do not model the relations and properties between the classes in an ontology. 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 $(\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3))$.

Type expressions build the term algebra $T_{\mathcal{F}}(\mathcal{V})$. DAML-S agents can be polymorphic with respect to their input and output. An example of a polymorphic agent is one which simply returns its input of arbitrary type, as output. 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 `(>>=)`: `∀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. The initial proposal for the DAML-S grounding, based on WSDL, also defines communication between services in terms of ports and messages. As we shall see, however, our notion of ports is related to the WSDL ports, but they are different abstractions.

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 $Exp(\Sigma)^\tau$.*

In Table 1, the base constructs which represent a composition of agents are `cond`, `>>=` (which is a binary service representing the sequential execution of its subprocesses), `spawn` and `choice`. Other constructs such as `Split+Join` can be defined in terms of these and we do not model them further.

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*

$$s \;\; x_1 \cdots x_n := e$$

$s \in \mathcal{S}$ is said to have type $\tau_1 \to \cdots \to \tau_n \to \tau$. $\mathcal{S}$ 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 $\tau_1 \to \tau_2 \to \tau_3$ instead of $(\tau_1, \tau_2) \to \tau_3$.

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` operation 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` operation takes two I/O actions as arguments, makes a non-deterministic choice between the two and returns it as

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	$e_1 ! e_2 \in Exp(\Sigma)^{\texttt{IO ()}}$ for $e_1 \in Exp(\Sigma)^{\texttt{Port}\ \tau}$, $e_2 \in Exp(\Sigma)^{\tau}$
rec	$e? \in Exp(\Sigma)^{\texttt{IO}\ \tau}$ 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 $e_i \in Exp(\Sigma)^{\tau_i}$, $s \in \mathcal{S}^{\tau_1 \to \cdots \to \tau_n \to \tau}$

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 [19] and the π-calculus [20].

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*

Table 2. Semantics of DAML-S Core - I

$$\text{(FUNC)}\ \frac{\phi \in \Omega}{\Pi, (E[\phi v_1 \cdots v_n], \varphi) \longrightarrow \Pi, (E[\phi_{\mathfrak{A}} v_1 \cdots v_n], \varphi)}$$

$$\text{(APPL)}\ \frac{\text{free}(u) \cap \text{bound}(e) = \emptyset}{\Pi, (E[(\backslash x\ \texttt{->}\ e)\ \ u)], \varphi) \longrightarrow \Pi, (E[e[x/u]], \varphi)}$$

$$\text{(CONV)}\ \frac{y \text{ is a fresh free variable}}{\Pi, (E[\backslash x\ \texttt{->}\ e], \varphi) \longrightarrow \Pi, (E[\backslash y\ \texttt{->}\ e[x/y]], \varphi)}$$

$$\text{(SERV)}\ \frac{sx_1 \cdots x_n := e \in \mathcal{S}}{\Pi, (E[sv_1 \cdots v_n], \varphi) \longrightarrow \Pi, (E[e'[x_1/v_1, \ldots, x_n/v_n]], \varphi)}$$

ports:

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

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 $\longrightarrow \subset$ State $\times$ 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'.*

An inference rule of the form

$$\frac{A}{B}$$

represents the drawing of a conclusion B on the basis of the premise A. Thus, if A is true, B is also true. In the following, we describe each inference rule in the Tables 2 and 3.

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

Table 3. Semantics of DAML-S Core - II

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

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

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

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

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

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

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

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.

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.

5 Subclass Polymorphism in DAML-S

Due to its roots in DAML+OIl, a distinguishing characteristic of DAML-S is that it enables some measure of semantic inferencing to be made by an agent. DAML+OIL enables many kinds of inferencing. Here, we model subsumption-based semantic inferencing as it has been done in object-oriented programming. Most object-oriented languages support subclass polymorphism, where when a value or instance of class A is expected, a value or instance of any subclass B of A can also be accepted. Thus, if an agent expects an input of type `Book`, it can also accept inputs of type `AudioBook`, provided an `AudioBook` is a kind of `Book`.

We use class constraints to model subclasses:

Definition 7 (Type Constraints). *The set of* type constraints *over* $T_{\mathcal{F}}(\mathcal{V})$ *is defined as* $\{p(\tau_1, \ldots, \tau_n) | p \in P_n, \tau_i \in T_{\mathcal{F}}(\mathcal{V})$*, where* $P = \bigcup P_n$ *is a finite* n*-indexed family of predicate symbols.*

For our purposes, we only need subtype constraints over DAML-S classes. Thus, we choose

$$P = P_2 = \{\lhd\}$$

We write a subtype constraint 'τ_1 is a subclass of τ_2' as $\tau_1 \lhd \tau_2$. With the help of type constraints, we can now extend type schemas with constraints on the contained type variables.

Definition 8 (Constrained Type Schemas). *A* constrained type schema *is a type schema of the form* $\forall\alpha_1, \ldots, \alpha_n.\tau | C$*, where* $\forall\alpha_1, \ldots, \alpha_n.\tau$ *is a type schema and* C *is a set of type constraints. The pair* $\tau|C$ *is also referred to as a* constrained type. *Instead of* $\tau|\emptyset$ *we also write* τ*.*

For example, the type $\forall$`a.a` $|\{$`a` $\lhd$ `Book`$\}$ is a constrained type schema, representing a subtype of `Book`. An instantiation of a constrained type schemas yields a generic instance.

Definition 9 (Generic instance). *A constrained type* $\tau'|C'$ *is called a* generic instance *of a type schema* $\forall\alpha_1, \ldots, \alpha_n.\tau|C$*, if a substitution* σ *exists, such that*

$$\sigma\tau|\sigma C = \tau'|C'$$

where $\sigma(\alpha_i) = \tau_i$ *for all* $i = 1, \ldots, n$ *and* $\sigma(\beta) = \beta$ *for all* $\beta \notin \{\alpha_1, \ldots, \alpha_n\}$*.*

Thus, the type `AudioBook` $|\{$`AudioBook` $\lhd$ `Book`$\}$ is a generic instance of the constrained type schema $\forall$`a.a` $|\{$`a` $\lhd$ `Book`$\}$. A well-typed expression must be well-formed according to the rules in Table 1 and satify the type constraints on its stated type.

The satisfaction of type constraints for a particular type expression depends on whether the classes in the constraints do in fact possess the required subclass relationships. The constraints can be checked against the class definitions in the specification and the ontologies it references.

To verify the satisfaction of type constraints for a particular specification, we need to formalise the `subClassOf` property, used for the definition of subclasses.

Definition 10 (Direct Subclass). *The* direct subclass *relation $\mathcal{H}_S$ for a service specification S is defined as follows: $(C_1, C_2) \in \mathcal{H}_S$ if and only if there exists an ontology referenced by specification S, where class C_1 is a* `subClassOf` *class C_2. In the following, we omit the subscript S and simply write $\mathcal{H}$ instead of $\mathcal{H}_S$, since we will usually be referring to a single specification S.*

Definition 11 (Subclass Hierarchy). *A* subclass hierarchy *$\mathcal{H}^*$ induced by a direct subclass relation $\mathcal{H}$ is formed by taking its reflexive and transitive closure.*

The base DAML+OIL ontology defines a top `Thing` class and a bottom `Nothing` class. If $Class_S$ represents the set of all classes defined for a specification S, then $\forall C \in Class_S$, the following hold:

$$(C, \texttt{Thing}) \in {\mathcal{H}_S}^*, \qquad (\texttt{Nothing}, C) \in {\mathcal{H}_S}^*$$

Definition 12 (Satisfiability of Subtype Constraints). *A substitution σ* satisfies *a subtype constraint $\tau_1 \lhd \tau_2$ with respect to a subclass hierarchy $\mathcal{H}^*$ if $(\sigma\tau_1, \sigma\tau_2) \in \mathcal{H}^*$. We notate this as $\sigma \models_{\mathcal{H}^*} \tau_1 \lhd \tau_2$.*

A substitution σ satisfies a set of subtype constraints C, if, for all $c \in C$: $\sigma \models_{\mathcal{H}^*} c$. We write this as $\sigma \models_{\mathcal{H}^*} C$.

A set of subtype constraints is satisfiable *with respect to a subclass hierarchy $\mathcal{H}^*$, if there exists a substitution σ such that $\sigma \models_{\mathcal{H}^*} C$. This is notated as $\models_{\mathcal{H}^*} C$.*

The subclass hierarchy $\mathcal{H}^*$ is defined over classes. However, the constraints can also involve functional type expressions and expressions involving the type constructors `Port` and `IO`. Such constraints can be broken down into simpler constraints involving solely classes as follows:

- If a port can accept values of type `b`, it can also accept values of any subtype `a` of `b` and thus also has type `Port a`.

$$\texttt{a} \lhd \texttt{b} \longleftrightarrow \texttt{Port b} \lhd \texttt{Port a} \tag{1}$$

- Similarly, an I/O action that returns a value of type `a` also returns a value of any supertype `b` of `a`. Therefore, an I/O action of type `IO a` also has type `IO b`:

$$\texttt{a} \lhd \texttt{b} \longleftrightarrow \texttt{IO a} \lhd \texttt{IO b} \tag{2}$$

- A function that accepts arguments of type `a` can also accept arguments of any subtype `a'` of `a`. A function that returns values of type `b` also returns values of any supertype `b'` of `b`. Thus, any function that has type $\texttt{a} \to \texttt{b}$ also has type $\texttt{a'} \to \texttt{b'}$:

$$\texttt{a'} \lhd \texttt{a}, \texttt{b} \lhd \texttt{b'} \longleftrightarrow \texttt{a} \to \texttt{b} \lhd \texttt{a'} \to \texttt{b'} \tag{3}$$

  This relationship holds even if the type variables `a` and `b` are themselves substituted with functional type expressions.

Table 4. The DAML-S Core Type Environment $\Gamma_{\mathcal{D}}$

cond	$\forall$a,b,c,d.IO a $\rightarrow$ b $\rightarrow$ c $\rightarrow$ d | {a $\lhd$ boolean, b $\lhd$ d, c $\lhd$ d}
return	$\forall$a.a $\rightarrow$ IO a
>>=	$\forall$a,b,c.IO a $\rightarrow$ (b $\rightarrow$ IO c) $\rightarrow$ IO c | {a $\lhd$ b}
!	$\forall$a,b.Port a $\rightarrow$ b $\rightarrow$ IO () | {b $\lhd$ a}
?	$\forall$a.Port a $\rightarrow$ IO a
newPort	$\forall$a.IO (Port a)
spawn	$\forall$a.IO a $\rightarrow$ IO (IO a)
choice	$\forall$a,b,c.IO a $\rightarrow$ IO b $\rightarrow$ IO c | {a $\lhd$ c, b $\lhd$ c}

The satisfiability of the simpler constrainrs obtained through this transformation can now be checked against the subclass hierarchy $\mathcal{H}^*$.

Definition 13 (Type Environment). *A* type environment Γ *is a mapping from type variables to constrained type schemas.*

The initial typing environment used for typing DAML-S Core expressions will be denoted by $\Gamma_{\mathcal{D}}$. Most DAML-S Core constructs can be defined as higher-order agents, i.e. agents that can take agents as input, analogous to higher-order functions. To simplify the typing rules, we will consider them as pre-defined higher-order agents and extend $\Gamma_{\mathcal{D}}$ with their constrained type schemata, as summarised in Table 4.

With the help of a type environment, which maps all the identifiers in a service description onto constrained type schemata, we can define the well-typedness of an agent specification.

Definition 14 (Well-typedness of Service Definitions). *A service definition*

$$s \;\; x_1 \cdots x_n := e$$

is called well-typed *with respect to a type environment* Γ *and a subclass hierarchy* $\mathcal{H}^*$*, if the following conditions are fulfilled:*

- $\Gamma(s) = \forall \alpha_1, \ldots, \alpha_n.\tau | C$
- $\Gamma, \mathcal{H}^* \vdash \backslash e_1 \ldots \backslash e_n.e : \tau | C$ *can be derived from the typing rules in Table 5.*
- $\models_{\mathcal{H}^*} C$

Table 5. Typing DAML-S expressions

[Axiom]	$\frac{}{\Gamma, \mathcal{H}^* \vdash x : \tau \mid C}$ if $\tau \mid C$ is a generic instance of $\Gamma(x)$
[Abstraction]	$\frac{\Gamma[x/\tau \mid C], \mathcal{H}^* \vdash e : \tau' \mid C'}{\Gamma, \mathcal{H}^* \vdash \lambda x.e : \tau \rightarrow \tau' \mid C'}$
[Application]	$\frac{\Gamma, \mathcal{H}^* \vdash e_1 : \tau_1 \rightarrow \tau_2 \mid C_1 \quad \Gamma, \mathcal{H}^* \vdash e_2 : \tau_1' \mid C_2}{\Gamma, \mathcal{H}^* \vdash e_1 e_2 : \tau_2 \mid C_1 \cup C_2 \cup \{\tau_1' \lhd \tau_1\}}$

Thus, for a service definition to be well-typed, the type derived for e must match the constrained type schema of s in the type environment. Furthermore, the constraints C on the type for e must be satisfiable with respect to the class hierarchy $\mathcal{H}^*$.

The typing rule [Axiom] states that given a type environment Γ, x has the constrained type $\tau \mid C$, if $\tau \mid C$ is the generic instance of $\Gamma(x)$. This rule thus examines the type environment Γ to determine the type of the variable x. The rule [Abstraction] states that the expression $\lambda x.e$ has type $\tau \rightarrow \tau' \mid C'$ under the assumption that e has the type $\tau' \mid C'$ in the type environment extended with the assignment $[x/\tau \mid C]$. The constraint set C' does not need to be extended with the constraints in C. If the expression e already contains x in an application, then the constraints C' already contain the constraints C, that is $C \subseteq C'$. If the expression e does not contain x, then the type of x and its constraints are immaterial.

The rule [Application] assigns the type τ_2 to the application $e_1 e_2$, if under the same type environment e_1 has type $\tau_1 \rightarrow \tau_2$ and e_2 has type τ_1'. The type constraints on $e_1 e_2$ is the union of the constraints on e_1 and e_2. The requirement that τ_1' must be a subtype of τ_1 is captured through the additional constraint $\tau_1' \lhd \tau_1$.

6 Related Work

An alternative semantics for the process model has been proposed by Narayanan et al. [11], which uses the situation calculus to model a subset of DAML-S, essentially processes and their inputs, outputs, preconditions and effects. Axioms in the situation calculus are mapped onto Petri net representations, which are then used to describe the semantics of the DAML-S control constructs.

The situation calculus describes a state or situation in the world in terms of propositions, which can be true or false in that state. Actions are described in terms of the their preconditions and effects on the state, which propositions

must hold for the action to take place and which propositions hold true after the action has taken place.

In the case of multi-agent systems, however, every agent will have its own set of propositions, its own view on the world. Not only does this place a significant burden on the system designer to define the comprehensive set of relevant propositions and axioms, it is also not clear how the differing world views of the agents will be reconciled when they interact. How much does each agent need to know about the knowledge of the other agent? Or even about the world, to be able to perform an action?

An additional issue arises with respect to the composition of agents. Although, one can represent and reason about the sequences of actions a single agent can perform with the help of planning systems, the composition of agents is a slightly different matter. For instance, when performing two actions a_1 and a_2 in sequence, the agent's knowledge about the world after completing the first action a_1 is the same as the agent's knowledge before beginning the second action a_2. On the other hand, if the actions are performed by two separate agents, the knowledge of the first agent after performing a_1 is unlikely to be the same as the knowledge of the second agent about to perform a_2.

The Petri net semantics and the semantics described in this paper are equivalent in most respects, but there are a couple of minor differences. The `choice` agent described in this paper chooses a single agent for execution from among a set S of agents, whereas in the Petri net semantics, it is defined as choosing a subset of agents for concurrent execution. In our semantics, this alternate definition can be easily modelled as a `choice` between all the subsets of S that are to be concurrently executed. Since S contains a finite number of elements, the `choice` agent also has a finite number of arguments. The Petri net semantics of the `Concurrent` class also does not explicitly model the possibility of interaction between the concurrently running agents. Similarly, we do not describe the `Unordered` class explicitly because it is equivalent to the `Concurrent` class.

Within our approach, the inputs, outputs, preconditions and effects of a composite agent can be determined relatively easily from those of its component subagents. Furthermore, our semantics explicitly describes subclass polymorphism and it is close to an execution model, since the grounding maps easily onto our semantics.

7 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. 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. We extended the DAML-S formalisation with subtype polymorphism, which captures certain aspects of DAML inferencing.

This work was partially supported by DARPA/AFRL Contract No. F30602-00-2-0592.

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. Simon Peyton Jones. *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. Dan Connolly, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider and Lynn Andrea Stein. DAML+OIL (March 2001) Reference Description
   `http://www.w3.org/TR/daml+oil-reference`
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.
11. Srini Narayanan and Sheila A. McIlraith. *Simulation, Verification and Automated Composition of Web Services.* To appear in the Eleventh International World Wide Web Conference 2002, Hawaii, USA.
12. Philipp Niederau. *Objectorientierte Erweiterungen einer deklarativen Programmiersprache.* Diplomarbeit, RWTH Aachen, August 2000.
13. Stefan Kaes. *Type inference in the presence of overloading, subtyping and recursive types.* In 1992 ACM Conference on Lisp and Functional Programming, pp 193-204. ACM, August 1992.
14. Anupriya Ankolekar, Frank Huch and Katia Sycara. *Concurrent Semantics for the Web Services Specification Language DAML-S.* To appear in Proceedings of the Fifth International Conference on Coordination Models and Languages, York, UK, 2002.

15. UDDI Technical White Paper, September 6, 2000.
http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.PDF
16. Web Services Description Language (WSDL) 1.1, W3C Note, March 15, 2001.
http://www.w3.org/TR/wsdl
17. Frank Leymann, Web Services Flow Language (WSFL) 1.0, May 2001.
http://www-4.ibm.com/software/solutions/webservices/pdf/WSFL.pdf
18. Satish Thatte, XLANG: Web Services for Business Process Design, 2001.
http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm
19. Robin Milner. Communication and Concurrency. Prentice Hall, 1989.
20. Robin Milner. The Polyadic pi-Calculus: A Tutorial.
http://www.lfcs.informatics.ed.ac.uk/reports/91/ECS-LFCS-91-180/ECS-LFCS-91-180.ps