

Multi-agent Coordination as Distributed Logic Programming

David Robertson

Informatics, University of Edinburgh

Abstract. A novel style of multi-agent system specification and deployment is described, in which familiar methods from computational logic are re-interpreted to a new context. One view of multi-agent system design is that coordination is achieved via an interaction model in which participating agents assume roles constrained by the social norms of their shared task; the state of the interaction reflecting the ways these constraints are mutually satisfied within some system for synchronisation that is open and distributed. We show how to harness a process calculus; constraint solving; unfolding and meta-variables for this purpose and discuss the advantages of these methods over traditional approaches.

1 Introduction

We are interested in the specification and deployment of multi-agent systems, which we define as systems of distributed components in which components can usefully be viewed as autonomous problem solvers that must collaborate in order to perform complex tasks. There are numerous difficulties in constructing such systems, beyond the normal difficulties associated with building individual agents. These specific issues include the following:

- Maintaining the conformity of interaction necessary to perform a shared task reliably without sacrificing the autonomy of each agent. One solution to this problem is to define a model of the interaction with which agents interact (via some appropriate controller) in order to perform a given task. Control and state information essential to the task resides in that model, minimising the impact on individual agents.
- Ensuring that when necessary an agent can determine its current role and obligations in the interaction, and not requiring that any agent monitor the interaction when that is unnecessary.
- Allowing constraints on variables established locally by agents to be shared by other agents if appropriate and for those others to be able to adapt these constraints.

Although it may seem surprising that standard methods from computational logic can be applied simply and directly to these sorts of issues, we shall explain how this can be done - in the process establishing a new niche for such methods. To emphasise the parallels between issue and method we write each section title in the form $I = M$, where I is an issue for multi-agent system design and M is the corresponding logic programming method. The methods taken together provide a basic formal approach to

designing and (in appropriate circumstances) deploying multi-agent systems. Section 2 describes key aspects of earlier research by others that relate to our approach.

In Section 9 we shall demonstrate how these methods work together on a short but (by current standards) complex scenario. The scenario, which concerns ordering and configuration of a computer, is as follows:

An internet-based agent acting on behalf of a customer wants to buy a computer but doesn't know how to interact with other agents to achieve this so it contacts a service broker. The broker supplies the customer agent with the necessary interaction information. The customer agent then has a dialogue with the given computer vendor in which the various configuration options and pricing constraints are reconciled before a purchase finally is made.

To deal with this scenario we shall introduce the basic components of a Lightweight Coordination Calculus (LCC). In Section 3 we introduce the basic calculus. We then, in Section 4, show how this permits multi-agent social norms to be controlled by mutual constraints on variables determining their message passing behaviour. Sections 5 and 6 show how traditional transformation methods may be applied to execute LCC specifications in distributed environments. Section 7 describes how finite domain constraint solving can be used to make protocols less brittle. Section 8 shows how LCC protocols are suited to brokering of interactions between agents - a key issue for open systems like the Web.

2 Background

Although we use the popular term "agent" in our research, an interest in coordination of processes in open, distributed environments extends more broadly across computer science. Much of the topical interest has come from burgeoning technological efforts - in particular the semantic web and multi-agent systems. There has, however, been long term interest in the logic programming community.

In [1] LCC is described from the perspective of those wishing to coordinate semantic web services, where the point of contact for LCC is the process specification component of (rapidly evolving) service specification languages. Seen from this perspective, the closest existing research is from those using temporal logics to specify different aspects of required service behaviours: for individual services (*e.g.* [2]); shared models for coordinating services (*e.g.* [3]) or the process of composing services (*e.g.* [4, 5]). In [6] LCC is presented as a compact way to describe electronic institutions of the sort recently made popular in the agent community through use of finite state machine models of coordination [7, 8].

In logic programming there is a history of interest in parallel computation and consequently an involvement in coordinating the distributed computations in multi-agent systems. One form of involvement is to invent a form of logic programming language that gives an overall architecture for coordination. The Go! language [9], for example, provides a multi-threaded environment in which agents may be coordinated via a shared memory store of beliefs, desires and intentions. In contrast to such languages, LCC requires nothing more than a traditional Prolog system to achieve its form of coordination

- the primary interest being in using specifications written in LCC to coordinate processes that may individually be in different environments. Perhaps closer to LCC is the work being done on modelling multi-agent coordination using logic programs, for example in [10] where the Event Calculus is used to specify and analyse social constraints between agents (the motivation for this being similar to that of [4]). A translator has been written from LCC to a version of the Event Calculus, although most of our current research on verification of LCC protocols operates more directly from the LCC notation [11].

3 Interaction Model = Process Calculus

LCC borrows the notion of role from agent systems that enforce social norms but reinterprets this in a process calculus. Process calculi have been used before to specify social norms (see for example [7]) but LCC is, to our knowledge, the first to be used directly in computation for multi-agent systems. Social norms in LCC are expressed as the message passing behaviours associated with roles. The most basic behaviours are to send or receive messages, where sending a message may be conditional on satisfying a constraint and receiving a message may imply constraints on the agent accepting it. The choice of constraint language depends on the constraint solvers used and we shall discuss this more fully in subsequent sections. More complex behaviours are specified using the connectives *then*, *or* and *par* for sequence, choice and parallelisation respectively. A set of such behavioural clauses specifies the message passing behaviour expected of a social norm. We refer to this as the interaction framework. Its syntax is as shown in Figure 1.

$$\begin{aligned}
 \textit{Framework} &::= \{ \textit{Clause}, \dots \} \\
 \textit{Clause} &::= \textit{Agent} :: \textit{Dn} \\
 \textit{Agent} &::= a(\textit{Type}, \textit{Id}) \\
 \textit{Dn} &::= \textit{Agent} \mid \textit{Message} \mid \textit{Dn then Dn} \mid \textit{Dn or Dn} \mid \textit{Dn par Dn} \mid \textit{null} \leftarrow C \\
 \textit{Message} &::= M \Rightarrow \textit{Agent} \mid M \Rightarrow \textit{Agent} \leftarrow C \mid M \leftarrow \textit{Agent} \mid C \leftarrow M \leftarrow \textit{Agent} \\
 C &::= \textit{Term} \mid C \wedge C \mid C \vee C \\
 \textit{Type} &::= \textit{Term} \\
 M &::= \textit{Term}
 \end{aligned}$$

Where *null* denotes an event which does not involve message passing; *Term* is a structured term in Prolog syntax and *Id* is either a variable or a unique identifier for the agent.

Fig. 1. Syntax of LCC dialogue framework

LCC is not the first specification language to describe social norms, although it is believed to be the first such logic programming language. Conversation policy languages (e.g. [12]) are similar to LCC in the sense that they apply constraints to the behaviours permitted by agents, thus giving a safe envelope of operation for agents. In Section 7

we shall consider this issue in more detail. LCC is also temporal, since it imposes partial orderings on message passing between agents, so in this respect it resembles efforts in the semantic web service arena to represent individual service behaviours (*e.g.* [2]); shared models for coordinating services (*e.g.* [3]) or the process of composing services (*e.g.* [4, 5]). A third view of LCC is as a way of describing state change during multi-agent interaction. In this aspect it resembles systems like Islander [7, 8] that represent social norms as finite state systems in which agents “move” between states according to given constraints. In Section 5 we describe a different view of state change but, first, we consider briefly the interplay in LCC between social norms and constraints.

4 Social Norms = Mutual Constraints

The LCC language ensures coherence of interaction between agents by imposing constraints relating to the messages they send and receive in their chosen roles. The clauses of a protocol are arranged so that, although the constraints on each role are independent of others, the ensemble of clauses operates to give the desired overall behaviour. For instance, the LCC protocol:

$$\begin{aligned} a(r1, A_1) :: offer(X) \Rightarrow a(r2, A_2) \leftarrow p(X) \text{ then } accept(X) \Leftarrow a(r2, A_2) \\ a(r2, A_2) :: offer(X) \Leftarrow a(r1, A_1) \text{ then } accept(X) \Rightarrow a(r1, A_1) \leftarrow q(X) \end{aligned} \quad (1)$$

places two constraints on the variable X : the first ($p(X)$) is a condition on the agent in role $r1$ sending the message $offer(X)$ and the second ($q(X)$) is a condition on the agent in role $r2$ sending message $accept(X)$ in reply. By (separately) satisfying $p(X)$ and $q(X)$ the agents mutually constrain the variable X .

How does each agent satisfy constraints? LCC allows two options:

- Internally according to whatever knowledge and reasoning strategies it possesses. This is the normal assumption in most multi-agent systems, yet it is not always ideal. In particular we sometimes would like to use knowledge specifically for a social interaction but not require an agent to internalise it (*e.g.* if that knowledge might be inconsistent with an agent’s own beliefs). In such cases LCC offers a second option:
- Externally using a set of Horn clauses defining common knowledge assumed for the purposes of the interaction. Like the LCC protocols themselves, this common knowledge is passed between agents along with messages during interaction (see Section 6) so it is ephemeral - lasting only as long as the interaction.

In Section 7 we consider constraint satisfaction in more detail but first we describe the basic mechanism provided in LCC for changing the state of the interaction during message passing.

5 Interaction State Change = Unfolding

In multi-agent systems with predictable behaviours we must be able to reason about state change. In Section 6 we shall discuss the distinction between state that is private

to individual agents and state associated with their interaction. Before this we describe how the state of the interaction from the perspective of an individual agent's role may change. The mechanism for performing this basic operation is a form of unfolding familiar from logic program transformation.

Unfolding of a Horn clause with respect to a set of Horn clauses is done by selecting a unit goal in the body of that clause and matching it to the head of a copy of an appropriate clause in the set. The body of that matched clause then replaces the unit goal in the original clause. State change in LCC uses a similar method.

Recall that the behaviour of an agent in a given role is determined by the appropriate LCC clause. Figure 2 gives a set of rewrite rules that are applied to give an unfolding of a LCC clause C_i in terms of protocol \mathcal{P} in response to the set of received messages, M_i , producing: a new LCC 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 from i to n :

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

We refer to the rewritten clause, C_n , as an expansion of the original clause, C_i and write $expanded(C_i, M_i, \mathcal{P}, C_n, O_n)$ when this expansion is performed. In the next section we describe how this basic expansion method is used for multi-agent coordination.

6 Coordination = Distributed Clauses

To coordinate an interaction between multiple agents, each agent must know its constraints on when to send and receive messages. We want this to have as low an impact as possible on the engineering of agents so the mechanism for achieving this should be modular, acting as an intermediary between the agent and the medium used to transmit messages. The module we supply has the following elements:

- A message encoder/decoder for receiving and transmitting messages from whatever message passing media are being used to transport messages between agents. For example, if the JADE platform is being used for inter-agent communication then the encoder/decoder must be able to read JADE messages (which use the FIPA-ACL performative language) and translate these into LCC protocol expressions; similarly for other platforms.
- A protocol expander that decides how to expand a protocol received via a message. This was described in Section 5.
- A constraint solver capable of deciding whether constraints passed to it by the protocol expander are satisfiable. This was introduced in Section 4 and is extended in Section 7.

Given the above, expression 2 defines how an agent can react to a received message M addressed to its identifier, X , in the role R and carrying protocol, \mathcal{P} . \mathcal{S} is the store of LCC clauses already known to the agent, from which an appropriate clause C_i may be drawn if it has already been involved in this role or, if not, C_i may be drawn from \mathcal{P} . After expansion to C_n the clause is replaced in \mathcal{S} to give new clause store \mathcal{S}_n . The

The following ten rules define a single expansion of a clause. Full expansion of a clause is achieved through exhaustive application of these rules. Rewrite 1 (below) expands a protocol clause with head A and body B by expanding B to give a new body, E . The other nine rewrites concern the operators in the clause body. A choice operator is expanded by expanding either side, provided the other is not already closed (rewrites 2 and 3). A sequence operator is expanded by expanding the first term of the sequence or, if that is closed, expanding the next term (rewrites 4 and 5). A parallel operator expands on both sides (rewrite 6). A message matching an element of the current set of received messages, M_i , expands to a closed message if the constraint, C , attached to that message is satisfied (rewrite 7). A message sent out expands similarly (rewrite 8). A null event can be closed if the constraint associated with it can be satisfied (rewrite 9). An agent role can be expanded by finding a clause in the protocol with a head matching that role and body B - the role being expanded with that body (rewrite 10).

$$\begin{array}{ll}
A :: B \xrightarrow{M_i, M_o, \mathcal{P}, O} A :: E & \text{if } B \xrightarrow{M_i, M_o, \mathcal{P}, O} E \\
A_1 \text{ or } A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} E & \text{if } \neg \text{closed}(A_2) \wedge \\
& A_1 \xrightarrow{M_i, M_o, \mathcal{P}, O} E \\
A_1 \text{ or } A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} E & \text{if } \neg \text{closed}(A_1) \wedge \\
& A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} E \\
A_1 \text{ then } A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} E \text{ then } A_2 & \text{if } A_1 \xrightarrow{M_i, M_o, \mathcal{P}, O} E \\
A_1 \text{ then } A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} A_1 \text{ then } E & \text{if } \text{closed}(A_1) \wedge \\
& A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O} E \\
A_1 \text{ par } A_2 \xrightarrow{M_i, M_o, \mathcal{P}, O_1 \cup O_2} E_1 \text{ par } E_2 & \text{if } A_1 \xrightarrow{M_i, M_n, \mathcal{P}, O_1} E_1 \wedge \\
& A_2 \xrightarrow{M_n, M_o, \mathcal{P}, O_2} E_2 \\
C \leftarrow M \leftarrow A \xrightarrow{M_i, M_i - \{M \leftarrow A\}, \mathcal{P}, \emptyset} c(M \leftarrow A) & \text{if } (M \leftarrow A) \in M_i \wedge \\
& \text{satisfy}(C) \\
M \Rightarrow A \leftarrow C \xrightarrow{M_i, M_o, \mathcal{P}, \{M \Rightarrow A\}} c(M \Rightarrow A) & \text{if } \text{satisfied}(C) \\
\text{null} \leftarrow C \xrightarrow{M_i, M_o, \mathcal{P}, \emptyset} c(\text{null}) & \text{if } \text{satisfied}(C) \\
a(R, I) \leftarrow C \xrightarrow{M_i, M_o, \mathcal{P}, \emptyset} a(R, I) :: B & \text{if } \text{clause}(\mathcal{P}, a(R, I) :: B) \wedge \\
& \text{satisfied}(C)
\end{array}$$

A protocol term is decided to be closed, meaning that it has been covered by the preceding interaction, as follows:

$$\begin{array}{l}
\text{closed}(c(X)) \\
\text{closed}(A \text{ or } B) \leftarrow \text{closed}(A) \vee \text{closed}(B) \\
\text{closed}(A \text{ then } B) \leftarrow \text{closed}(A) \wedge \text{closed}(B) \\
\text{closed}(A \text{ par } B) \leftarrow \text{closed}(A) \wedge \text{closed}(B) \\
\text{closed}(X :: D) \leftarrow \text{closed}(D)
\end{array}$$

$\text{satisfied}(C)$ is true if C can be solved from the agent's current state of knowledge.
 $\text{satisfy}(C)$ is true if the agent's state of knowledge can be made such that C is satisfied.
 $\text{clause}(\mathcal{P}, X)$ is true if clause X appears in the dialogue framework of protocol \mathcal{P} , as defined in Figure 1.

Fig. 2. Rewrite rules for expansion of a protocol clause

resulting set of output messages, O_n can then be sent to appropriate agents via whatever message encoder is provided.

$$react(M, X, R, \mathcal{P}, \mathcal{S}, O_n, \mathcal{S}_n) \leftarrow \begin{aligned} & clause_for_role(\mathcal{P}, a(R, X), \mathcal{S}, C_i) \wedge \\ & expanded(C_i, \{M\}, \mathcal{P}, C_n, O_n) \wedge \\ & replace(C_i, C_n, \mathcal{S}, \mathcal{S}_n) \end{aligned} \quad (2)$$

$$\begin{aligned} clause_for_role(\mathcal{P}, a(R, X), \mathcal{S}, a(R, X) :: D) \leftarrow \\ a(R, X) :: D \in \mathcal{S} \vee \\ (\neg(a(R, X) :: D \in \mathcal{S}) \wedge a(R, X) :: D \in \mathcal{P}) \end{aligned} \quad (3)$$

The reactive definition above is one of a range of ways that LCC protocols may be used in coordinating distributed interactions. To date we have built two other forms of deployment mechanism:

- A Java-based mechanism, implemented by Walton, in which a thread is created for each role and these threads then control the message passing. Although this paper views LCC from a logic programming perspective, this illustrates that one does not necessarily require a Prolog interpreter to compute with it.
- A Prolog-based mechanism in which clause store, \mathcal{S} , is not resident on each agent but carried with the protocol as messages are sent. This has the advantage that it requires no clause storage on agents but it works only for interactions that are linear, in the sense that at any given time only one agent has the protocol. In Section 7 we return to this form of interaction when dealing with finite domain restrictions on variables in protocols.

LCC is intended to be as neutral to the choice of mechanism for communicating between agents so it is natural (and desirable) that several such mechanisms exist for it. It is not neutral, however, to the choice of constraint solver. In the next section we discuss in more detail the interplay between constraints and protocol.

7 Interaction Scope = Constraint Space

In previous sections we have not discussed in detail the way in which constraints stipulated in protocols are satisfied. The hooks given for this in Figure 2 (via the predicates *satisfy*(C) and *satisfied*(C)) tacitly assume a solver that would find a satisfiable instance of constraint C . It is, however, well known that satisfying instances of constraints too early can result in the wrong instance being selected. The price paid for this in standard logic programs is the computational cost of backtracking. In multi-agent interactions we do not have this option because messages sent to other agents remain sent and we cannot assume that agents having received messages are able to backtrack, since they may not be implemented in a language that supports backtracking. Hence if we rely entirely on satisfying instances of constraints our protocols are liable to be brittle.

One remedy for brittleness is to have a more sophisticated view of mutual constraints between agents (recall Section 4) in which we maintain a constraint space that

bounds the scope of the interaction. With a finite domain constraint solver, for instance, this constraint space can be described by the range constraints on all the variables in the protocol clauses expanded by participating agents. Applying this to our initial example of mutual constraints in expression 1, if the range of values permitted for X by $p(X)$ is $\{1, 2, 3\}$ while the range of values permitted for X by $q(X)$ is $\{2, 3, 4\}$ then were we to demand instances for variables in all constraints then the agent in role $r1$ might (arbitrarily) choose $p(1)$ and consequently send $offer(1)$ to the agent in role $r2$. Unfortunately, $r2$ would then need to satisfy $q(1)$ in order to reply with an acceptance and it cannot. The brittle protocol then, in the absence of a means of backtracking, can only fail. Were we instead to use a finite domain solver we would obtain the range $\{1, 2, 3\}$ for $p(X)$ and, since $q(X)$ is constraining the same variable at the time it is called, this would be reduced to $\{2, 3\}$ - a range that would be attached to the variable returned in the $accept(X)$ message. By maintaining a constraint space we make our protocols less brittle.

To maintain a constraint space during interaction between agents we must send, along with each message, a description of that space. For a finite domain solver this description might be in the form of variable ranges for each variable in the expanded protocol clauses. These ranges must then be applied on each expansion step, thus maintaining the ranges as constraints change. The simplest way of doing this, given that a protocol may be distributed across any number of agents, is to restrict our interactions to those which are linear (in the sense given at the end of Section 6) and send along with the protocol the clause store, \mathcal{S} (this time defining the state of the interaction between all participants) and a set, \mathcal{V} , containing the current restriction for each variable in the clauses of \mathcal{S} . Individual agents then react to messages in a similar way to definition 2, except in this case the *react* definition follows a sequence of messages, M_i to M_n , between (possibly different) agents, X_i in role R_i to X_j in role R_j , with each instance of *react* changing both the clause store \mathcal{S} and its variable restrictions, \mathcal{V} .

$$\langle react(M_i, X_i, R_i, \mathcal{P}, \mathcal{S}_i, \mathcal{V}_i, M_{i+1}, \mathcal{S}_{i+1}, \mathcal{V}_{i+1}), \dots \\ react(M_{n-1}, X_j, R_j, \mathcal{P}, \mathcal{S}_{n-1}, \mathcal{V}_{n-1}, M_n, \mathcal{S}_n, \mathcal{V}_n) \rangle \quad (4)$$

In Section 9 we give a detailed example of this sort of mechanism in operation, where the variable restrictions are finite domain restrictions and the constraint solver is a finite domain solver. Before reaching this example we cover the final concept we consider essential for open agent systems: the ability to broker interactions.

8 Brokering = Meta-variables

A broker is a kind of Web service that, upon being asked by a client to suggest a collaboration appropriate for some task, will send that client a description of the dialogue with which the client can initiate that collaboration. Brokering is required in open agent systems, where agents newly entering an environment may not know the forms of social norm expected and may not even know which agents are available. A basic, generic definition of brokering is represented succinctly using expressions 5 and 6 below.

A broker, B , can receive a request for a protocol for a task, T , and will send the protocol \mathcal{P} if it has it.

$$\begin{aligned}
&a(\text{broker}, B) :: \\
&\quad \text{ask}(\text{send_protocol}(T)) \Leftarrow a(\text{client}(B), A) \text{ then} \\
&\quad \text{inform}(\text{protocol}(T, \mathcal{P})) \Rightarrow a(\text{client}(B), A) \Leftarrow \text{protocol_for_task}(A, T, \mathcal{P})
\end{aligned} \tag{5}$$

A client, A , for broker, B , can send a request for a protocol for a task, T ; then receives the protocol segment \mathcal{P} ; then continues its protocol by following \mathcal{P} .

$$\begin{aligned}
&a(\text{client}(B), A) :: \\
&\quad \text{ask}(\text{send_protocol}(T)) \Rightarrow a(\text{broker}, B) \Leftarrow \text{have_task}(T) \text{ then} \\
&\quad \text{inform}(\text{protocol}(T, \mathcal{P})) \Leftarrow a(\text{broker}, B) \text{ then} \\
&\quad \mathcal{P}
\end{aligned} \tag{6}$$

The generality of this definition of brokering comes from the last step in clause 6 which allows a protocol sent via a message to be inserted into the protocol followed by the recipient of that message. This is similar to the construction of executable goals in logic programming languages, but moved to a distributed dialogue setting.

Notice also that this form of brokering appears to be compatible with forms of brokering and matchmaking developed elsewhere (for example [13]). The point at which matchmaking occurs in the protocol above is when the *protocol_for_task*(A, T, \mathcal{P}) constraint is solved in clause 5, producing a protocol, \mathcal{P} , for collaboration to solve task T . Different matchmaking algorithms solve this constraint with differing levels of sophistication:

- “Yellow pages” brokers normally allow only propositional tasks and would return as \mathcal{P} a protocol only of the form:

$$\text{ask}(Q) \Rightarrow a(R, S) \text{ then } \text{inform}(A) \Leftarrow a(R, S)$$

where Q is a query appropriate for the task, T ; A is the identifier of the agent who may answer that query while in role R ; and S is the answer obtained in response.

- Brokers returning linear sequences of agent interactions (e.g. [14]) generalise yellow pages brokering by offering more than one query-response interaction in performing a task, so \mathcal{P} can in this case be of the form:

$$\begin{aligned}
&\text{ask}(Q_1) \Rightarrow a(R_1, S_1) \text{ then } \text{inform}(A_1) \Leftarrow a(R_1, S_1) \text{ then } \dots \\
&\dots \text{ then } \text{ask}(Q_n) \Rightarrow a(R_n, S_n) \text{ then } \text{inform}(A_n) \Leftarrow a(R_n, S_n)
\end{aligned}$$

where n is the number of agents needed to perform the task. More sophisticated brokers of this type can generate conditional messages in \mathcal{P} to deal with constraints such as ontology translation between terms in messages.

- Brokers exist for assembling more complex structures for \mathcal{P} . Those of which we are aware assume that process specifications for each individual service are available (expressed in a language such as DAML-S) and use a planning system to compose these service components into a plan for service invocation. Examples of such systems include the planning component of the RETSINA system [13] and the SHOP2

system applied to DAML-S [15]. Protocols in our LCC language could be viewed as a form of plan that might be constructed using methods analogous to those of the RETSINA and SHOP2 experiments. It is not yet clear, however, whether these plan-based composition methods can be applied directly to composition of LCC protocols.

9 Example Combining Sections 3 to 8

We now combine the elements introduced earlier to demonstrate how they apply to the scenario given at the end of Section 1. Figure 3 describes the protocol for interaction between vendor and supplier.

The agents involved in the protocol of Figure 3 must be capable of satisfying the constraints it imposes. Some of the axioms used in constraint solving might be standard for all agents, in which case they are shared among all agents (and propagated with the protocol). The definition necessary to choose constraints on attributes is an example of this sort of standardisation, since all agents would be assumed to have precisely the same interpretation of *choose/2*, which constructs a conjunctive constraint on attribute values from a set of attribute names. Expression 11 gives a definition, which assumes that a predicate *choice/2* that determines each constraint is satisfiable by the agent asked to choose.

$$\begin{aligned} \text{choose}([\text{att}(\text{Att})|T], C \wedge R) &\leftarrow \text{choice}(\text{Att}, C) \wedge \text{choose}(T, R) \\ \text{choose}([], \text{true}) & \end{aligned} \quad (11)$$

As an example of knowledge private to an agent, we now define for the customer the ranges of acceptable values for attributes of the personal computer under discussion. For instance, the customer would accept disk space of 40 or above. We also define how the specific values for attributes are chosen by the customer from the ranges agreed via earlier dialogue with the vendor: the maximum from the range being taken for every attribute except for price which is minimised.

$$\begin{aligned} \text{need}(\text{pc}) \\ \text{sells}(\text{pc}, s1) \\ \text{acceptable}(\text{disk_space}(D)) &\leftarrow D \text{ in } 40..sup \\ \text{acceptable}(\text{monitor_size}(M)) &\leftarrow M \text{ in } 15..sup \\ \text{acceptable}(\text{price}(-, -, P)) &\leftarrow P \text{ in } 800..2000 \\ \text{choice}(\text{disk_space}(D), \text{true}) \\ \text{choice}(\text{monitor_size}(M), \text{true}) \\ \text{choice}(\text{price}(-, -, P), \text{minimise}(P)) \end{aligned} \quad (12)$$

The vendor agent's local constraints are defined in a similar way to that of the customer. We define the available ranges for the attributes needed to configure a PC and relate these to its price via a simple equation (the aim being to demonstrate the principle of relating constraints rather than to have a realistic pricing policy).

A customer, C , can send a request to vendor, V , to buy an item, X , that the customer needs and believes the vendor sells. Then the customer takes the role of negotiator with the vendor.

$$\begin{aligned}
& a(\text{customer}, C) :: \\
& \quad \text{ask}(\text{buy}(X)) \Rightarrow a(\text{vendor}, V) \leftarrow \text{need}(X) \wedge \text{sells}(X, V) \text{ then} \quad (7) \\
& \quad a(\text{neg_customer}(X, V, []), C)
\end{aligned}$$

A negotiating customer with a set, S , of negotiated attributes of the desired item, X , either receives an offer of a new attribute, A , and accepts that (continuing in the negotiating role with A added to S) or it receives a request to commit to the current set of negotiated attributes and replies with the constraints, C_a , it wishes to impose on those attributes; then receives confirmation from the vendor of the final values, F , for the attributes once the customer's constraints have been applied at the vendor's side.

$$\begin{aligned}
& a(\text{neg_customer}(X, V, S), C) :: \\
& \quad \left(\begin{array}{l} \text{offer}(A) \Leftarrow a(\text{neg_vendor}(X, C, _), V) \text{ then} \\ \text{accept}(A) \Rightarrow a(\text{neg_vendor}(X, C, _), V) \leftarrow \text{acceptable}(A) \text{ then} \\ a(\text{neg_customer}(X, V, [\text{att}(A)|S]), C) \end{array} \right) \\
& \quad \text{or} \\
& \quad \left(\begin{array}{l} \text{ask}(\text{commit}) \Leftarrow a(\text{neg_vendor}(X, C, _), V) \text{ then} \\ \text{tell}(\text{commit}(S, C_a)) \Rightarrow a(\text{neg_vendor}(X, C, _), V) \leftarrow \text{choose}(S, C_a) \text{ then} \\ \text{tell}(\text{sold}(F)) \Leftarrow a(\text{neg_vendor}(X, C, _), V) \end{array} \right) \quad (8)
\end{aligned}$$

A vendor, V , receives a request from a customer, C , to buy an item, X ; then takes the role of negotiator with the customer over the attribute set, S , that applies to that item.

$$\begin{aligned}
& a(\text{vendor}, V) :: \\
& \quad \text{ask}(\text{buy}(X)) \Leftarrow a(\text{customer}, C) \text{ then} \quad (9) \\
& \quad a(\text{neg_vendor}(X, C, S), V) \leftarrow \text{attributes}(X, S)
\end{aligned}$$

A negotiating vendor with a set, S , of negotiable attributes of the desired item, X , either takes the first element, A , of S and offers it to the customer for acceptance (continuing then in its negotiating role with the remaining attributes, T) or if S is empty it asks the customer to commit to the attributes they have discussed and receives the customer's constraints, C_a , on the final values of those attributes then, if these are satisfiable, it informs the customer of the final attribute values, F , for the sold item.

$$\begin{aligned}
& a(\text{neg_vendor}(X, C, S), V) :: \\
& \quad \left(\begin{array}{l} \text{offer}(A) \Rightarrow a(\text{neg_customer}(X, V, _), C) \leftarrow S = [A|T] \wedge \text{available}(A) \text{ then} \\ \text{accept}(A) \Leftarrow a(\text{neg_customer}(X, V, _), C) \text{ then} \\ a(\text{neg_vendor}(X, C, T), V) \end{array} \right) \\
& \quad \text{or} \\
& \quad \left(\begin{array}{l} \text{ask}(\text{commit}) \Rightarrow a(\text{neg_customer}(X, V, _), C) \leftarrow S = [] \text{ then} \\ \text{tell}(\text{commit}(F, C_a)) \Leftarrow a(\text{neg_customer}(X, V, _), C) \text{ then} \\ \text{tell}(\text{sold}(F)) \Rightarrow a(\text{neg_customer}(X, V, _), C) \leftarrow C_a \end{array} \right) \quad (10)
\end{aligned}$$

Fig. 3. Protocol for our example

$$\begin{aligned}
& \text{attributes}(pc, [disk_space(D), monitor_size(M), price(D, M, P)]) \\
& \text{available}(disk_space(D)) \leftarrow D \text{ in } 40..80 \\
& \text{available}(monitor_size(M)) \leftarrow M \text{ in } 15..18 \\
& \text{available}(price(D, M, P)) \leftarrow 1000 + ((M - 15) * 100) + ((D - 40) * 10) \neq P \\
& \text{minimise}(V) \leftarrow fd_min(V, Vm) \wedge V \text{ in } Vm..Vm
\end{aligned}
\tag{13}$$

The sequence of message passing that follows from the protocol of Figure 3 and the constraints of expressions 11, 12 and 13 is shown below. The dialogue iterates between a customer, *b1*, and a vendor, *s1*. Each illocution shows: the type of the agent sending the message; the message itself; the type of agent to which the message is sent; and the variable restrictions applying to the message (the term $r(V, C)$ relating a finite domain constraint C to a variable V). The first illocution is the customer making initial contact with the vendor. Illocutions two to seven then are offers of ranges for attributes (*disk_space*, *monitor_size* and *price*) each of which are accepted by the customer. At illocution eight the vendor, which has worked through all its relevant attributes, asks for commitment from the customer. In reply, the customer asks the vendor to minimise its price (the variable C in illocution nine). Finally, the vendor offers a sale at a price of 1000 (it having decided simply to take the minimum of the current range restriction on C). The variables for *disk_space* and *monitor_size* (B and A respectively) are left without commitment, although the vendor could have committed to these if it so desired.

```

Sender : a(customer, b1)
Message : ask(buy(pc))
Recipient : a(vendor, s1)
Restrictions : []

Sender : a(neg_vendor(pc, b1, [disk_space(A), monitor_size(B), price(A, B, -)]), s1)
Message : offer(disk_space(A))
Recipient : a(neg_customer(pc, s1, -), b1)
Restrictions : [r(A, [[40|80]])]

Sender : a(neg_customer(pc, s1, []), b1)
Message : accept(disk_space(A))
Recipient : a(neg_vendor(pc, b1, -), s1)
Restrictions : [r(A, [[40|80]])]

Sender : a(neg_vendor(pc, b1, [monitor_size(A), price(B, A, -)]), s1)
Message : offer(monitor_size(A))
Recipient : a(neg_customer(pc, s1, -), b1)
Restrictions : [r(B, [[40|80]]), r(A, [[15|18]])]

Sender : a(neg_customer(pc, s1, [att(disk_space(A))]), b1)
Message : accept(monitor_size(B))
Recipient : a(neg_vendor(pc, b1, -), s1)
Restrictions : [r(B, [[15|18]]), r(A, [[40|80]])]

Sender : a(neg_vendor(pc, b1, [price(A, B, C)]), s1)
Message : offer(price(A, B, C))
Recipient : a(neg_customer(pc, s1, -), b1)
Restrictions : [r(C, [[1000|1700]]), r(B, [[15|18]]), r(A, [[40|80]])]

Sender : a(neg_customer(pc, s1, [att(monitor_size(A)), att(disk_space(B))]), b1)
Message : accept(price(B, A, C))
Recipient : a(neg_vendor(pc, b1, -), s1)
Restrictions : [r(C, [[1000|1700]]), r(B, [[40|80]]), r(A, [[15|18]])]

Sender : a(neg_vendor(pc, b1, []), s1)
Message : ask(commit)
Recipient : a(neg_customer(pc, s1, -), b1)
Restrictions : []

Sender : a(neg_customer(pc, s1, [att(price(A, B, C)), att(monitor_size(B)), att(disk_space(A))]), b1)
Message : tell(commit([att(price(A, B, C)), att(monitor_size(B)), att(disk_space(A))], minimise(C)))

```

Recipient : $a(\text{neg_vendor}(pc, b1, _), s1)$
Restrictions : $[r(C, [[1000|1700]]), r(B, [[15|18]]), r(A, [[40|80]])]$

Sender : $a(\text{neg_vendor}(pc, b1, []), s1)$
Message : $\text{tell}(\text{sold}([\text{att}(\text{price}(A, B, 1000)), \text{att}(\text{monitor_size}(B)), \text{att}(\text{disk_space}(A))]))$
Recipient : $a(\text{neg_customer}(pc, s1, _), b1)$
Restrictions : $[r(B, [[15|18]]), r(A, [[40|80]])]$

Recall that the means of each agent maintaining an appropriate role in the interaction is by expanding the clause it selects for its initial role (see Section 5). The term below is the fully expanded clause used by agent $b1$ in the role of a customer. By following through the nesting in this clause, the reader may reconstruct the expansions of the initial customer clause (clause 7 in Figure 3) performed using the transformations of Figure 2 and observe that this allows the message sequence for the customer described above.

$$\begin{aligned}
& a(\text{customer}, b1) :: \\
& c(\text{ask}(\text{buy}(pc)) \Rightarrow a(\text{vendor}, s1)) \text{ then} \\
& (a(\text{neg_customer}(pc, s1, []), b1) :: \\
& \quad c(\text{offer}(\text{disk_space}(A)) \Leftarrow a(\text{neg_vendor}(pc, b1, \left[\begin{array}{l} \text{disk_space}(A), \\ \text{monitor_size}(B), \\ \text{price}(A, B, 1000) \end{array} \right]), s1)) \text{ then} \\
& \quad c(\text{accept}(\text{disk_space}(A)) \Rightarrow a(\text{neg_vendor}(pc, b1, \left[\begin{array}{l} \text{disk_space}(A), \\ \text{monitor_size}(B), \\ \text{price}(A, B, 1000) \end{array} \right]), s1)) \text{ then} \\
& (a(\text{neg_customer}(pc, s1, [\text{att}(\text{disk_space}(A))]), b1) :: \\
& \quad c(\text{offer}(\text{monitor_size}(B)) \Leftarrow a(\text{neg_vendor}(pc, b1, \left[\begin{array}{l} \text{monitor_size}(B), \\ \text{price}(A, B, 1000) \end{array} \right]), s1)) \text{ then} \\
& \quad c(\text{accept}(\text{monitor_size}(B)) \Rightarrow a(\text{neg_vendor}(pc, b1, \left[\begin{array}{l} \text{monitor_size}(B), \\ \text{price}(A, B, 1000) \end{array} \right]), s1)) \text{ then} \\
& (a(\text{neg_customer}(pc, s1, \left[\begin{array}{l} \text{att}(\text{monitor_size}(B)), \\ \text{att}(\text{disk_space}(A)) \end{array} \right]), b1) :: \\
& \quad c(\text{offer}(\text{price}(A, B, 1000)) \Leftarrow a(\text{neg_vendor}(pc, b1, [\text{price}(A, B, 1000)], s1)) \text{ then} \\
& \quad c(\text{accept}(\text{price}(A, B, 1000)) \Rightarrow a(\text{neg_vendor}(pc, b1, [\text{price}(A, B, 1000)], s1)) \text{ then} \\
& (a(\text{neg_customer}(pc, s1, \left[\begin{array}{l} \text{att}(\text{price}(A, B, 1000)), \\ \text{att}(\text{monitor_size}(B)), \\ \text{att}(\text{disk_space}(A)) \end{array} \right]), b1) :: \\
& \quad c(\text{ask}(\text{commit}) \Leftarrow a(\text{neg_vendor}(pc, b1, [], s1)) \text{ then} \\
& \quad c(\text{tell}(\text{commit}(\left[\begin{array}{l} \text{att}(\text{price}(A, B, 1000)), \\ \text{att}(\text{monitor_size}(B)), \\ \text{att}(\text{disk_space}(A)) \end{array} \right], \text{minimise}(1000))) \Rightarrow \\
& \quad \quad a(\text{neg_vendor}(pc, b1, [], s1)) \text{ then} \\
& \quad c(\text{tell}(\text{sold}(\left[\begin{array}{l} \text{att}(\text{price}(A, B, 1000)), \\ \text{att}(\text{monitor_size}(B)), \\ \text{att}(\text{disk_space}(A)) \end{array} \right])) \Leftarrow a(\text{neg_vendor}(pc, b1, [], s1))))))
\end{aligned}$$

Although this example is compact it demonstrates capabilities beyond standard semantic web service specification languages (such as OWL-S), which do not allow recursion over data structures and therefore could not represent a recursive negotiation like the one in this example. It also allows the protocol of Figure 3 to be brokered to any agent asking for it (that brokering interaction also being represented using clauses 5 and 6 of Section 8) - a capability not possessed by other forms of computation for agent social norms. Finally, it demonstrates a simple way of managing finite domain constraints across multi-agent interactions.

10 Conclusions

A criticism of the LCC approach from the mainstream semantic web or agent communities might be that it too closely resembles logic programming. It is true that the language described in this paper uses data structures familiar to logic programmers but those that are highly specific to Prolog (e.g. the list expressions used) are not essential to LCC and could be replaced by others according to taste. The essence of LCC

is its mixture of process calculus and Horn clauses. Both of these aspects do appear in the mainstream (though sometimes heavily disguised), for example in the process component of OWL-S specifications or in the rule-based reasoners being constructed to supplement Description Logic reasoners for semantic web services. The advantage of being closer to logic programming than is fashionable in the mainstream is that we are able to make our specifications executable through simple, well known methods that (because they are well established) we know can be taught to engineers of more traditional systems. Many comparable specification languages in the semantic web services domain do not possess this advantage.

The emphasis of this paper is on the way in which we adapt traditional methods to this new application. The agents research group at Edinburgh is developing LCC in ways we shall describe in other papers:

- Walton ([11]) has produced a translator from a variant of the language to Promela, allowing him to model check protocols using the SPIN model checker.
- McGinnis ([16, 17]) is exploring how to make interactions more adaptable by allowing transformations to the protocol by participating agents, leading to notions of “safe” adaptations.
- Barker is applying LCC to the problem of experiment coordination on e-science grids, requiring him to reconcile the data-flow paradigm assumed by many grid service architectures with the messaging processes of LCC.
- Guo ([18]) is studying how to translate to LCC from traditional business process modelling languages.
- Hassan ([19]) is developing more sophisticated forms of constraint management beyond those described in the current paper.

Ultimately, our aim is to produce a single form of specification that supports specification, analysis and modelling for complex, coordinated, multi-agent systems.

References

1. Robertson, D.: A lightweight method for coordination of agent oriented web services. In: Proceedings of AAAI Spring Symposium on Semantic Web Services, California, USA (2004)
2. Decker, K., Pannu, A., Sycara, K., Williamson, M.: Designing behaviors for information agents. In: Proceedings of the First International Conference on Autonomous Agents. (1997)
3. Giampapa, J., Sycara, K.: Team-oriented agent coordination in the retina multi-agent system. Technical Report CMU-RI-TR-02-34, Robotics Institute, Carnegie Mellon University (2002)
4. McIlraith, S., Son, T.: Adapting golog for composition of semantic web services. In: Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning. (2002) 482–493
5. Sheshagiri, M., desJardins, M., Finin, T.: A planner for composing services described in daml-s. In: International Conference on Automated Planning and Scheduling. (2003)
6. Robertson, D.: A lightweight coordination calculus for agent social norms. In: AAMAS Workshop on Declarative Agent Languages and Technologies, New York, USA (2004)

7. Esteva, M., Padget, J., Sierra, C.: Formalizing a language for institutions and norms. In: *Intelligent Agents VIII, Lecture Notes in Artificial Intelligence*. Volume 2333. Springer-Verlag (2002) 348–366
8. Esteva, M., de la Cruz, D., Sierra, C.: Islander: an electronic institutions editor. In: *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems*. (2002) 1045–1052
9. Clark, K.L., McCabe, F.G.: Go! for multi-threaded deliberative agents. In Leite, J.A., Omicini, A., Sterling, L., Torroni, P., eds.: *Declarative Agent Languages and Technologies, First International Workshop, DALT 2003*. Melbourne, Victoria, July 15th, 2003. Workshop Notes. (2003) 17–32
10. Artikis, A., Pitt, J., Sergot, M.: Animated specifications of computational societies. In Castelfranchi, C., Lewis Johnson, W., eds.: *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems*, Bologna, Italy, Association for Computing Machinery (2002) 1053–1061
11. Walton, C.: Model checking multi-agent web services. In: *Proceedings of AAAI Spring Symposium on Semantic Web Services*, California, USA (2004)
12. Kagal, L., Finin, T., Joshi, A.: A policy language for pervasive systems. In: *Fourth IEEE International Workshop on Policies for Distributed Systems and Networks*. (2003)
13. Paolucci, M., Sycara, K., Kawamura, T.: Delivering semantic web services. Technical Report CMU-RI-TR-02-32, Robotics Institute, Carnegie Mellon University (2003)
14. Robertson, D., Correa da Silva, F., Agusti, J., Vasconcelos, W.: A lightweight capability communication mechanism. In: *Proceedings of the 13th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, New Orleans, Louisiana (2000)
15. Wu, D., Parsia, B., Sirin, E., Hendler, J., Nau, D.: Automating daml-s web services composition using shop2. In: *Proceedings of 2nd International Semantic Web Conference*. Volume 2870 of *Lecture Notes in Computer Science*., Springer-Verlag (2003) 195–210
16. McGinnis, J., Robertson, D., Walton, C.: Using distributed protocols as an implementation of dialogue games. In: *Proceedings of the First European workshop on Multi-Agent Systems*. (2003)
17. McGinnis, J., Robertson, D.: Dynamic and distributed interaction protocols. In: *Proceedings of the Fourth Symposium on Adaptive Agents and Multi-Agent Systems*. (2004)
18. Li, G., Chen-Burger, J., Robertson, D.: Mapping a business process model to a semantic web services model. In: *Proceedings of the IEEE International Conference on Web Services*, San Diego (2004)
19. Hassan, F., Robertson, D.: Constraint relaxation to reduce brittleness of distributed agent protocols. In: *Proceedings of the ECAI Workshop on Coordination in Emergent Agent Societies*, Valencia, Spain (2004)