A Lightweight Method for Coordination of Agent Oriented Web Services*

David Robertson
Informatics, University of Edinburgh
dr@inf.ed.ac.uk

Abstract

Large scale multi-agent systems and Web services that are expected automatically to solve problems collaboratively must have ways of understanding the dialogues in which they are engaged. This understanding must be possible in an asynchronous environment and it must allow agents to be engaged in many dialogues simultaneously. It should not require centralised control for dialogue coordination, nor should it require individual agents to adapt their knowledge or beliefs solely for the purposes of dialogue. We describe a means of achieving this with the aid of a lightweight protocol.

Introduction

The Internet raises the prospect of engineering large scale systems that are not engineered in the traditional way, by tightly integrating modest numbers of components familiar to a single design team, but are assembled oportunistically from components built by disparate design teams. Ideally such systems would make it easy for new components to be designed and deployed in competition with existing components, allowing large systems to evolve through competitive design and service provision. That requires standardisation of the languages used for description of the interfaces between components - hence Web service specification efforts such as DAML-S (in the Semantic Web community) and performative-based message passing protocols such as FIPA-ACL and KQML (in the agent systems community). Although helpful these are, in themselves, insufficient to coordinate groups of disparate components. This is especially difficult in unbounded, distributed systems (like the Internet) because coordination depends on each component "being aware" of the state of play in its interaction with others when performing a shared task and being able to continue that interaction in a way likely to be acceptable to those others. As an example, consider the following problem to which we shall return later in this paper.

We decide to build a suite of Web services for arranging research workshops. The suite of services includes the following services:

1. A service that is able to suggest authorities on given research areas, based on data from academic Web sites, research citations, *etc*. The AKT triple store query interface is an example of this sort of "headhunter" component.
2. A service that, given the name of a person, can guess where an appropriate "Web link" (*e.g.* a URL) exists for that person. Google is an example of this sort of "finder" component.
3. A service that maintains a diary of appointments for a person and is capable of answering queries about when particular days are free in the diary. Various Web-based diary managers currently exist.
4. A service that assists a workshop organiser in coordinating all of the above to arrange a workshop on a given topic.

Although the problem above is mundane it demonstrates a coordination issue for Semantic Web services. The coordination service (4) must interact with all the other services and the choice of those services with which it interacts will be determined by the replies it obtains from earlier services - so the results from the headhunter component (1) will determine the form and frequency of interactions with the finder component. There are likely to be many such interactions so it is non-trivial to keep track of the state of play of the dialogue between services. This is particularly important in an asynchronous and distributed system like the Web where each service may be taking part in a number of different problem solving activities and must not confuse them - for example we might expect a workshop coordinator to be arranging several workshops at once.

Solving coordination problems like these requires some description of the focus of coordination. One way of doing this is by the use of policy languages (*e.g.* (Kagal, Finin, & Joshi 2003)) that describe what services are and are not allowed to do with respect to a given aspect of their behaviour (*e.g.* their security policies). By enforcing appropriate policies we may provide a safe envelope of operation within which services operate. This is useful but not the same as specifying more directly the interactions required between services. For this it has been more natural to use concepts from planning and temporal reasoning to represent the required behaviours of individual services (*e.g.* (Decker *et al.* 1997)); shared models for coordinat-

*This work was supported by the Advanced Knowledge Technologies IRC funded by UK Engineering and Physical Sciences Research council (grant GR/N15764/01).

ing services (*e.g.* (Giampapa & Sycara 2002)) or the process of composing services (*e.g.* (McIlraith & Son 2002; Sheshagiri, desJardins, & Finin 2003)). The method of coordination described in this paper is a temporal style of description used for coordination, although it also allows the incorporation of constraints into interaction protocols.

Distributed Dialogues

In what follows we shall use the word "agent" (rather then the less suggestive "component" or lengthier "semantic web service") to describe any component that may engage in complex coordinations. We assume that these agents provide Web services. A distributed dialogue is a conversation among a group of agents which can be described as a collection of dialogue sequences between agents. In a multi-agent system the speech acts conveying information between agents are performed only by sending and receiving messages. For example, suppose a dialogue allows agent $a(r1, a1)$ to broadcast a message $m1$ to agents $a(r2, a2)$ and $a(r3, a3)$ and wait for a reply from both of these. Agent $a(r2, a2)$ is expected to reply with message $m2$ to agent $a(r1, a1)$. Agent $a(r3, a3)$ is expected to do the same. Assuming that each agent operates sequentially (an assumption that is not essential to this paper but helpful for the purpose of example) the sets of possible dialogue sequences we wish to allow for the three agents in this example are as given below, where $M_o \;\Rightarrow\; A_r$ denotes a message, M_o, sent to agent A_r and $M_i \;\Leftarrow\; A_s$ denotes a message, M_i, received from agent A_s.

$$\textbf{For } a(r1,a1) \;:\; \left\{ \begin{array}{l} \langle\; m1 \;\Rightarrow\; a(r2,a2),\; m1 \;\Rightarrow\; a(r3,a3), \\ \;\; m2 \;\Leftarrow\; a(r2,a2),\; m2 \;\Leftarrow\; a(r3,a3)\rangle, \\ \langle\; m1 \;\Rightarrow\; a(r2,a2),\; m1 \;\Rightarrow\; a(r3,a3), \\ \;\; m2 \;\Leftarrow\; a(r3,a3),\; m2 \;\Leftarrow\; a(r2,a2)\rangle, \\ \langle\; m1 \;\Rightarrow\; a(r3,a3),\; m1 \;\Rightarrow\; a(r2,a2), \\ \;\; m2 \;\Leftarrow\; a(r2,a2),\; m2 \;\Leftarrow\; a(r3,a3)\rangle, \\ \langle\; m1 \;\Rightarrow\; a(r3,a3),\; m1 \;\Rightarrow\; a(r2,a2), \\ \;\; m2 \;\Leftarrow\; a(r3,a3),\; m2 \;\Leftarrow\; a(r2,a2)\rangle \end{array} \right\}$$

$$\textbf{For } a(r2,a2) \;:\; \{\langle m1 \;\Leftarrow\; a(r1,a1),\; m2 \;\Rightarrow\; a(r1,a1)\rangle\}$$

$$\textbf{For } a(r3,a3) \;:\; \{\langle m1 \;\Leftarrow\; a(r1,a1),\; m2 \;\Rightarrow\; a(r1,a1)\rangle\}$$

We can specify this dialogue using a notation, similar in style to a process algebra, to describe the permitted message passing behaviour of each agent. Each agent, A, is defined by a term, $A :: D$, where D describes the messages it is allowed to send. D can be constructed using the operators: $D_i \;\; then \;\; D_j$ (requiring D_i to be satisfied before D_j), $D_i \;\; or \;\; D_j$ (requiring a choice between D_i or D_j) or $D_i \;\; par \;\; D_j$ (requiring the agent to wait until D_i and D_j both are satisfied). A specification using this notation for our example is the following set of three clauses:

$$\begin{array}{l} a(r1,a1) :: \;\; (m1 \;\Rightarrow\; a(r2,a2)\; par\; m1 \;\Rightarrow\; a(r3,a3))\; then \\ \qquad\qquad\quad (m2 \;\Leftarrow\; a(r2,a2)\; par\; m2 \;\Leftarrow\; a(r3,a3)) \\ a(r2,a2) :: m1 \;\Leftarrow\; a(r1,a1)\; then\; m2 \;\Rightarrow\; a(r1,a1) \\ a(r3,a3) :: m1 \;\Leftarrow\; a(r1,a1)\; then\; m2 \;\Rightarrow\; a(r1,a1) \end{array}$$

We refer to this definition of the message passing behaviour of the dialogue as the *dialogue framework*. Its syntax is as follows, where $Term$ is a structured term and $Constant$ is a constant symbol assumed to be unique when identifying each agent:

$$\begin{array}{rcl} Framework & := & \{Clause, \ldots\} \\ Clause & := & Agent :: Def \\ Agent & := & a(Type, Id) \\ Def & := & Agent \mid Message \mid Def\; then\; Def \mid \\ & & Def\; or\; Def \mid Def\; par\; Def \\ Message & := & M \;\Rightarrow\; Agent \mid M \;\Rightarrow\; Agent \leftarrow C \mid \\ & & M \;\Leftarrow\; Agent \mid C \leftarrow M \;\Leftarrow\; Agent \\ C & := & Term \mid C \wedge C \mid C \vee C \\ Type & := & Term \\ Id & := & Constant \\ M & := & Term \end{array}$$

As we have seen, dialogue framework defines a space of possible dialogues. This is larger than the space we typically wish to allow in practice, since dialogues normally assume constraints on the circumstances under which messages are sent or received. These constraints are of two kinds: proaction constraints and reaction constraints. The *proaction constraints* in the protocol define the circumstances under which a message allowed by the dialogue framework is allowed to be sent. The framework makes no assumption about the mechanism for deciding whether these constraints hold in the current state of the agent, this being a matter for the engineer of that agent. Each constraint is of the form:

$$M \;\Rightarrow\; A_r \;\leftarrow\; C_p \tag{1}$$

where A and A_r are agent descriptors (of the form $a(Type, Id)$); M is a message sent by A addressed to A_r; and C_p is the condition for sending the message (either empty or a conjunction of sub-conditions which should hold in A). In our earlier example, to constrain agent $a(r1, a1)$ to send message $m1$ to agent $a(r2, a2)$ when condition c_1 holds in $a(r1, a1)$ we would define the proaction constraint: $m1 \;\Rightarrow\; a(r2, a2) \;\leftarrow\; c_1$.

Reaction constraints in the protocol define what should be true in an agent following receipt of a message allowed by the dialogue framework. As for proaction constraints, the protocol makes no assumption about the mechanism for ensuring that these constraints hold in the current state of the agent. Each constraint is of the form:

$$C_r \;\leftarrow\; M \;\Leftarrow\; A_s \tag{2}$$

where A and A_s are agent descriptors (of the form $a(Type, Id)$); M is a message sent by A_s and received by A; and C_r is the reaction upon receiving the message (either null or a conjunction of sub-conditions which A should be able to satisfy). In our earlier example, to constrain agent $a(r2, a2)$ to receive message $m1$ from agent $a(r1, a1)$ when condition c_2 holds in $a(r2, a2)$ we would define the proaction constraint: $c_2 \;\leftarrow\; m1 \;\Leftarrow\; a(r1, a1)$.

Agent dialogues may also assume *common knowledge*, either as an inherent part of the dialogue or generated by agents in the course of a dialogue. For example in an auction it might be assumed that the reserve price of an item is knowledge shared between agents. This knowledge could be

expressed in any form, as long as it can be processed reliably by all appropriate agents, but for the purposes of this discussion it is assumed that common knowledge is described as a set of axioms in first order predicate logic.

As a dialogue protocol is shared among a group of agents it is essential that each agent when presented with a message from that protocol can retrieve the state of the dialogue relevant to it and to that message. This is done by retaining (separately) in the protocol the instances of dialogue clauses used by each agent participating in the dialogue. The principle of this is similar to unfolding in the transformation of logic programs, where we can take a clause; find a sub-goal satisfiable by other clauses; then replace this with subgoal with the subgoals of that clause. In our case we have two such "unfolding" operations: when we can establish that an appropriate message has been sent or received (in which case we mark that part of the dialogue clause as closed); or when we can extend the protocol by unfolding on one of the dialogue framework clauses. This is described in detail in a later section of this paper. We shall refer to the set of clauses thus constructed as the *dialogue state*.

Pulling all the above elements together, we describe a dialogue protocol as the term:

$$protocol(S, F, C_p, C_r, K) \tag{3}$$

where S is the dialogue state; F is the dialogue framework (a set of dialogue clauses); C_p is a set of proaction constraints; C_r is a set of reaction constraints; and K is a set of axioms defining common knowledge assumed among the agents.

Using Distributed Protocols for Coordination

The practical value of the form of protocol we have described is that it can be used by one agent to tell another agent (hitherto unaware of the first agent or its dialogue conventions) precisely how it expects to converse with it. This can be achieved with only two engineering commitments on the part of the designers of individual agents. First, the proaction and reaction constraints associated with the dialogue must be in an ontology recognised (possibly via translation) by the agent. This is a requirement for any form of meaningful knowledge interchange between inference systems so we do not expand on this issue here. Second, the agent must be able to interpret the current state of the dialogue by inspecting the protocol sent to it. The remainder of this section shows how this is done.

To enable an agent to conform to a dialogue protocol it is necessary to supply it with a way of unpacking any protocol it receives; finding the next moves that it is permitted to take; and updating the state of the protocol to describe the new state of dialogue. There are many ways of doing this but perhaps the most elegant way is by applying rewrite rules to expand the dialogue state. This works as follows:

- An agent receives from some other agent a message with an attached protocol, $\mathcal{P}$, of the form $protocol(S, F, C_p, C_r, K)$ (as defined in expression 3). The message is added to the set of messages currently under consideration by the agent - giving the message set M_i.
- The agent extracts from $\mathcal{P}$ the dialogue clause, C_i, determining its part of the dialogue.
- The rewrite rules of Figure 1 are applied to give an expansion of C_i in terms of protocol $\mathcal{P}$ in response to the set of received messages, M_i, produces: a new dialogue clause C_n; an output message set O_n and remaining unprocessed messages M_n (a subset of M_i). These are produced by applying the protocol rewrite rules above exhaustively to produce the sequence:

$$\langle C_i \xrightarrow{M_i, M_{i+1}, \mathcal{P}, O_i} C_{i+1}, \ldots, C_{n-1} \xrightarrow{M_{n-1}, M_n, \mathcal{P}, O_n} C_n \rangle$$

- The agent's original clause, C_i, is then replaced in $\mathcal{P}$ by C_n to produce the new protocol, $\mathcal{P}_n$.
- The agent can then send the messages in set O_n, each accompanied by a copy of the new protocol $\mathcal{P}_n$.

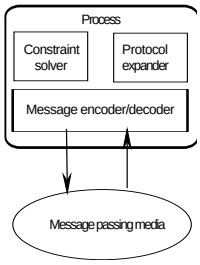

Figure 2: Basic agent architecture

Deployment of Protocols

The coordination of dialogues between agents accounts for a substantial part of the design complexity of a multi-agent system, so by standardising this we can reduce the complexity of design for individual agents in a collaboration. This section demonstrates this principle by showing how the protocol and the expansion mechanism of Figure 1 can be used to provide a system of coordination involving very little commitment to the design of each of the participating agents. We shall use the simple architecture for agent construction shown in Figure 2. This shows the three required constituents of an agent process:

- A message encoder/decoder for receiving and transmitting messages from whatever message passing media are being used to transport messages between agents.
- A protocol expander that decides how to expand a protocol recieved via a message.
- A constraint solver capable of deciding whether constraints passed to it by the protocol expander are satisfiable.

We begin with an informal statement of a collaboration scenario. We then describe a protocol for this sort of activity. Finally, we describe how agents are coordinated using the protocol.

$$
\begin{array}{lll}
A :: B \xrightarrow{M_i, M_o, \mathcal{P}, O} A :: E & if & B \xrightarrow{M_i, M_o, \mathcal{P}, O} E \\
A_1 \; or \; A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} E & if & \neg closed(A_2) \wedge A_1 \xrightarrow{M_i, M_o, \mathcal{P}, O} E \\
A_1 \; or \; A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} E & if & \neg closed(A_1) \wedge A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} E \\
A_1 \; then \; A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} E \; then \; A_2 & if & A_1 \xrightarrow{M_i, M_o, \mathcal{P}, O} E \\
A_1 \; then \; A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} A_1 \; then \; E & if & closed(A_1) \wedge A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} E \\
A_1 \; par \; A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O_1 \cup O_2} E_1 \; par \; E_2 & if & A_1 \xrightarrow{M_i, M_n, \mathcal{P}, O_1} E_1 \wedge A_2 \xrightarrow{M_n, M_o, \mathcal{P}, O_2} E_2 \\
M \Leftarrow A_s \xrightarrow{M_i, M_i - \{M \Leftarrow A_s\}, \mathcal{P}, \emptyset} c(M \Leftarrow A) & if & (M \Leftarrow A_s) \in M_i \wedge receive(\mathcal{P}, M) \\
M \Rightarrow A \xrightarrow{M_i, M_o, \mathcal{P}, \{M \Rightarrow A\}} c(M \Rightarrow A) & if & send(\mathcal{P}, M) \\
A \xrightarrow{M_i, M_o, \mathcal{P}, \emptyset} A :: B & if & clause(\mathcal{P}, A :: B, C) \wedge satisfy(C)
\end{array}
$$

- Any marked term ($c(Term)$) is closed.
- $A \; or \; B$ is closed if either A or B is closed.
- $A \; then \; B$ is closed if both A and B are closed.
- $A \; par \; B$ is closed if both A and B are closed.
- $A :: B$ is closed if B is closed.

Figure 1: Rewrite rules for expansion of a protocol clause

A Scenario

The scenario enacted in the remainder of this section is given below. It is an instance of the coordination problem discussed in the introduction to this paper.

Scenario: Dave Robertson decides to stage a workshop on the topic of agency, to take place on 20th or 21st February 2003. Using his agent system (with identifier `dr`), he contacts an agent identified as `wh` run by Wendy Hall, who runs a headhunting service for finding the current best people in given research areas. Wendy sends Dave a list of the two most prominent agency researchers in the UK: Nick Jennings and Mike Wooldridge. Dave then contacts Derek Sleeman's service (identifier `ds`) for finding the agents associated with each of these two people - the results being `nj` and `mw`. Dave then contacts each of these agents to find out whether Nick and Mike can make either of the two available days. It turns out they can both manage the 20th so this day is chosen for the workshop.

We assume that the individual agents above are capable of satisfying the following constraints:

- `dr` knows which times are suitable for him.
- `wh` can chose people appropriate to a given research theme.
- `ds` can find the likely location of an agent for a given person.
- `nj` and `mw` can decide whether or not the people they represent are available on a given day.

The Protocol

Clauses 4 to 10 below describe a protocol for dealing with the coordination scenario given above. Note the use of recursion in clauses 5, 6 and 7 to allow collaboration to be specified over chains of interaction determined by the arguments supplied to the roles of participants - with these arguments being obtained as a consequence of earlier interactions.

An organiser, O, for a workshop on topic A using headhunter H and finder F will send a message to H asking for the best people on that topic; then receive a message nominating the set, $\mathcal{S}$, of best people; then will assume the role of a locator that finds the set, $\mathcal{L}$, of Web locations for the diaries of each person in $\mathcal{P}$; then becomes a time coordinator that establishes a mutually convenient workshop time, T_w, from the set $\mathcal{T}$ of times suitable for the organiser.

$$
\begin{array}{l}
a(organiser(A, H, F), O) :: \\
\quad ask(best\_people(A)) \Rightarrow a(headhunter, H) \; then \\
\quad inform(best\_people(A, \mathcal{S})) \Leftarrow a(headhunter, H) \; then \\
\quad a(locator(F, \mathcal{S}, \mathcal{L}), O) \; then \\
\quad a(time\_coordinator(\mathcal{S}, \mathcal{L}, A, \mathcal{T}, T_w), O) \leftarrow times(\mathcal{T})
\end{array} \tag{4}
$$

A locator using finder F to obtain the Web locations, $\mathcal{L}$, of the people in set $\mathcal{S}$ will recursively go through the set taking each person, X_p, and asking F for the appropriate location, X_l. Note how the recursion is defined through the change of role: from $a(locator(F, \mathcal{S}, \mathcal{L}), L)$ to $a(locator(F, \mathcal{S}_r, \mathcal{L}_r), L)$ in the step case and terminating when $\mathcal{S}$ contains no elements (represented by the $[]$ symbol).

$$
\begin{array}{l}
a(locator(F, \mathcal{S}, \mathcal{L}), L) :: \\
\quad \left(\begin{array}{l}
ask(locate(X_p)) \Rightarrow a(finder, F) \leftarrow \\
\qquad \mathcal{S} = [X_p | \mathcal{S}_r] \wedge \mathcal{L} = [X_l | \mathcal{L}_r] \; then \\
inform(located(X_p, X_l)) \Leftarrow a(finder, F) \; then \\
a(locator(F, \mathcal{S}_r, \mathcal{L}_r), L)
\end{array} \right) \\
\quad or \\
\quad null \leftarrow \mathcal{S} = [] \wedge \mathcal{L} = []
\end{array} \tag{5}
$$

A time coordinator seeking to establish the time, T_w, for a workshop will take the first time, T, from the set of times

available (leaving remaining times, $\mathcal{T}_r$) and become a proposer for time T to the set of people, $\mathcal{S}$, at locations, $\mathcal{L}$. If the result, R, of the time proposal is ok then T is the chosen time (T_w) - otherwise if the result is not_ok the time coordinator continues with the remaining times, $\mathcal{T}_r$.

$$\begin{array}{l} a(time\_coordinator(\mathcal{S},\mathcal{L},A,[T|\mathcal{T}_r],T_w),X) :: \\ \quad a(time\_proposer(\mathcal{S},\mathcal{L},A,T,R),X)\ \ then \\ \quad \left(\begin{array}{c} null\ \leftarrow\ (R = ok \wedge T_w = T) \\ or \\ a(time\_coordinator(\mathcal{S},\mathcal{L},A,\mathcal{T}_r,T_w),X)\ \leftarrow \\ R = not\_ok \end{array}\right) \end{array} \tag{6}$$

A time proposer asks each person, P, from the set of nominated people whether they are available by sending a message to their location, L. If the reply from L confirms availability then the time proposer recurses through the remainder of the people, $\mathcal{S}_r$ to check their availability. If the reply disconfirms availability then the result, R, is not_ok and no further people receive this proposed time.

$$\begin{array}{l} a(time\_proposer([P|\mathcal{S}_r],[L|\mathcal{L}_r],A,T,R),X) :: \\ \quad ask(available(T))\ \Rightarrow\ a(P,L)\ \ then \\ \qquad \left(\begin{array}{c} inform(available(T))\ \Leftarrow\ a(P,L)\ \ then \\ a(time\_proposer(\mathcal{S}_r,\mathcal{L}_r,A,T,R),X) \end{array}\right) \\ \qquad\qquad or \\ \quad R = not\_ok\ \leftarrow\ inform(unavailable(T))\ \Leftarrow\ a(P,L) \end{array} \tag{7}$$

A headhunter receives a message asking for the best people on some research topic, A, and replies with a set $\mathcal{S}$ of best people if it can choose these best people.

$$\begin{array}{l} a(headhunter,H) :: \\ \quad ask(best\_people(A))\ \Leftarrow\ a(organiser(T,H,F),O)\ \ then \\ \quad inform(best\_people(A,\mathcal{S}))\ \Rightarrow\ a(organiser(T,H,F),O) \\ \qquad \leftarrow\ choose\_persons(A,\mathcal{S}) \end{array} \tag{8}$$

A finder receives a message asking for the Web location of a person named X_p and returns a message giving the location X_l if a likely location can be found.

$$\begin{array}{l} a(finder,P) :: \\ \quad ask(locate(X_p))\ \Leftarrow\ a(Role,L)\ \ then \\ \quad inform(located(X_p,X_l))\ \Rightarrow\ a(Role,L)\ \leftarrow \\ \qquad likely\_location(X_p,X_l)\ \ then \\ \quad a(finder,P) \end{array} \tag{9}$$

A person is asked about availability at time T and replies to confirm or disconfirm.

$$\begin{array}{l} a(person(P),X) :: \\ \quad ask(available(T))\ \Leftarrow\ Agent\ \ then \\ \quad \left(\begin{array}{c} inform(available(T))\ \Rightarrow\ Agent\ \leftarrow \\ am\_available(T) \\ or \\ inform(unavailable(T))\ \Rightarrow\ Agent\ \leftarrow \\ am\_unavailable(T) \end{array}\right) \end{array} \tag{10}$$

Connecting Services

Clauses 4 to 10 describe a protocol for our workshop scenario. Any service is able to use this protocol, provided that it has the abilities described at the beginning of the section on protocol deployment: it can send/receive messages carrying the protocol; it can expand protocols; and it can solve those constraints attached to its role in the protocol. There currently are two implemented methods for running a confederation of services:

- Linear scripting method: The protocol and any instances of its clauses resulting from clause expansion by individual agents is sent along with each message. Intuitively, this method treats the protocol as a single "script" which carries all the state associated with the coordination. The advantage of this is that, having taken its turn in the interaction, an agent need not maintain its own record of the state of play - that will be maintained along with the protocol itself as it passes between agents - so agents involved in long-running collaborations do not need to preserve state or maintain processes corresponding to their individual roles. This method, however is limited to collaborations which can be linearised into a sequence of interactions between pairs of agents, since it does not permit two different agents to alter the state of the protocol simultaneously. In practice many forms of collaboration linearise in this way.
- Distributed process method: This method has been developed by Walton. The clauses from the protocol are sent to the participating agents, each according to the role the agent intends to play. These clauses are then used to drive the appropriate behaviours on each agent. This method does not have the linearity restriction of scripting method above but it does require some additional control notation within the protocol language (timeouts for example). It also requires individual agents to maintain the state of their role in each collaboration, rather than being able to drop and resume their place in the interaction with each sent and received message.
- Distributed scripting method: This is a combination of the two methods above. As in the scripting method, the protocol is carried with each message. Unlike the scripting method (but like the distributed process method) the clauses for individual agents are retained on those agents, rather than being resent with the protocol on messages. This keeps the notion of a script but without the linearity restriction.

Space limitations prevent a full description of the methods above being applied to the scenario of this section. It is however, possible for the reader to work through clauses 4 to 10 of the protocol. Start with the workshop organiser (the initiator of the collaboration) and match the the role $agentorganiser(agency, wh, ds)dr$ to the role of clause 4; then apply the expansion of Figure 1. This will allow the message $ask(best\_people(agency))$ to be sent to $a(headhunter, wh)$ (that is, the agent wh in the role of $headhunter$). When wh receives this message it can lookup the clause for this role (clause 8); expand it; and send the ap-

Figure 3: Window for initiating dialogue protocol

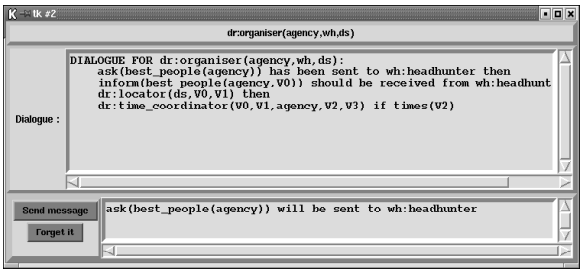

Figure 4: Protocol at point of initiation

propriate message. The coordination continues in a similar way, involving the other agents in the workshop scenario.

It is straightforward to implement a basic, generic user interface to allow human interaction in the execution of the protocol described above. We demonstrate one such interface below. Notice that we do not claim this to be the appropriate interface for deployment (in practice deployment would involve several, bespoke interfaces tuned to the different agents involved). Our only interest is to show how interfaces, in general, connect to protocols.

Figure 3 shows the window used to initiate a dialogue. At the top of the tool is a field to allow the identity of the agent to be stipulated (in this case it is dr). Clicking the button labeled "`Identity`" loads the agent with the given identity. Below that is a list of protocols, each one accompanied by the role assumed by the initiator of that protocol. The arguments of each role appear as fields to be completed by the person using the agent. The window of Figure 3 lists two protocols: `brokering` and `workshop`. The `organiser` role for the `workshop` protocol has been instantiated with the topic of `agency`; a headhunter agent named `wh`; and a finder agent named `ds`. Clicking the button labeled `workshop` will initiate this protocol for the agent named `dr` in this `organiser` role.

Initiation of the protocol causes the window shown in Figure 4 to appear. This window shows, in the lower pane, the messages that can be sent and, in the upper pane, the consequent state of the collaborative interaction. In Figure 4 the message sent to `wh` in the role of `headhunter` asks for best people on the topic of `agency`. The consequent state of the dialogue is described by an instance of protocol clause 4 where that message has been sent. Clicking the "`Send message`" button sends the message along with the protocol.

Figure 5 shows the dialogue window for the `wh` agent after receiving the message sent in Figure 4. This sends back a

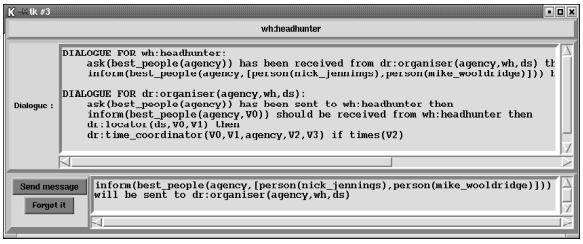

Figure 5: Protocol at first point of return

message to `dr` in the role of `organiser` informing it of the best people (according to `wh`) on the topic of `agency`. The consequent state of coordination now is described by two clauses: for `wh` in a `headhunter` role and for `dr` in the role of `organiser`.

The sequence of interactions continues in a style similar to that of Figures 3 to 5. At the conclusion of this, the instance of the organiser clause in the dialogue state is shown in Figure 6.

Distributed Dialogues and Web Service Ontologies

The protocols of this paper are intended to describe collaborations between agents by specifying message passing behaviours of each agent. This invites a comparison to recent standard markup languages for Web services - in particular the DARPA Agent Markup Language for Services (DAML-S) (Coalition 2003; McIlraith & Martin 2003). DAML-S consists of three parts: a service profile that describes properties of a service useful for choosing it (*e.g.* the types of inputs it requires; a service model describing how the process is enacted by the service with the aim of supporting service composition and execution; and a service grounding that connects the process model to the mechanisms used for inter-agent communication. A naive comparison to DAML-S might: equate our message specifications with a DAML-S service profile (*e.g.* input messages correspond to inputs in DAML-S profile); equate our message sequencing structure with a DAML-S process model; and equate whatever mechanism we choose to send and receive messages with a DAML-S service grounding. This naive comparison, however, ignores important conceptual and technical differences which make the comparison to DAML-S more subtle.

Perhaps the most important difference is in the way processes are composed. In DAML-S, the way to specify coordination between two processes is to define a larger process of which they are sub-processes, so composition is hierarchical and requires sub-processes to be enumerated at specification time. Coordination in our system is by message passing (so there is no rigid hierarchy) and it is not essential to enumerate all the players in a collaboration at specification time (since it is possible for messages or constraints evaluated at run time to determine which agents are involved in an interaction). These technical differences stem from differing conceptual views of the problem. The DAML-S view starts from the perspective of an individual service and "opens up"

$$
\begin{array}{l}
a(organiser(agency, wh, ds), dr) :: \\
\quad c(ask(best\_people(agency)) \;\Rightarrow\; a(headhunter, wh))\ then \\
\quad c(inform(best\_people(agency, \left[\begin{array}{l} person(nick\_jennings), \\ person(mike\_wooldridge) \end{array}\right])) \;\Leftarrow\; a(headhunter, wh))\ then \\
\left(\begin{array}{l}
a(locator(ds, \left[\begin{array}{l} person(nick\_jennings), \\ person(mike\_wooldridge) \end{array}\right], [nj, mw]), dr) :: \\
\quad c(ask(locate(person(nick\_jennings))) \;\Rightarrow\; a(finder, ds))\ then \\
\quad c(inform(located(person(nick\_jennings), nj)) \;\Leftarrow\; a(finder, ds))\ then \\
\left(\begin{array}{l}
a(locator(ds, [person(mike\_wooldridge)], [mw]), dr) :: \\
\quad c(ask(locate(person(mike\_wooldridge))) \;\Rightarrow\; a(finder, ds))\ then \\
\quad c(inform(located(person(mike\_wooldridge), mw)) \;\Leftarrow\; a(finder, ds))\ then \\
a(locator(ds, [], []), dr)
\end{array}\right)
\end{array}\right)\ then \\
(a(time\_coordinator(\left[\begin{array}{l} person(nick\_jennings), \\ person(mike\_wooldridge) \end{array}\right], [nj, mw], agency, \left[\begin{array}{l} day(20, 2, 2003), \\ day(21, 2, 2003) \end{array}\right], day(20, 2, 2003)), dr) :: \\
\left(\begin{array}{l}
a(time\_proposer(\left[\begin{array}{l} person(nick\_jennings), \\ person(mike\_wooldridge) \end{array}\right], [nj, mw], agency, day(20, 2, 2003), ok), dr) :: \\
\quad c(ask(available(day(20, 2, 2003))) \;\Rightarrow\; a(person(nick\_jennings), nj))\ then \\
\quad c(inform(available(day(20, 2, 2003))) \;\Leftarrow\; a(person(nick\_jennings), nj))\ then \\
\left(\begin{array}{l}
a(time\_proposer([person(mike\_wooldridge)], [mw], agency, day(20, 2, 2003), ok), dr) :: \\
\quad c(ask(available(day(20, 2, 2003))) \;\Rightarrow\; a(person(mike\_wooldridge), mw))\ then \\
\quad inform(available(day(20, 2, 2003))) \;\Leftarrow\; a(person(mike\_wooldridge), mw)\ then \\
\quad a(time\_proposer([], [], agency, day(20, 2, 2003), ok), dr)
\end{array}\right)
\end{array}\right)
\end{array}
$$

Figure 6: Concluding state of dialogue for organiser

that service in a controlled way, for example by describing the process it uses and the sub-processes it requires. Our view starts from the model of interaction between services and keeps individual services as "closed" as possible, in the sense that we assume nothing about their internal processes other than the ability to process the protocol. Hence DAML-S has a more expressive language for describing internal structure of services while ours is a more expressive language for specifying interactions between services.

A second fundamental difference between these approaches is the emphasis in DAML-S on specification for automatic service discovery. This is crucial to semantic web efforts because of the intended scale and openness of such systems. There is at present no means to attach to our protocols properties analogous to those of the DAML-S service profile. We can describe both specific and general-purpose brokering interactions but this is different from representing properties of the protocols themselves. It would be useful to extend our language in this way.

A final difference from DAML-S is that DAML-S a restricted, typed language in which the various features of a Web service are described as classes and their properties. Our language is not typed. It does, however, allow variables (in the normal sense in first order logic) to be used in the style standard in logic programming. This makes it easy for those accustomed to logic programming (or similar methods) to define protocols where information is "threaded" through an interaction via variables carried in roles or messages and to share variables between messages and constraints. DAML-S provides no such functionality at present, although it is possible to imagine this appearing in an extended version of the language. In this aspect, DAML-S feels more like a type declaration language while ours feels more like a logic programming language.

Conclusions and Future Work

This paper describes a calculus that is comparatively lightweight (in the sense that it can readily by those trained in declarative programming which is a standard skill for knowledge engineers) yet can be used to represent complex coordination between Web services described in an agent style. It can be deployed with different styles of computational model to synchronise dialogues between agents operating in complex coordination scenarios in an asynchronous environment. It also supports simulation in both sequential and asynchronous environments, although space limitations prohibit discussion of this in our current paper. The ability to coordinate dialogue in this way, and in particular to add constraints within the protocol, allows us to provide checks within the protocol according to the way in which we intend the dialogue to be understood. This is helpful in ensuring adherence to ontologies for communication - although of course it cannot entirely guarantee adherence to the human "meaning" of the ontologies we use, it allows ontological constraints to be made more explicit during coordination between agents.

The protocol requires a particular syntax for constraints but does not not assume a particular style of constraint solver, allowing different solvers to be used. Although not described in this paper, we have integrated a finite domain constraint solver with the the clause expansion method described in this paper - the only significant engineering difference being in the treatment of variable bindings during multi-agent interaction. There remains, however, much work still to do:

- Our work has been oriented to logic rather than to the languages emerging as *de facto* standards for semantic web services. We consider this an advantage from the point of view of research because logic is comparatively stable while web service languages are in flux. There is, however, a practical need to connect to standard interchange languages and some efforts to do this are underway. Walton, for example, has an XML representation for our protocols and has written a Java interpreter for them.
- Our current protocols are not adaptive. It is assumed that the agents involved in a collaboration will follow the protocol given and will not opportunistically change it during collaboration (for instance by taking out a section of the protocol and replacing it with another). Although it is easy to allow agents to perform this sort of adaptation it is difficult to ensure that such adaptations will always preserve the purpose for which the protocol was intended. McGinnis is studying ways of allowing controlled adaptation.
- Protocols may fail (for example when an agent refuses to respond or provides an information which breaks constraints imposed by other agents). If we want systems that are tolerant to this then there has to be a way of recovering from failure - in the simplest case by exploring new parts of the protocol; in more complex cases by patching the protocol to address the failure. Osman has produced an algorithm for the former of these two cases.

References

Coalition, D. S. 2003. DAML-S: Semantic markup for web services. Technical overview, DARPA Agent Markup Language group. URL: www.daml.org/services/.

Decker, K.; Pannu, A.; Sycara, K.; and Williamson, M. 1997. Designing behaviors for information agents. In *Proceedings of the First International Conference on Autonomous Agents*.

Giampapa, J., and Sycara, K. 2002. Team-oriented agent coordination in the retsina multi-agent system. Technical Report CMU-RI-TR-02-34, Robotics Institute, Carnegie Mellon University.

Kagal, L.; Finin, T.; and Joshi, A. 2003. A policy language for pervasive systems. In *Fourth IEEE International Workshop on Policies for Distributed Systems and Networks*.

McIlraith, S., and Martin, D. 2003. Bringing semantics to web services. *IEEE Intelligent Systems* 18(1):90–93.

McIlraith, S., and Son, T. 2002. Adapting golog for composition of semantic web services. In *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning*, 482–493.

Sheshagiri, M.; desJardins, M.; and Finin, T. 2003. A planner for composing services described in daml-s. In *International Conference on Automated Planning and Scheduling*.