Concurrent Execution Semantics of DAML-S with Subtypes

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

[1] Carnegie Mellon University, Pittsburgh PA 15213, USA
[2] Christian-Albrechts-University of Kiel, 24118 Kiel, Germany

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, which incorporates subtype polymorphism. The semantics we describe is derived from the semantics for Erlang and Concurrent Haskell. 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 [4], 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) [17], for instance, is primarily a repository technology and concerns itself with the storage and retrieval of Web service descriptions. WSDL (Web Services Description Language) [3] 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) [11] and XLANG [16] 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.

An informal description of the semantics of DAML-S[1] has been given in [1]. An interleaving, strict operational semantics for DAML-S is presented in [2]. In this paper, we extend the type system for DAML-S with 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 next section, Section 2, presents the DAML-S ontologies and the process model of a service. A core subset of DAML-S, referred to as *DAML-S Core*, is modelled in [2], such that 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 following sections 3 and 4 discuss some of the issues involved in developing a formal model for DAML-S and present the syntax and semantics of DAML-S Core. In the following Section 5, we extend the type system of DAML-S Core with subclass. Subclasses in DAML-S with the help of type 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 [14].

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 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

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

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 [4], 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. 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 $\tau_1, \ldots, \tau_n$.*

[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.

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 $(\tau_1 \rightarrow (\tau_2 \rightarrow \tau_3))$.

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 [9], 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* [8], 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 $x_i \in Var^{\mathcal{T}_i}, x_i$ pairwise different and $e \in Exp(\Sigma)^{\mathcal{T}}$. 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 $\tau_1 \to \cdots \to \tau_n \to \tau$. $\mathcal{S}$ 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 `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 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

A formal semantics for DAML+OIL has been defined denotationally [18] and axiomatically [6]. Although the DAML-S ontology is defined in terms of DAML+OIL and therefore inherits its semantics, they are clearly inappropriate for a definition

Table 1. DAML-S Core Expressions

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

of the operational meaning of DAML-S constructs. Describing the operational semantics of concurrent and distributed systems such as the DAML-S environment is often far simpler and more natural than describing the denotational semantics. Distributed systems tend to be non-terminating and non-deterministic, making it difficult to describe them simply on the basis of their input-output behaviour. Defining differing semantics for DAML+OIL and DAML-S does not constitute a problem. The operational semantics of DAML-S is layered on top of the denotational semantics of DAML+OIL, with the type system mediating between the two.

In this section, we describe a formal operational semantics for Core DAML-S. Our semantics is based on the operational semantics for Erlang [7] and Concurrent Haskell [9] programs, inspired by the structural operational semantics of CCS [12] and the π-calculus [13].

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: $State := \mathcal{P}_{fin}(Agent)$*

Table 2. Semantics of DAML-S Core - I

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

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

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

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

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:*

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

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 *$\mathcal{EC}$ [5] 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'.*

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. The evaluation of a receive expression $p?$ retrieves and returns the first value of p.

Table 3. Semantics of DAML-S Core - II

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

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

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

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

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

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

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

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, $p\texttt{!}v$, 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 subtype polymorphism. Our type system is similar to that of ObjectCurry [15]. In the case of ObjectCurry, subtype polymorphism was present only for objects and messages. In our case, it can occur during any service invocation. Additionally, subtype polymorphism is not restricted to classes, it can also occur over functional types. Furthermore, DAML-S Core supports multiple inheritance, which is not present in ObjectCurry.

Subtypes in DAML-S are expressed with the help of constraints on type expressions. Type schemas are thus extended to *constrained type schemas* 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 subtype constraints. A subtype constraint 'τ_1 is a subtype of τ_2' is written as $\tau_1 \lhd \tau_2$. Instead of $\tau|\emptyset$ we also write τ. For example, the class $\forall$`a.a` $|\{$`a` $\lhd$ `Book`$\}$ is a constrained type schema, representing a subtype of `Book`. An instantiation of a constrained type schema yields a generic instance.

A *generic instance* of a type schema $\forall \alpha_1, \ldots, \alpha_n.\tau | C$ is a constrained type $\tau' | C'$, 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`$\}$. To be well-typed, an expression must be well-formed according to the rules in Table 1 and furthermore, it should satisfy the type constraints on its stated type. Whether the type constraints in a type expression are satisfied depends on whether the types in the constraints do in fact possess the required subtype relationships. This can be verified by checking the constraints against the class definitions in the specification and the ontologies it references.

Definition 7 (Direct Subclass and subclass hierarchy). *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.*[3] *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.*

Definition 8 (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 $\sigma \models_{\mathcal{H}^*} \tau_1 \lhd \tau_2$.*[4]

[3] 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.

[4] A substitution σ satisfies a set of subtype constraints C, if, for all $c \in C$: $\sigma \models_{\mathcal{H}^*} c$. Notation: $\sigma \models_{\mathcal{H}^*} C$. A set of subtype constraints is *satisfiable* with respect to a subclass hierarchy $\mathcal{H}^*$, if there exists a substitution σ such that $\sigma \models_{\mathcal{H}^*} C$. Notation: $\models_{\mathcal{H}^*} C$.

The subclass hierarchy $\mathcal{H}^*$ is defined over classes. However, subtype 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, using the transformation described in Table 4. If a port can accept values of type τ_1, it can also accept values of any subtype τ_2 of τ_1 and thus also has type $\texttt{Port}\ \tau_2$. Thus, a constraint of the form $\texttt{Port}\ \tau_1 \lhd \texttt{Port}\ \tau_2$ can be simplified to the constraint $\tau_2 \lhd \tau_1$. Similarly, an I/O action of type $\texttt{IO}\ \tau_1$ also returns a value of any supertype τ_2 of τ_1 and thus also has type $\texttt{IO}\ \tau_2$. A function that accepts arguments of type τ_1 can also accept arguments of any subtype τ_3 of τ_1. A function that returns values of type τ_2 also returns values of any supertype τ_4 of τ_2. Thus, any function that has type $\tau_1 \rightarrow \tau_2$ also has type $\tau_3 \rightarrow \tau_4$. This relationship holds even if the type variables are themselves substituted with functional type expressions.

Table 4. Simplifying Constraints with Type Constructors

[`Port`]	$\dfrac{\{\texttt{Port}\ \tau_1 \lhd \texttt{Port}\ \tau_2\} \cup C}{\{\tau_2 \lhd \tau_1\} \cup C}$ if $(\tau_2 \lhd \tau_1) \notin C$
[`IO`]	$\dfrac{\{\texttt{IO}\ \tau_1 \lhd \texttt{IO}\ \tau_2\} \cup C}{\{\tau_1 \lhd \tau_2\} \cup C}$ if $(\tau_1 \lhd \tau_2) \notin C$
[$\rightarrow$]	$\dfrac{\{\tau_1 \rightarrow \tau_2 \lhd \tau_3 \rightarrow \tau_4\} \cup C}{\{\tau_3 \lhd \tau_1, \tau_2 \lhd \tau_4\} \cup C}$ if $(\tau_3 \lhd \tau_1), (\tau_2 \lhd \tau_4) \notin C$

The satisfiability of the simpler constraints obtained through this transformation can then be checked against the subclass hierarchy $\mathcal{H}^*$. The initial typing environment used for typing DAML-S Core expressions will be denoted by $\Gamma_{\mathcal{D}}$. Most DAML-S Core constructors can be considered to be pre-defined higher-order agents and to simplify the typing rules, we will consider them as such. $\Gamma_{\mathcal{D}}$ can then be extended with their constrained type schemata, as summarised in Table 5.

Definition 9 (Well-typedness of Service Definitions). *A service definition* $s\ x_1 \cdots x_n$ `:=` e *is considered to be* well-typed *with respect to a type environment* Γ *and a subclass hierarchy* $\mathcal{H}^*$*, if the following conditions are fulfilled:*

- $\Gamma(s) = \forall \alpha_1, \ldots, \alpha_m.\tau | C$
- $\Gamma, \mathcal{H}^* \vdash \lambda e_1 \ldots \lambda e_m.e : \tau | C$ *can be derived from the typing rules in Table 6.*
- $\models_{\mathcal{H}^*} C$

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,

Table 5. The DAML-S Core Type Environment $\Gamma_{\mathcal{D}}$

`cond:`	`∀a,b,c.IO boolean → a → b → c`
`return:`	`∀a.a → IO a`
`>>=:`	`∀a,b,c.IO a → (b → IO c) → IO c`
`!:`	`∀a,b.Port a → b → IO ()`
`?:`	`∀a.Port a → IO a`
`newPort:`	`∀a.IO (Port a)`
`spawn:`	`∀a.IO a → IO (IO a)`
`choice:`	`∀a,b,c.IO a → IO b → IO c`

Table 6. Typing DAML-S expressions

[Axiom] $$\frac{}{\Gamma, \mathcal{H}^* \vdash x : \tau | C} \text{if } \tau|C \text{ is a generic instance of } \Gamma(x)$$

[Abstraction] $$\frac{\Gamma[x/\tau|C], \mathcal{H}^* \vdash e : \tau'|C'}{\Gamma, \mathcal{H}^* \vdash \lambda x.e : \tau \rightarrow \tau'|C'}$$

[Application] $$\frac{\Gamma, \mathcal{H}^* \vdash e_1 : \tau_1 \rightarrow \tau_2|C_1 \qquad \Gamma, \mathcal{H}^* \vdash e_2 : \tau_1'|C_2}{\Gamma, \mathcal{H}^* \vdash e_1 e_2 : \tau_2|C_1 \cup C_2 \cup \{\tau_1' \lhd \tau_1\}}$$

the constraints C on the type for e must be satisfiable with respect to the class hierarchy $\mathcal{H}^*$.

The typing rules [Axiom], [Abstraction] and [Application] are quite standard. Note that the constraint set C' in [Abstraction] 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. In [Application], the type constraints on $(e_1 e_2)$ are 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 $\tau_1' \lhd \tau_1$.

5.1 Type Inference for DAML-S

We present an algorithm for determining the type of a expression in DAML-S, assuming the expression does not contain polymorphic data structures. Our algorithm extends the algorithm described in [10] and [15] with the verification of constraint satisfaction for type expressions with multiple inheritance. The algorithm $\mathcal{E}$, presented in Table 7, determines the type expression of an expression in DAML-S. First, a type environment Γ and an inheritance hierarchy $\mathcal{H}^*$ need to be constructed. We let the initial type environment be $\Gamma_{\mathcal{D}}$. Given an ontology, a corresponding inheritance hierarchy $\mathcal{H}^*$ can be easily constructed. With the definition of the inheritance hierarchy and the type environment, the function $\mathcal{E}$ can now be used to determine the constrained types of the defined service descriptions.

Table 7. Type Inference Algorithm for DAML-S

$\mathcal{E}[\![x]\!](\Gamma, \mathcal{H}^*) = (\emptyset, \tau|C)$
 if $\tau|C$ is a generic instance of $\Gamma(x)$

$\mathcal{E}[\![\lambda x.e]\!](\Gamma, \mathcal{H}^*) = (\sigma, (\sigma\alpha) \to \tau|\sigma C)$
 if there exists a substitution σ and α a fresh variable, such that
 $(\sigma, \tau|C) = \mathcal{E}[\![e]\!](\Gamma[x/\alpha], \mathcal{H}^*)$

$\mathcal{E}[\![(e_1 e_2)]\!](\Gamma, \mathcal{H}^*) = (\sigma \circ \sigma_2 \circ \sigma_1, \sigma\beta|\sigma C)$
 if there exist substitutions $\sigma, \sigma_1, \sigma_2$ and fresh variables α, β, such that
 $(\sigma_1, \tau_1|C_1) = \mathcal{E}[\![e_1]\!](\Gamma, \mathcal{H}^*)$
 $(\sigma_2, \tau_2|C_2) = \mathcal{E}[\![e_2]\!](\sigma_1\Gamma, \mathcal{H}^*)$
 $\sigma = mgu(\tau_1, \alpha \to \beta)$
 $C = \sigma_2 C_1 \cup C_2 \cup \{\tau_2 \lhd \alpha\}$

The function $\mathcal{E}$ determines the type of an expression e with respect to a type environment Γ and an inheritance hierarchy $\mathcal{H}^*$ as a pair, consisting of a substitution σ and a constrained type expression $\tau|C$. If e is a variable, then $\mathcal{E}$ simply looks up its type in the type environment Γ and returns an empty substitution.

If e is an abstraction of the form $\lambda x.e'$, we first assign the variable x a fresh type variable α and determine the inferred type of e' under the type environment Γ and the inheritance hierarchy $\mathcal{H}^*$. The type of the abstraction is then $(\sigma\alpha) \to \tau$ under the constraints σC, where $\tau|C$ is the type inferred for e' with the substitution σ. Finally, if the type of an application $(e_1 e_2)$ is to be determined, we first determine the inferred types of e_1 and e_2 individually. The inferred type of e_1 is then unified with a functional type $\alpha \to \beta$. The constraints of the inferred type of $(e_1 e_2)$ is the union of the inferred types of e_1 and e_2 and a constraint that the inferred type of e_2 is a subtype of the argument type expected by e_1.

The application of $\mathcal{E}$ can result in type expressions with subtype constraints, which must also be satisfiable. An algorithm to test the satisfiability of such a set of subtype constraints is presented in [15].

6 Related Work

Web services are modelled as processes in a distributed system in DAML-S. Although the Web does introduce new concepts such as service advertisements, service brokering, auctions and so on, these exist over the distributed model. XLANG also takes the process-oriented approach and models Web services through a process calculus. WSFL, in contrast, uses a workflow approach, which is well-suited to define the composition of processes, but is less so for the precise specification of processes themselves.

An alternative semantics for the process model has been proposed by Narayanan et al. [14], 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 cannot be guaranteed 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 sub-agents. Furthermore, the semantics we describe 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, which can 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. We extended the DAML-S formalisation with subtype polymorphism, which captures certain aspects of DAML inferencing, in particular subsumption. Since DAML-S is still evolving, the semantics needs to be constantly updated to keep up with current specifications of the language. Additionally, as DAML-S grows to incorporate service transactions and service brokering, these notions can be formalised on top of the formal framework presented here.

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

References

1. A. Ankolekar, M. Burstein, J. Hobbs, O. Lassila, D. Martin, S. McIlraith, S. Narayanan, M. Paolucci, T. Payne, K. Sycara, and H. Zeng. DAML-S: Semantic markup for Web services. In *Proceedings of the International Semantic Web Working Symposium (SWWS)*, pages 411–430, 2001.
2. A. Ankolekar, F. Huch, and K. Sycara. Concurrent semantics for the web services specification language DAML-S. In *Proceedings of the Fifth International Conference on Coordination Models and Languages*, volume 2315 of *Springer Lecture Notes in Computer Science*. Springer Verlag, April 2002.
3. E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1, 2001.
4. D. Connolly, F. van Harmelen, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein. DAML+OIL (march 2001) reference description. `http://www.w3.org/TR/daml+oil-reference`.
5. M. Felleisen, D. P. Friedman, E. E. Kohlbecker, and B. Duba. A syntactic theory of sequential control. *Theoretical Computer Science*, 52(3):205–237, 1987.
6. R. Fikes and D. McGuinness. An axiomatic semantics for RDF, RDF-S, and DAML+OIL, W3C note 12.
7. F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99)*, volume 34-9 of *ACM SIGPLAN Notices*, pages 261–272. ACM Press, September 1999. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).
8. F. Huch and U. Norbisrath. Distributed programming in Haskell with ports. *Lecture Notes in Computer Science*, 2011, 2000.

9. S. P. Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Conference Record of POPL '96: The* 23$^{\text{rd}}$ *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 295–308, St. Petersburg Beach, Florida, 1996.
10. S. Kaes. Type inference in the presence of overloading, subtyping and recursive types. In *Proceedings of the Conference on Lisp and Functional programming*, pages 193–204. ACM Press, 1992.
11. F. Leymann. Web services flow language (WSFL) 1.0.
12. R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.
13. R. Milner. The polyadic π-calculus: A tutorial. Technical report, University of Edinburgh, 1991.
14. S. Narayanan and S. McIllraith. Simulation, verification, and automated composition of web services. In *Proceedings of the Eleventh International World Wide Web Conference (WWW2002)*, 2002.
15. P. Niederau. Objectorientierte erweiterungen einer deklarativen programmiersprache. Master's thesis, RWTH Aachen, August 2000.
16. S. Thatte. XLANG: Web services for business process design, 2001.
17. UDDI. The UDDI technical white paper. `http://www.uddi.org/`, 2000.
18. F. van Harmelen, P. F. Patel-Schneider, and I. Horrocks. A model-theoretic semantics for DAML+OIL. `http://www.daml.org/2001/03/model-theoretic-semantics.html`.