A Lightweight Coordination Calculus for Agent Systems

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

No Institute Given

Abstract. The concept of a social norm is used in multi-agent systems to specify behaviours required of agents interacting in a given social context. We describe a method for specifying social norms that is more compact than existing methods without loss of generality and permits simple but powerful mechanisms for analysis and deployment. We explain the method and how to compute with it.

1 Introduction: A Broad View of Social Norms

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 opportunistically 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 in a way that allows substantial autonomy for individual agents while maintaining the basic rules of social interaction appropriate to particular coordinated tasks. 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. This is the broad sense in which "social norm" is used in this paper, recognising that it possesses more specific connotations for part of the multi-agent systems community.

Solving coordination problems requires some description of the focus of coordination. One way of doing this is by the use of policy languages (*e.g.* [7]). 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 temporal reasoning to represent the required behaviours of individual services (*e.g.* [1]); shared models for coordinating services (*e.g.* [4]) or the process of composing services (*e.g.* [9, 11]). As recognised in earlier studies on conversation policies [5] the constraints

on interaction between agents often are more "fine grained" than those anticipated in standard performative languages like FIPA-ACL. One solution to this problem is the concept of an electronic institution [3, 2] to which we return in the next section.

In what follows we shall present an approach to coordination that we intend to be consistent with the views described above but which is also comparatively lightweight to use. We begin, in Section 2, by summarising the concept of social norms using the Islander system as an example. In Section 3 we introduce the Lightweight Coordination Calculus (LCC) which is a process calculus for specifying social norms. A basic example of its use is in Section 4. LCC is a comparatively simple but flexible language and can be supplied with a straightforward method for constraining the behaviour of an individual agent in a collaboration, as described in Section 5. It is then possible to construct simple, general-purpose mechanisms for multi-agent coordination that harness this method (see Section 6). LCC is intended as a practical, executable specification language and has been used for a variety of purposes which we summarise in Section 7. Finally, in Section 8 we return to mainstream performative languages and show how LCC may be used to describe the social norm aspects of those types of system.

2 Islander: A Means of Enforcing Social Norms

The Islander system [2] is sketched here as an example of a traditional means of enforcing social norms. In this section we introduce the approach and main representational features of Islander. In Section 3 we shall return to these when introducing the LCC notation. The framework for describing agent interactions in Islander relies upon a (finite) set of state identifiers representing the possible stages in the interaction. Agents operating within this framework must be allocated roles and may enter or leave states depending on the illocutions (via message passing) that they have performed. In order to structure the description, states are grouped into scenes. An institution is then defined by a set of scenes and a set of connections between scenes with constraints determining whether agents may move across these connections. A scene is defined as a collection of the following sets: roles; state identifiers; an initial state identifier; final state identifiers; access state identifiers for each role; exit state identifiers for each role; and cardinality constraints on agents per role. A social norm for an agent is defined by an antecedent (defined as a list of scene-illocution pairs) and a consequent (the predicates obliged to be true if the antecedent illocutions have taken place). This thumbnail sketch of the main components of an institution model suffices to give the reader an overview of the approach. Later we shall revisit these components in more detail.

This sort of state transition model has been shown to be adequate for constraining multi-agent dialogue in situations, such as auctions, where social norms are essential for reliable behaviour. It also permits a style for enforcement of the model during deployment, in which the state-based model of interaction is used to check that the agents involved do indeed conform to the model. It suffers, however, from two weaknesses. The first weakness is its reliance on representing the entire model of interaction as a single (albeit structured) state transition model. This makes enforcement of the model difficult except via some form of representation of the global state of the interaction as it applies to the group of agents involved in it.

Thus far, the only solutions to this problem have been to maintain a single institution model with which all agents must synchronise or to have synchronised distribution of a single model. Both these solutions undermine the distributed nature of the computation by enforcing centralised control over interactions between agents. The second weakness (related to the first) is that its focus on global state of multi-agent interaction makes it difficult to disentangle the specification of constraints on individual agent processes contributing to that state. This is of practical importance because all current efforts on large scale agent deployment via standardised Web services (*e.g.* DAML-S) use process models specific to individual agents. The relevance of LCC to the modelling and deployment of semantic web services has previously been argued in [10, 12]. In the current paper we concentrate on the related but separable issue of its relevance to multi-agent social norms. The system described in the remainder of this paper is a process calculus that can be used to describe social norms as complex as those of state-based systems such as Islander, with the advantage that these can be deployed without requiring centralised control.

3 LCC Syntax

LCC borrows the notion of role from institution based systems, as described in the previous section but reinterprets this as a form of typing on a process in a process calculus. Process calculi have been used before to specify social norms (see for example [3]) 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, although the LCC constraints used in current implementations are in first order predicate calculus. 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 given in Figure 1.

Although LCC looks different to state-based systems like Islander it provides all the representational features we saw in Section 2. These are:

Role and scene identification : These are described by the agent type definition ($Type$ in Figure 1) which permits any structured term to be used to describe the agent type, hence this structure could include the agent's scene and role.

Initial state : Although LCC does not require a single initial state we can choose to have one of the clauses (an instance of $Clause$ in Figure 1) determine the scene and role of the agent that initiates the interaction.

Final and exit states : Although states are not labelled in LCC each agent can determine its current position in the interaction protocol by using the definition of protocol closure described in Figure 3.

Movement between states : Each agent moves between states by following its clause in the protocol. LCC allows changes of scene/role and recursion over scenes/roles

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

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. The operators $\leftarrow$, $\wedge$ and $\vee$ are the normal logical connectives for implication, conjunction and disjunction. $M \Rightarrow A$ denotes that a message, M, is sent out to agent A. $M \Leftarrow A$ denotes that a message, M, from agent A is received. The implication operator dominates the message operators, so for example $M \Rightarrow Agent \leftarrow C$ is scoped as $(M \Rightarrow Agent) \leftarrow C$

Figure 1. Syntax of LCC interaction framework

(recall that states and roles are described in LCC using structured terms so these can be used to describe recursive orderings).

Access to protocol for agents : Agents can access a protocol by selecting an appropriate clause. The means of distributing protocols described in Section 6 allow agents hitherto unaware of a protocol to be "invited into" an interaction, so LCC-enabled agents may either initiate interaction or reactively join interactions.

Constraints on individual agents : Constraints can be applied to sending messages, accepting messages and to change of scene/role (see use of C in Figure 3). In order to keep the LCC language simple there is no special notation in LCC for representing temporal constraints (such as timeouts or temporal prohibitions) so one must construct these from normal first-order expressions.

Constraints on groups of agents : Although LCC clauses are used by individual agents it is easy to "thread" information through a group of interacting agents via arguments in the structured terms defining each agent's type ($Type$ in Figure 1). Constraints relevant to the group (such as cardinality constraints on the set of agents participating in an interaction) can then be checked by constraints on the individual agents.

In Section 7 we describe aspects of LCC that go beyond current abilities of systems such as Islander. First we give an illustrative example of LCC in use.

4 Example LCC Interaction Framework

Figure 2 shows an example of a protocol in LCC for a basic multi-agent auction. There are two initial roles - a bidder and an auctioneer - with the auctioneer's role changing during the interaction between that of a caller of bids and a vendor collecting offers from

bidders (notice the use of mutual recursion between auctioneer and vendor in clauses 2 and 4). The list of bidders known to the auctioneer (the variable named S in clauses 1 to 4) is assumed to be fixed throughout the auction but it is straightforward to extend the protocol to allow new bidders to join - for example we could add a clause for an introductory bidder that would ask for entry to the auction and then become a bidder; then extend clause 4 to allow acceptance of an invitation to bid.

The point of Figure 2 is not to describe an optimal auction protocol but to give the reader a flavour of what it is like to describe protocols in LCC. For those familiar with logic programming the style of description should be reassuringly familiar, since each clause of the protocol can be read similarly to a Horn clause with the "head" of the clause being the agent role and the "body" being the definition of its behaviour when discharging that role. Our preliminary efforts at teaching this language to first year postgraduate students encourages us to believe that teaching LCC as a form of declarative programming language is comparable in difficulty to teaching other declarative languages, such as Prolog. LCC is, however, a language for coordinating distributed processes so forms of debugging and analysis appropriate to asynchronous systems also are required to support LCC engineers. For example, model checking has been performed for a variant of LCC [12], analogous to model checking applied to systems like Islander [6].

Although analytical techniques like model checking help support engineers, a simple and predictable computational model of the behaviour of protocols in deployment is fundamental to good engineering. In the next two sections we describe this model for LCC, beginning in Section 5 with the most basic computational step of accessing and updating the protocol; then in Section 6 showing how this is harnessed to provide flexible styles of multi-agent coordination.

5 Clause Expansion

To enable an agent to conform to a LCC 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 recording 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. In this section we describe an expansion algorithm, showing in Section 6 how to use it with a selection of coordination systems.

The mechanism described below for coordinating agents using LCC assumes some means by which messages may be sent to a message exchange system and some means by which messages may be read from that system. The means of transmitting messages is not prescribed by LCC so this could be done using any appropriate distributed communication infrastructure. LCC does, however, make the following assumptions related to the format of messages:

- A message must contain (at least) the following information, which can be encoded and decoded by the sending and receiving mechanisms attached to each agent:
  - An identifier, I, for the social interaction to which the message belongs. This identifier must be unique and is chosen by the agent initiating the social interaction.

The role of an auctioneer, A, is performed by performing the role of an auctioneer for an item, X, with a set of bidders, S, at initial reserve price, R, and an initial empty list, $[]$, of bids. The constraint $item(X, R)$ determines the initial reserve price for the item and the constraint $bidders(S)$ determines the set of bidding agents.

$$a(auctioneer, A) \quad :: \quad a(auctioneer(X, S, R, []), A) \leftarrow item(X, R) \wedge bidders(S) \quad (1)$$

An auctioneer is first a caller for bids and then becomes a vendor.

$$a(auctioneer(X, S, R, Bids), A) \quad :: \quad \begin{array}{l} a(caller(X, S, R), A)\ then \\ a(vendor(X, S, R, Bids), A) \end{array} \quad (2)$$

A caller recurses through the list, S, of bidders, sending each an invitation to bid.

$$a(caller(X, S, R), A) \ :: \ \begin{array}{l} (invite\_bid(X, R) \Rightarrow a(bidder, B) \leftarrow S = [B|Sr]\ then\ a(caller(X, Sr, R), A))\ or \\ null \leftarrow S = [] \end{array} \quad (3)$$

A vendor receives a bid which is added to its current collection of bids, C, to give the updated set, C_n. It then does one of the following: sells to the highest bidder if there is one at the current reserve price; continues as a vendor if not all of the bids are collected; reverts to being an auctioneer if all the bids are in but there is no highest bidder or the highest bid exceeds the current reserve.

$$a(vendor(X, S, R, C), A) \quad :: \quad add\_bid(B_b, V_b, C, C_n) \leftarrow bid(X, V_b) \Leftarrow a(bidder, B_b)\ then \left(\begin{array}{l} (sold(X, V_s) \Rightarrow a(bidder, B_s) \leftarrow all\_bid(S, C_n) \wedge highest\_bid(C_n, B_s, V_s) \wedge V_s = R)\ or \\ (a(vendor(X, S, R, C_n), A) \leftarrow not(all\_bid(S, C_n)))\ or \\ (a(auctioneer(X, S, R, []), A) \leftarrow all\_bid(S, C_n) \wedge not(highest\_bid(C_n, \_)))\ or \\ (a(auctioneer(X, S, Rn, []), A) \leftarrow all\_bid(S, C_n) \wedge highest\_bid(C_n, Rn) \wedge Rn > R) \end{array} \right) \quad (4)$$

A bidder receives an invitation to bid from an auctioneer agent; then sends a bid to that agent (in its role as vendor); then either receives a message informing it that the item has been sold to it or it reverts to being a bidder again.

$$a(bidder, B) \quad :: \quad \begin{array}{l} invite\_bid(X, R) \Leftarrow a(auctioneer(X, \_, \_, \_), A)\ then \\ bid(X, V_b) \Rightarrow a(vendor(X, \_, \_, \_), A) \leftarrow bid\_at(X, R, V_b)\ then \\ (sold(X, V_s) \Leftarrow a(vendor(X, \_, \_, \_), A)\ or\ a(bidder, B)) \end{array} \quad (5)$$

Figure 2. LCC framework for an auction example

- A unique identifier, A, for the agent intended to receive the message.
- The role, R, assumed of the agent with identifier A with respect to the message.
- The message content, M, in the syntax defined in Section 3.
- The protocol, $\mathcal{P}$, for continuing the social interaction. This consists of: a set, $\mathcal{C}$, of LCC clauses defining the dialogue framework (see Section 3); and a set, $\mathcal{K}$, of axioms defining any common knowledge assumed during the social interaction. This provides a way of preserving information context as the protocol moves between agents.

– The agent must have a mechanism for satisfying any constraints associated with its clause in the dialogue framework. Where these can be satisfied from common knowledge (the set K above) it is possible to supply standard constraint solvers with the protocol. Otherwise, this is the responsibility of the agent.

Given these assumptions about message format, the basic operation an agent must perform when interacting via LSS is to decide what its next steps for its role in the interaction should be, using the protocol information carried with the message it obtains from some other agent. Recall that the behaviour of an agent in a given role is determined by the appropriate LCC clause. Figure 3 gives a set of rewrite rules that are applied to give an expansion 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:

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

We refer to the rewritten clause, C_n, as an expansion of the original clause, C_i. In the next section this basic expansion method is used for multi-agent coordination.

6 Coordination Mechanisms

Figure 4 depicts two methods of distributed coordination using LCC. Both use the clause expansion mechanism given in Section 5, the only difference between them being in the way the state of the interaction is preserved during interactions. For simplicity, the diagrams of Figure 4 depict an interchange between only two agents (Agent 1 and Agent 2), with a message (Message 1) being sent from Agent 1 to Agent 2 and another message (Message 2) being returned in response. We describe below the first coordination mechanism in detail, then explain the second as a special case of the first.

Method 1 of Figure 4 depicts an instance of the coordination method described in detail as follows (from the point of view of Agent 2 in the diagram):

– An agent with unique identifier, A, retrieves a message of the form $(I, M, R, A, \mathcal{P})$ where: I is a unique identifier for the coordination; M is the message; R the role assumed of the agent when receiving the message; A the agent's unique identifier; and $\mathcal{P}$ the attached protocol consisting of a set of clauses, $\mathcal{C}$, and a set of axioms, $\mathcal{K}$, describing common knowledge. The message is added to the set of messages currently under consideration by the agent - giving the message set M_i.

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).

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

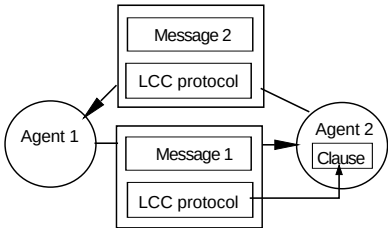
A protocol term is decided to be closed, meaning that it has been covered by the preceding interaction, as follows:

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

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

Figure 3. Rewrite rules for expansion of a protocol clause

Method 1: LCC clauses distributed with protocol (carried with message); used and retained on appropriate agent.



Method 2: LCC clauses distributed with protocol (carried with message); used by appropriate agent but stored with protocol.

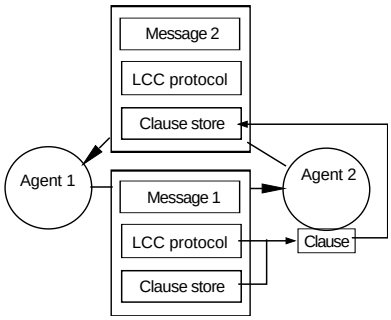


Figure 4. Two methods of coordination

– The agent checks its internal store of dialogue clauses to see if it already has a clause, C_i, indexed under coordination identifier I. If so, it selects it. If not it makes a copy of C_i as an element of $\mathcal{C}$, thus determining its part of the dialogue.
– The rewrite rules of Figure 3 are applied to give an expansion, C_n, of C_i in terms of protocol $\mathcal{P}$ in response to the set of received messages, M_i, producing: a new dialogue clause C_n; an output message set O_n and remaining unprocessed messages M_n (a subset of M_i).
– 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 $\mathcal{P}_n$.

In Method 1 the clauses determining the behaviours of the interacting agents are distributed among the agents as the protocol is passed between them - these are the clauses named C_i in the algorithm above. The state of the interaction is described by the set of these distributed clauses. Notice that each agent must retain only the clause (or clauses if it has multiple roles) appropriate to it. Agents do not need to retain the whole protocol because this is passed with the message, so will return to the agent if other messages arrive as part of the appropriate interaction.

Method 1 is comparatively lightweight because it requires only that an agent can perform clause expansion, as described in Section 5, and that it can store its own copies of LCC clauses. It is possible, however, to place even less burden on individual agents if we have interactions that are linear, in the sense that (regardless of how many agents interact) at any given time exactly one agent alters the state of the interaction. An example of a linear interaction is a dialogue between two agents where each agent takes alternate turn in the interaction. An example of a non-linear interaction is an auction involving a broadcast call for bids (like the one in Figure 2). When the interaction is linear then we can store agents' clauses (named C_i in the algorithm above) with the message rather than with the agent. This is the "Clause store" depicted in the lower diagram of Figure 4. Agents then look up their clauses from this clause store, and the state of the whole interaction is preserved by the message as it passes between agents.

7 Computing with LCC

LCC can be used to tackle a variety of different forms of coordination problem, from those in which agents' behaviours are tightly constrained by the protocol to those in which agents are constrained only in terms of the message sequences they may send. The difference between these two extremes is made by the number and rigidity of the constraints included with the protocol. A tightly constrained protocol has many constraints, all of which have a precise interpretation determined by the protocol designer (Figure 2 is an example), in which case the interaction is similar to a traditional distributed computation with the participating agents acting as processors for the computation described by the LCC protocol. A loosely constrained protocol has few constraints, any of which may have an interpretation given to it by the agent designers, in which case the agents involved may have a greater degree of autonomy within the message passing framework set by the protocol.

Since constraints attach to messages or roles, it has proved most natural in practice for those writing LCC protocols to begin by specifying the (unconstrained) sequences of messages and changes of role for each of the roles in an interaction. Then, once this skeletal structure is in place, constraints can be added to tighten the protocol in whatever way suits the application. The form of refinement is similar to the style of design used in conventional relational and functional programming where a skeletal control structure often is described as a precursor to detailed design. This is why it is advantageous for LCC to resemble these kinds of traditional language, despite being also a process calculus.

Although recent, LCC has been used for a variety of practical purposes:

- In simulation, where we have built simulators for empirical comparison of LCC protocols under controlled conditions. For example, we have compared the performance of different protocols for resource mediation under varying supply and demand regimes. The simulators needed for this sort of empirical analysis have been simple to construct for LCC because we re-use the expansion algorithm of Section 5 within the simulation harnesses.
- In model checking, where we have written a translator from a variant of LCC (Walton's MAP language) to the Promela language which can then be fed into the SPIN model checker.
- In constraint solving, where we have extended the basic clause expansion mechanism to preserve the ranges of finite-domain constraints on variables. This allows agents to restrict rather than simply instantiate constraints when interacting, thus allowing a less rigid interaction.
- To permit human interaction, where we have built a generic user interface (in Tcl-TK interacting with SICStus Prolog) for accepting, viewing and replying to LCC messages. This is intended for prototyping to get a feel for the sort of interaction occurring between agents.

8 LCC and Performative Languages

Although LCC was not intended for direct comparison to performative languages such as FIPA-ACL or KQML, there is a relationship that may be of practical value. Performative languages provide a language for communication between agents that is oriented to the demands of dialogue. They provide ways of describing basic "speech acts" such as asking for information or telling an agent some new information, via performative expressions. This is of benefit because an agents receiving a message with content "wrapped" within performative expressions can have some idea of the role of that message in dialogue. Such languages are, however, limited in the extent to which they can describe dialogue:

- When an agent receives a message this is wrapped only in a single performative, so it can know for example that the message is a "tell" but it is not given any further reference to the broader dialogue of which this message may be a part.

– The semantics of performatives is defined (more or less formally depending on the performative language) in documents describing the language but it is entirely up to the engineers of individual agents to ensure that they adhere to an appropriate semantics. Thus, the sender of a performative has no way of helping the recipient to understand what is meant by it, nor of checking that it was used appropriately.

The remainder of this section shows how LCC overcomes these limitations, offering comparable precision in description of semantics plus the practical benefit of linking these more closely to the mechanics of actual agent dialogue.

There are various ways of describing the semantics of performatives but a common form of description is by defining preconditions and postconditions on the performative message. Preconditions "indicate the necessary states for an agent to send a performative and for the receiver to accept it and successfully process it". Postconditions "describe the states of the sender after the successful utterance of a performative, and of the receiver after the receipt".

An example of this sort of definition is the $tell(A, B, X)$ performative in KQML which describes the act of agent A telling agent B some information, X. Below are the constraints given for this in [8] (ignoring the issue of how the agent knows what it should be telling another agent about). We use the predicates: $k(A, X)$ to denote that A knows X; $b(A, X)$ to denote that A believes X; $i(A, X)$ to denote that A intends to know X and $w(A, X)$ to denote that A wants to know X. These correspond to the predicates $know$, bel, $intend$ and $want$ in [8].

– Preconditions:
  - Agent A believes X and knows that agent B wants to know about X:

$$\begin{array}{l} b(A, X) \wedge \\ k(A, w(B, k(B, X))) \end{array} \tag{6}$$

  - Agent B intends to know that B knows X:

$$i(B, k(B, X)) \tag{7}$$

– Postconditions:
  - Agent A knows that agent B knows that A believes X:

$$k(A, k(B, b(A, X))) \tag{8}$$

  - Agent B knows that agent A believes X:

$$k(B, b(A, X)) \tag{9}$$

These are the basic constraints on $tell$ according to [8]. A more sophisticated set of constraints (described informally in [8]) would accommodate refusal of a $tell$ message by the recipient agent (for example by replying with a $sorry$ or $error$ performative). This allows for more sophisticated dialogue constraints than in expressions 8 and 9 above but is a similar specification task so, to save space, we limit ourselves to the basic interaction.

A difficulty in practice when constraining the use of performatives such as '$tell$', above, is in ensuring that the constraints set in the specification of these performatives

actually hold during the course of a dialogue. How, for example, can both agents (A and B) ensure that B wants to know about X (as preconditions 6 and 7 require)? How can agent A be sure, after it sent the message $tell(A, B, X)$, that postcondition 9 holds, since (for instance) its message may by accident never have been delivered to B. Using LCC we can tackle this problem as follows.

First, it is necessary to define the dialogue associated with the '$tell$' performative. In order to provide acknowledgement of receipt of this message we require a confirmatory response from the recipient (B). For this we add a '$heard$' performative. The message passing framework for the '$tell$' protocol is then as shown in expressions 10 and 11, with the first clause requiring the agent doing the telling (in role T_a) to tell the recipient (in role T_b) and await confirmation that the recipient has heard. KQML pre- and post-conditions 6 and 8 are added to apply the appropriate constraints on Agent A's beliefs. The second clause obliges the recipient to receive the information and confirm that it has heard, again with appropriate constraints 7 and 9.

$$\begin{aligned} a(T_a, A) :: tell(X) \;\Rightarrow\; a(T_b, B) \leftarrow & \\ \begin{pmatrix} b(A, X) \wedge \\ k(A, w(B, k(B, X))) \end{pmatrix} & \; then \\ k(A, k(B, b(A, X))) \;\leftarrow\; heard(X) \;\Leftarrow\; a(T_b, B) & \end{aligned} \tag{10}$$

$$\begin{aligned} a(T_b, B) :: i(B, k(B, X)) \;\leftarrow\; tell(X) \;\Leftarrow\; a(T_a, A)\; then & \\ heard(X) \;\Rightarrow\; a(T_a, A) \;\leftarrow\; k(B, b(A, X)) & \end{aligned} \tag{11}$$

In the example above we included the constraints imposed on the semantics of a performative in the definition of the constraints embedded in our dialogue protocol. This makes them explicit so, if the application demands high reliability, they could be part of a system of automatic checking or endorsement. This is not intrinsic to traditional performative languages.

9 Conclusions

LCC is a language for describing social norms as interacting, distributed processes. Although it is comparatively simple in design (comparable to traditional logic programming languages) it is able to represent concepts generally considered to be essential for representing and reasoning about social norms. A primary aim of LCC (as with other social norm systems) is to interfere as little as possible with the design and operation of individual agents. We have coded (separately for Prolog and Java) compact algorithms for unpacking LCC protocols to yield the illocutions implied by them in whatever is the current state of interaction (see Section 5). Little more than this is required beyond a method for parsing incoming and outgoing LCC-enabled messages (on whatever is the chosen message passing infrastructure) and for satisfying the constraints (if any) associated with appropriate clauses in the protocol.

LCC protocols are modular in the sense that they can be understood separately from the agents participating in the interactions they describe and are neutral to the implementation of those agents. The clauses within an LCC protocol also are modular, so individual roles within an interaction are easy to identify. This makes it comparatively

straightforward to design different models of coordination for LCC depending on the demands of the problem. Section 6 describes three such models.

Since LCC is an executable specification language, work continues on both aspects of the system. On the specification side we have translations from LCC to other more traditional styles of temporal specification, currently a modal logic and a form of situation calculus. On the deployment side we are investigating ways of making the LCC protocols adaptable in ways which preserve the intent of the social norms they describe. We are also investigating how LCC may be adapted to support workflow in computational grids.

References

1. K. Decker, A.S. Pannu, K. Sycara, and M. Williamson. Designing behaviors for information agents. In *Proceedings of the First International Conference on Autonomous Agents*, 1997.
2. M. Esteva, D. de la Cruz, and C. Sierra. Islander: an electronic institutions editor. In *Proceedings of the 1st International Joint Conference on Autonomous Agents and MultiAgent Systems*, pages 1045–1052, 2002.
3. M. Esteva, J. Padget, and C. Sierra. Formalizing a language for institutions and norms. In *Intelligent Agents VIII, Lecture Notes in Artificial Intelligence*, volume 2333, pages 348–366. Springer-Verlag, 2002.
4. J.A. Giampapa and K. Sycara. Team-oriented agent coordination in the retsina multi-agent system. Technical Report CMU-RI-TR-02-34, Robotics Institute, Carnegie Mellon University, December 2002.
5. M. Greaves, M. Holmback, and J. Bradshaw. What is a conversation policy? In F. Dignum and F. Greaves, editors, *Issues in Agent Communication*, pages 118–131. Springer-Verlag, 1999.
6. M. Huget, M. Esteva, S. Phelps, C. Sierra, and M. Wooldridge. Model checking electronic institutions. In *Proceedings of ECAI Workshop on Model Checking and Artificial Intelligence*, Lyon, France, 2002.
7. L. Kagal, T. Finin, and A. Joshi. A policy language for pervasive systems. In *Fourth IEEE International Workshop on Policies for Distributed Systems and Networks*, 2003.
8. Y. Labrou and T. Finin. A semantics approach for KQML: a general purpose communication language for software agents. In *Third International Conference on Knowledge and Information Management*, 1994.
9. S. McIlraith and T. Son. Adapting golog for composition of semantic web services. In *Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning*, pages 482–493, 2002.
10. D. Robertson. A lightweight method for coordination of agent oriented web services. In *Proceedings of AAAI Spring Symposium on Semantic Web Services*, California, USA, 2004.
11. M. Sheshagiri, M. desJardins, and T. Finin. A planner for composing services described in daml-s. In *International Conference on Automated Planning and Scheduling*, 2003.
12. C. Walton. Model checking multi-agent web services. In *Proceedings of AAAI Spring Symposium on Semantic Web Services*, California, USA, 2004.