Component-Based Standardisation of Agent Communication

Frank Guerin and Wamberto Vasconcelos

Dept. of Computing Science, Univ. of Aberdeen, Aberdeen AB24 3UE, UK
{fguerin,wvasconc}@csd.abdn.ac.uk

Abstract. We address the problem of standardising the semantics of agent communication. The diversity of existing approaches suggests that no single agent communication language can satisfactorily cater for all scenarios. However, standardising the way in which different languages are specified is a viable alternative. We describe a standard meta-language in which the rules of an arbitrary institution can be specified. In this way different agent communication languages can be given a common grounding. From this starting point, we describe a component based approach to standardisation, whereby a standard can develop by adding component sets of rules; for example to handle various classes of dialogs and normative relations. This approach is illustrated by example. Eventually we envisage different agent institutions publishing a specification of their rules by simply specifying the subset of standard components in use in that institution. Agents implementing the meta-language can then interoperate between institutions by downloading appropriate components.

1 Introduction

We are interested in facilitating interoperability for agents interacting with different institutions on the Internet. For example, consider a personal agent of a professor who is invited to participate in a conference (say to give a keynote address and chair a session). The personal agent may connect with the conference site and enter a collaborative dialogue with the agents of the various other speakers, and the conference organiser, in order to arrange the schedule of events. Subsequently the agent will connect to various online travel sites to procure airline tickets and accommodation, most likely by means of some auction mechanism. Finally the agent may discover that an airline ticket it has bought does not conform to what was advertised, thus it may seek compensation, lodging an appeal with some arbitration site, and bringing evidence to support the claim. Each of these interactions occurs in a different institution; the requirements for the agent communication language (ACL) in each institution are quite different. Yet, would it be possible to provide a standard language which encompasses all requirements?

Past attempts to standardise agent communication [8,15,7][1] have managed to standardise certain syntactic or pragmatic aspects, but fared poorly when it

[1] Note that the FIPA'97 specification is cited here because the communication language semantics has not changed since then.

M. Baldoni et al. (Eds.): DALT 2007, LNAI 4897, pp. 227–244, 2008.

comes to the issue of the semantics of the communication language. In practice, implementers who claim to be using a particular standard ACL tend to ignore those aspects of the standard that pose difficulties for their implementation (often the formal semantics are ignored); additionally they often create ad hoc extensions when none of the constructs of the standard quite fits their needs. Effectively they invent their own custom dialect, which will not be understood by any other system [22]. Given the diverse needs of different domains, it is probably not feasible to come up with a single standard ACL which will cater for the needs all possible agent systems. Furthermore, a standard ACL would be rigid, precluding the possibility of agents negotiating the semantics of their own custom ACL on the fly, for a particular purpose. The ACL would seem to be the wrong level to standardise at; instead, it would seem appropriate to have a standard way of specifying semantics, to allow developers (or agents themselves) to create their own languages in a standard and structured way. Our proposal is to create a standard meta-language which would allow different interaction languages to be defined for different domains.

The core language, on which developers will build, must be sufficiently expressive to allow any reasonable language to be specified. For this purpose we identify a class of agent communication languages which are universal in the sense that they could be used to simulate any other agent communication language which is computable. We specify one such language and demonstrate its generality by showing how it allows the specification of institutions in which agents can change the rules of the institution itself. With this core in place, we envisage a standard evolving gradually by adding "components". Note that we are using a non-standard meaning of "component", i.e. we are not talking in the software engineering sense where it encapsulates functions and communicates with other components. However it does share some properties of a software component in that it should be reusable and composable with other components. By "component" we mean a set of rules to govern a certain aspect of an interaction. For example, a component may provide rules for normative relations, defining abstractions such as permissions and obligations, and how these change as messages are sent. Further components could then use these abstractions to specify high-level protocols. High level protocols themselves can be specified as components, and so composed with other protocols for flexible interactions. In this way we can give developers the flexibility to define their own components, and publish the specifications, so that others can develop further components, and agents, to work with that language. It is hoped that this could bring together the efforts of the community as similar efforts have done in software engineering by specifying standards for programming languages. A further advantage of the component based approach is that all agents in a society do not necessarily need to support the same components. Some agents may be less sophisticated than others and may support simple reactive protocol components, while other more sophisticated agents may be able to use components which allow them to express their intentions and desires, with a well defined meaning. When agents wish to communicate they would firstly discover which components they support

in common, and then they can determine the level at which they can interact. This ability to implement lightweight versions of an agent communication language is one of the desiderata for agent communication languages outlined by Mayfield *et al.* [19].

In this paper we will illustrate the proposed approach with some examples. We must stress that we are not advocating that the components described in this paper be adopted as a standard; we merely provide simple examples to demonstrate the feasibility of the component-based approach. This paper will focus exclusively on the semantic issues as these have proved to be the most problematic for the standardisation of agent communication. We therefore ignore all pragmatic issues, e.g. how to find the name of an agent who provides some service, authentication, registration, capability definition and facilitation [16]. We assume an agent platform which can take care of all these issues. Pragmatic issues are of course important, but they would require a full treatment in their own right. Furthermore, the types of ACL which we will consider will be restricted to those that have a social semantics; the primary reason for this is the impossibility of verifying languages with mental semantics in an open system, where agents' internals cannot be easily inspected [22].

This paper is organised as follows. Section 2 looks at the most general framework within which all practical ACLs could be specified. Section 3 defines an agent communication language which allows unlimited extensions, and so forms the base component which we later build on. Section 4 adds a component for normative relations. Section 5 discusses how protocols can be added in general, and adds an auction protocol. Section 6 describes a temporal logic component we have added. Section 7 discusses related work and Section 8 concludes with a look to the future.

2 Definition of an Agent Communication Language

In this section we want to define what an ACL is in the most general terms, and to have a formal framework which captures the space of possible ACLs. Following Singh's seminal work on social commitments [22,23], there does seem to be a consensus in the community that the semantics of communication for open systems should be based on social phenomena rather than agents' private mental states [25,9,4,5]. We follow this social approach and we consider that all "reasonable" languages for use in an open system must be of the social type. We do not restrict ourselves to commitments: we allow arbitrary social facts[2].

We define an ACL by specifying an institution. The existence of a communication language presumes the existence of an institution, which defines the

[2] This means that we are not precluded from representing mental states that have been publicly expressed by an agent [12]. This can be handled by treating the agent's publicly expressed mental attitudes as part of the state of affairs in the institution. The difference between this and earlier mentalistic semantics [8,15,7] is that we do not require that agents actually hold the mental states which they have publicly expressed.

meaning of language. Institutions are man-made social structures created to regulate the activities of a group of people engaged in some enterprise. They may be created deliberately, as is the case for formal organisations, or they may be created by conventions evolving over time, as is the case for human culture. Institutions regulate the activities of their members by inventing *institutional facts*, a term due to Searle [21]. Some institutional facts take the form of *rules* while others merely describe a *state of affairs* in the institution. *Rules* describe how institutional facts can be created or modified. An example of an institutional fact of the *state of affairs* type is having the title "doctor"; examples of institutional facts which are rules are the rules in a University which describe how the title can be awarded and by whom. The *rule* type of facts can be used to provide a relationship between the real physical world and the institution; rules can have preconditions which depend on the physical world and/or on other institutional facts. For example, the submission of a thesis physically bound in a specified format is a necessary precondition to the awarding of the title "doctor"; the passing of an examination (a purely institutional fact) is another precondition.

Rules relating to the physical world describe how events or states of the world (typically the actions of members) bring about changes in the institutional facts. The classic example of this is where an utterance by a member of an institution can bring about an institutional fact, for example the naming of a ship: "I hereby name this ship the Queen Elizabeth." [3]. It is not possible for institutional facts to bring about changes in the physical world because the institution itself, being a collection of intangible institutional facts, cannot directly effect any physical change in the world.[3] Institutions may describe their rules in a form which specifies physical effects in the world, but such rules are not strictly true because the physical effects are not guaranteed to happen; the only way in which the institution can influence the actions of its members is through the threat of further institutional facts being created. For example a legal institution may prescribe a term of imprisonment as the consequence of an institutional fact, but it cannot directly bring about the imprisonment of a member; instead it can state that a policeman should use physical force to bring the member to prison, and the policeman can be threatened with the loss of his job if he does not. The rule prescribing imprisonment can be reformulated as a rule which states that if the policeman does not imprison the member by physical force or otherwise, then the policeman loses his job. Thus all the rules relating to the physical world take the form of descriptions of how events or states of the world bring about changes in the institutional facts.

[3] Institutions may indirectly affect the physical world if agents of the institution take physical actions in response to institutional facts. We consider a bank balance to be an institutional fact; it happens that banks have implemented physical agents which act on this institutional fact and dispense money. Nevertheless, the bank balance itself is not a physical fact. Likewise, if certain institutional facts are valued or feared by agents, then they will act in response to them (hence the institutional facts affect the physical world only through the agents).

A further point to note is that the institutional facts being modified by a rule could be rules themselves. Many institutions do modify their rules over time; a legal institution may allow arguments about the rules by which argumentation should take place. This is accommodated by the framework described above, because a rule can modify an institutional fact, and that institutional fact could be another rule.

If we assume that any relevant change in the world's state can be translated into an event, then we can say (without loss of generality) that the institutional facts change only in response to events in the world (we do not allow rules to refer to states of the world). In a typical agent system we rely on the agent platform to handle the generation of events. Typical events include messages being transmitted, timer events and possibly other non communicative actions of agents or events such as agent death. Let E be the set of possible events and let $\mathcal{F}$ be the set of possible institutional facts. Let *update* be a function which updates the institutional facts in response to an event; $update : E \times 2^{\mathcal{F}} \rightarrow 2^{\mathcal{F}}$. Now in an institution $\mathcal{I}$, it is the institutional rules R which indirectly define this update function. The institution interprets the rules in order to define the update function, let the interpreter function be I, where I maps R to some update function. An institution $\mathcal{I}$ can then be fully specified by specifying the interpreter I and the facts $F \subset \mathcal{F}$. F is itself composed of the *rule* type of facts R and the *state of affairs* type of facts A, so $F = \langle R, A \rangle$. Therefore institution $\mathcal{I}$ can be represented by a tuple $\langle I, F \rangle$. The F component fully describes the facts and rules which currently hold in the institution.

This gives us the most general view of agent communication languages; by specifying the tuple $\langle I, F \rangle$ we can specify any ACL. It describes how institutional facts F change in response to events as the multi-agent system runs. Given an institution described by $\langle I, F_0 \rangle$ at some instant, and a subsequent sequence of events $e_1, e_2, e_3 \ldots$, we can calculate the description of the institutional facts after each event, obtaining a sequence of facts descriptions: $F_0, F_1, F_2, \ldots$, where each F_i is related to F_{i-1} as follows: $F_i = update_{i-1}(e_i, F_{i-1})$ where $update_{i-1} = I(R_{i-1})$ (and $F_i = \langle R_i, A_i \rangle$ for all i). Interpreter I remains fixed throughout runs.

2.1 A Universal Agent Communication Machine

The rule interpreter I specified above is the immutable part of an institution. The choice of I can place limits on what is possible with that institution, or give it universal expressive power. Just as a universal Turing machine can simulate the behaviour of any Turing machine, we can have an analogous universal agent communication machine.

Definition 1. *A universal agent communication machine is a machine which can simulate the behaviour of any computable agent communication language.*

By "simulate" here we mean that (given an appropriate set of input rules) it could generate the same sequence of institutional facts in response to the same sequence of events. In fact a universal Turing machine is a universal agent communication machine. The input R to the machine produces the function *update*.

Any update function that is computable can be produced in this way. Any Turing complete programming language can be used to implement a universal agent communication machine.

3 Specifying Extensible Languages

Given a universal agent communication machine it is possible to specify an ACL which has *universal expressive power*, in the following sense.

Definition 2. *An agent communication language is said to have universal expressive power if the agents using it can transform its rules so that it simulates the behaviour of any computable agent communication language.*

Given a language defined by an institutional specification $\mathcal{I} = \langle I, \langle R, A \rangle \rangle$ (as described above), if I is a universal agent communication machine, then the language will have universal expressive power if the rules R allow messages sent (i.e. events) to program the machine in a Turing complete way.[4] Languages with universal power are of particular interest because they allow unlimited extension. It is our thesis that a minimal language with universal expressive power is an appropriate basis for standardising agent communication; i.e. the specification of the programming language and core code can be agreed upon and published. Such a choice of standard does not restrict agents to the rules given because it can provide a mechanism through which agents can change the rules at runtime; this can allow agents to introduce new protocols at runtime, for example. Such protocols could come from trusted libraries, or could be generated by the agents on the fly for the scenario at hand. If necessary, agents could also have a phase of negotiation before deciding on accepting some new rules.

We define one such language in Fig. 1. We make use of Prolog as the logic programming paradigm is particularly appropriate for agent communication; there is also evidence that Prolog already enjoys considerable popularity in the agent communication semantics community [1,20,14]. The `interpretEvent`/3 predicate takes as input the current set of facts `F` and an event `Event`, and generates the new set of facts `NewF`. In line *3* the event is converted from its predicate form to a list form, so that line *4* can append the old and new facts variables to it. In line *5* the event is converted back from list form to its predicate form. The next step will be to match the head of the event with the appropriate rule in `Rules` (this corresponds to R in the formal model); however, we do not want to change the rule itself by unifying its variables, this is why we make a clean copy of it in line *6* before doing the matching in line *7*, via the `member`/2 predicate. Now that the body of the rule (`Tail`) has been retrieved, we can invoke it in line *8* via `callPred/2`. Lines *9* to *17* define the recursive `callPred/2` predicate. Line *10* handles the case where the rule body to be executed invokes another rule within `Rules`, in which case `callPred/2` is called to handle it. Line

[4] This expressiveness implies undecidability, hence it may be impossible to prove properties for a system of agents using the language. However, if desired one can specify a restricted and decidable language on top of this, by restricting the agents' ability to modify rules, as described in the sequel.

```
1  interpretEvent(F,Event,NewF):-
2     F=[Rules,Asserts],
3     Event=..EventAsList,
4     append(EventAsList,[F,NewF],NewEventAsList),
5     Pred=..NewEventAsList,
6     copy_term(Rules,Rules2),
7     member([_|[Pred|Tail]],Rules2),
8     callPred(Tail,Rules).

9  callPred([],_).
10 callPred([HeadPred|Tail],Rules):-
11    copy_term(Rules,Rules2),
12    member([_|[HeadPred|NestTail]],Rules2),
13    callPred(NestTail,Rules),
14    callPred(Tail,Rules).
15 callPred([HeadPred|Tail],Rules):-
16    call(HeadPred),
17    callPred(Tail,Rules).
```

Fig. 1. Extensible Communication Language in Prolog

15 handles the case where the rule body to be executed invokes a built in Prolog predicate, in which case it is called directly via `call/1`. It is important that `interpretEvent`/3 forces the event to use a rule from `Rules` (i.e. it checks that the rule is a member of `Rules` before passing control to `callPred/2`) so that agents are unable to directly invoke Prolog predicates with their messages; their messages are interpreted first. Without this precaution our interpreter would not truly have universal expressive power, as it would always accept Prolog predicates, which could be used to reprogram it; hence it would be impossible to define a language which restricted the possible things which events could change.

Rules stored in R are written in the form of lists, with an index number at the head of each rule. A Prolog clause of the form "`pred1(A,B):-pred2(A), pred3(B)`" becomes "`[1,pred1(A,B),pred2(A),pred3(B)]`". This corresponds to the Horn clause $pred2(A) \wedge pred3(B) \rightarrow pred1(A, B)$. Some sample rules are:

```
[  [ 1, addRule(Rule,[R1,A1],[NewR1,A1]),
        append(R1,[Rule],NewR1)                ],
   [ 2, deleteRule(Index,[R2,A2],[NewR2,A2]),
        delete(R2,[Index|_],NewR2)             ]
]
```

Let the above program be called *prog* and the interpreter $I = \langle prog, Prolog \rangle$. Let the assertions A be initially empty and rules R be the two rules above.

Theorem 1. *The ACL specified by institution $\langle I, \langle R, A \rangle\rangle$ has universal expressive power.*

The truth of this follows from that fact that Prolog is Turing-complete, and `addRule` can be used to add arbitrary predicates, and can therefore give subsequent events access to the underlying Prolog language (or restrict their access). Despite the ease with which this can be done, to our knowledge this is the first

example of such an ACL. We propose that an ACL such as this would form the core component of a standard. This is only the first step of standardisation however. Standards will also need to define libraries and tools which will make the base machine more usable.

Let us briefly illustrate how we can begin to use the above ACL. The following is an example of an event:

```
addrule([3,assert(Fact,[R,A],[R,[Fact|A]])])
```

After interpreting this event, the rules R will be updated so that subsequent `assert` events cause the addition of an element to the assertions A. For example, a subsequent event `assert(alive(agent1))` would add `alive(agent1)` to A. Note that this is invoking our rule 3 and not Prolog's built-in `assert`/1 predicate. At this point we will avoid giving an extended example of the kind of interaction we can capture. Instead we want to show the component based approach to standardisation, so we will eventually illustrate only a very simple auction protocol, but we will build it upon some useful components.

We now add some basic "housekeeping" rules. We will have a *timer* predicate in A, which records the current time, e.g. *timer(524)*. We will assume that our agent platform generates timer events at regular intervals. Whenever a timer event happens we want to update the clock and execute a set of housekeeping rules. These rules perform housekeeping checks, for example to see if an agent has failed to meet a deadline. The following rule (in R) handles the timer event:

```
[ 3, timer(Time,[R,A],[NewR,NewA]),
     replace(A,timer(Time),UpdatedA),
     housekeeping([R,UpdatedA],[NewR,NewA])
]
```

Here we have assumed the existence of a *replace* predicate which replaces a named predicate in A with a new version. The initial *housekeeping* predicate simply preserves the institutional facts F; subsequent components will modify the predicate, adding their own rules.

It is desirable to add another layer for the interpretation of agent communications. We create a *speechAct* rule for this purpose. Agents communicate by sending messages (events) of the form *speechAct(sender, receiver, performative, content)*. We must rely on the platform's message handling layer to discard any messages where the *sender* parameter is falsified; there is no way to do this once the event is processed by the interpreter. We also rely on the platform to distribute the message to the intended recipients. The message event is then handled by our *speechAct* rule. With this in place we protect the lower level operations from direct access by the agents. We do not want agents to be able to directly invoke the timer event or the rule changing events; however, if desired, we can still create speech acts which allow the modification of certain rules in R by suitably empowered agents. Now the *speechAct* predicate becomes particularly useful to gather together all those operations which need to be done during the processing of any message (e.g. check roles, permissions and empowerments). This is described in Section 4.

It is worth noting that agents can only process the events they have observed; hence, when the update rule is implemented in agents, they only build a view of the institutional facts from their individual perspectives. If each agent applies the rules on limited information in this way, it is entirely possible that the institutional facts from the perspective of two different agents may have contradictory assertions. This is not a problem, so long as the developer bears this in mind when designing components (protocols for example). Specifications of norms should not create "unfair" rules, for example creating an obligation for an agent to do something, and leaving the agent unaware of the existence of the obligation. In most practical systems which we envisage, there will be no need for any agent to maintain this global view and indeed in a large system it might not be feasible to maintain it; it will be sufficient for each agent to maintain an individual perspective, which coincides with the perspective of other agents for any interactions they share.

Obviously we need to be particularly careful if we allow agents to change the rules R, lest conversational participants have contradictory beliefs about the meanings of the messages they are exchanging. At least two solutions can be envisaged: either all members of the institution need to be informed of any change, or a subgroup can decide to set up a virtual organisation (having new communicative actions and corresponding rules which apply only within that virtual organisation, the old ones still applying outside). The later solution is probably the more practical of the two.

4 Normative Relations Component

Various different notions are employed by institutions to describe their rules; Sergot distinguishes between notions of power, permission and practical possibility [13]. Power is the ability of a member to bring about changes in the institutional facts; i.e. for each event which changes F we can describe which members of the institution can effect those changes. For convenience it is common to define roles and define the power of a role. This is because the occupants of roles often change, while the powers associated with the role do not.

Permission can be used to describe those actions which ought or ought not to be taken. Permission is distinct from power because a member may be empowered to do something even though he is not permitted; in this case: if he does it then it counts, i.e. it creates the institutional fact. For example, an examiner could award a student a pass on submission which falls short of the required standards as set out by the institution. In this case the examiner's action is not permitted but still counts as a pass under the rules of the institution; the examiner may be subject to some sanction if the abuse is discovered, but this may not necessarily revoke the fact that the student has passed.

The notion of permission leads to its dual: obligation; obligation is equivalent to "not permitted not to". Obligation can be captured by a rule which specifies a sanction if an action is not done. Because we will be testing agents' compliance over finite models, we must always specify a time-limit for obligations. It is no

good for an agent to promise something and deliver "eventually", if there is no upper bound on the time taken.

Practical possibility is another distinct notion which some institutions may need to represent explicitly. For example, suppose there is a rule defining the sanction to be placed on a member in the case of failing to fulfil an obligation, there may be a need to exempt the case where the member was physically incapable of fulfilling the obligation at the time. Thus there could be institutional facts to represent the physical capabilities of each agent; i.e. a rule will define the events in the physical world which *count as* the agent being recognised by the institution as being capable or incapacitated. We do not implement practical possibility however.

4.1 Implementing Norms

The normative relations we implement are defined by predicates stored in the assertions A. Relations can apply to agents directly or via *roles*; an agent occupies one or more roles (also stored in the assertions A). There are four types of normative predicate: *power, permitted, obliged* and *sanction*. Sanctions are defined for actions which agents should not do. Permitted or obliged actions are treated as exemptions to these sanctions, i.e. the sanction applies unless the agent was permitted or obliged. Power and permission have arity 3: the first parameter is the agent name (or role name), the second is the performative of the speech act he is empowered/permitted to do, and the third is a condition. For example

```
power(bidder,bid,[Content=[C],C>47])
```

means that any agent in the role of bidder is empowered to send a *bid* speech act provided it complies with the following conditions: the content of the act must be a list containing a single number whose value is greater than 47. If the condition is the empty list then it is always true. Sanctions and obligations add a further (fourth) parameter, which is the "sanction code". Following [20] we will associate a 3-figure "sanction code" with each norm violation (similar to the error codes used in the Internet protocol HTTP), in our case higher numbers are used for more egregious violations. The sanction codes gathered by each agent as it commits offences are merely recorded in a list. The use of codes is just a convenient way to record sanctions without yet dealing with them; we would require a separate component to impose some form of punishment. Finally the obligation adds a fifth parameter which is the deadline by which the specified speech act must be sent.

The algorithm shown in Fig. 2 is added to the *speechAct* rule to handle the normative relations, it effectively defines an operational semantics for the normative relations. With this implementation we make obligation imply permission and power. It is in this algorithm that roles are consulted to retrieve the names of the agents occupying the roles; e.g. when checking if an agent who has just sent a message is obliged (and hence permitted), the algorithm will consult the facts to see what roles the sending agent occupies. We also need to add the

following to the *housekeeping* rule (recall that the housekeeping rule is invoked on every timer event):

- For each obligation check if it has timed out. If so, apply the sanction to the agent (or all agents occupying the obliged role) and remove the obligation from A.

Note that we are assuming that the existence of a *speechAct* rule is an agreed standard across component developers, so that any new components can add checks and guards to this rule.

algorithm HANDLE-NORMS

1. Input: a speech act with *Sender, Receiver, Performative, Content*
2. Check if there is an obligation which requires that *Sender* (or one of the roles he occupies) send this speech act. If so remove the obligation from A and jump to 5.
3. Check if there is a sanction for *Sender* (or one of the roles he occupies) sending this speech act: If not, go to the next step; If so,
   - check if there is a permission for *Sender* (or one of the roles he occupies) to send this speech act: If so, go to the next step; If not, apply the specified sanction.
4. Check if *Sender* (or one of the roles he occupies) is empowered to send this speech act: If not, discard the act and exit this algorithm.
5. Process the act as normal.

Fig. 2. Algorithm to Handle Normative Relations

5 Protocol Components

Protocols are additional components of the ACL, they are each encoded via their own rules in R. Each protocol has a unique name and may be represented by a number of clauses in R. Protocols essentially determine what actions are to be taken next, given the current state and an event that happens. They do this by consulting the current state and modifying the normative relations according to the event that has just happened. Agents initiate protocols by using the special speech act *initProtocol*; the *speechAct* predicate passes control to the protocol on initiation. A protocol initiates a "sub-conversation" within the institution. All the assertions describing the protocol's state of execution are gathered together as an indexed list within A. In order to ensure the index is unique, the initiator will compose an index by concatenating his name with a positive integer which increases with each new protocol he initiates. Subsequently all speech acts indexed with the protocol's identifier will be processed by the protocol's rules (instead of the standard rules which process speech acts that are not part of any protocol). Normative relations defined within the protocol's "space" in A only apply to messages that are part of that protocol. Timer events are processed by all protocols running at any time. Agents are free to enter multiple parallel protocols, each being a separate sub-conversation. Sending a *exitProtocol* message terminates the protocol and removes its assertions from A.

5.1 Example Protocol: Auction

The Vickrey auction protocol below is expected to be invoked by a speech act with content [*Index,Protocol,Item,OffersOver,ClosingTime*]. These then become variables accessible to the initiator clause of the protocol rule, along with the initiator of the protocol and the list of receivers. Each clause has access to the variables *Sndr* and *Rcvr* from the event that invoked the clause (we cannot use the names *Sender* and *Receiver* as these are used by content checking conditions). The Prolog-style pseudocode below describes a series of clauses, one to handle each speech act that can happen during the execution of the protocol. To keep the presentation concise we have avoided presenting the example in real Prolog code.

```
initiator:
 add role(Sndr,auctioneer)
      // sender of initiating message takes on role of auctioneer
 for each Rcvr add role(Rcvr,bidder)
      // each receiver of initiating message takes on role of bidder
 add power(bidder,bid,[Content>OffersOver])
 add permitted(bidder,bid,[Receiver=R,
                role(R,auctioneer),Content>OffersOver])
      // bidders are empowered and permitted to bid
      // provided the content satisfies the specified constraints
 add sanction(bidder,bid,[],100)
      // any other bid incurs a sanction
 retrieve global.timer(Time)
 add timeout(closingTime+Time)
      // timeout is simply a newly defined predicate in the social facts
 add item(Item)
 add high1(0)
 add high2(0)

if bid([auctioneer,NewBid])
 retrieve high1(High1)
 retrieve high2(High2)
      // the current highest and second highest bids
 if NewBid>High1 then replace winner(_) with winner(Sndr)
                      replace high1(_) with high1(NewBid)
 else if NewBid>High2 then replace high2(_) with high2(NewBid)

if timer(Time)
 retrieve timeout(T)
 if Time>T then
  retrieve high1(High1)
  retrieve high2(High2)
  NewTime = Time+50
  if High1=High2 then
      // if nobody has bid then exit
   obliged(auctioneer,exitProtocol,[Receiver=bidder],101,250)
  else
      // else declare the winner
```

```
   remove power(bidder,bid,_)
   add power(auctioneer,inform,[])
   retrieve winner(Winner)
   add obliged(auctioneer,inform,
               [Receiver=Winner,Content=[won,High2]],103,NewTime)
   add obliged(auctioneer,exitProtocol,
                [Receiver=[bidder]],101,NewTime)

if inform([won,Price]) then
 retrieve global.timer(Time)
 retrieve winner(Winner)
 retrieve item(Item)
 NewTime = Time + 150
      // this is the time by which payment must be made
 add global.obliged(auctioneer,inform,
    [Receiver=bank,Content=[transfer,Item,Winner]],102,NewTime)
 add global.obliged(Winner,inform,[Receiver=bank,
    Content=[credit,Price,auctioneer]],102,NewTime)
```

Note that the final clause creates obligations which are to persist after the protocol's termination (this is the meaning of `add global...`). When this is done the agent's name is put in the obligation instead of the role name. This is because the role will cease to exist on termination of the protocol (i.e. the fact asserting it is within the indexed list of facts for that protocol, and hence will be deleted when the protocol terminates), whereas we want the agent to still be obliged to pay even after the auction is finished.

5.2 Auction Animation

The initiating speech act is

```
speechAct(alice, [bob,claire], initProtocol,
            [alice1,auction,IPRowner,47,200])
```

Here *initProtocol* is the performative and *auction* is the protocol to be initiated. This starts a new conversation state, having its assertions as an indexed list within A. The index is *alice1*. Subsequent messages which are part of this protocol execution must be tagged with this index at the head of their content list. After this the following assertions hold within the indexed list

```
role(alice,auctioneer)
role(bob,bidder)
role(claire,bidder)
power(bidder,bid,[Content=[C],C>47])
power(auctioneer,exitProtocol,[Receiver=[bob,claire]])
permitted(bidder,bid,[Receiver=R,role(R,auctioneer),Content=[C],>47])
sanction(bidder,bid,[],100)
```

The next speech act is a bid by *bob*:

```
speechAct(bob, alice, bid, [alice1,53])
```

The only effect of this is to add a predicate recording this as the highest bid. Bidders still retain the power and permission to revise their bids. Next we have *claire* bidding 51, which adds a predicate recording the second highest bid. Then the timeout event happens. This results in the power to bid being revoked. Agents are still permitted to bid, but it has no effect. We now have the following norms for the auctioneer:

```
power(auctioneer,inform,[])
obliged(auctioneer,inform,[Receiver=bob,Content=[won,51]],103,250)
obliged(auctioneer,exitProtocol,[Receiver=[bob,claire]],101,250)
```

Note that the auctioneer is empowered to inform anything to the bidders; whatever he says, it counts. However, he is obliged to announce the winner and losers as expected in a Vickrey auction. The auctioneer's next messages are

```
speechAct(alice, bob, inform, [alice1,won,52])
speechAct(alice, [bob, claire], exitProtocol,[alice1])
```

This terminates the protocol and generates two obligations which exist in the "root" of A, i.e. not in the sublist indexed by *alice1*.

```
obliged(alice,inform,[Receiver=bank,
         Content=[transfer,IPRowner,bob]],102,400)
obliged(bob,inform,[Receiver=bank,Content=[credit,52,alice]],102,400)
```

Note that *alice* has overcharged *bob*. Without any third party monitoring, there is no way for him to know. However, we could imagine a subsequent dialog where *claire* reveals her bid to him and he lodges a complaint with an arbitration authority. If the evidence is deemed to be sufficient, the protocol specification can be consulted again to determine the appropriate sanction, i.e. that sanction 103 should be enforced on *alice*.

6 Temporal Logic Component

Our *obliged* predicate only allows us to specify that an action must be done before some future time. We have also added a temporal logic component which allows us to express more complex conditions. For example we can specify that an agent is obliged to ensure that a certain condition holds true, where the condition is expressed in a simple temporal logic (a subset of LTL), but ultimately refers to the truth values of predicates in A. We are interested in making these kinds of normative specifications for the behaviour of agents, and then testing their compliance with the specification by observing their behaviour at runtime (by observing finite runs of the system). This is the same type of testing as described by Venkatraman and Singh [24]; i.e. given an observed run of the system we can determine if the agents have complied so far, but not if they will comply in other circumstances. We have found that the standard temporal logic operator $\Diamond$ (at some time in the future) is not very useful for our specifications. The linear temporal formula $\Diamond p$ promises that p will eventually true, but there is no way to falsify it in a finite model; i.e. if we require that an agent perform some action eventually, it is not possible to be non compliant in a finite model. Hence

this formula becomes meaningless when referring to agent behaviour in a finite observed sequence. A typical type of formula we need to specify is that some condition must hold continuously before a deadline. This can be done with the $\mathcal{U}$ (until) operator. The formula $p\mathcal{U}q$ means that p must hold continuously until q becomes true, and q should eventually become true (this second part is of course redundant in our finite sequences). We include Boolean connectives $\neg, \wedge$ and $\vee$ in the language. The $\Box$ operator (now and at all future times) is not included in our language because, in a finite sequence, $\Box p$ is the same as $p\mathcal{U}false$. Despite the fact that our temporal logic only has one temporal operator, it is still quite expressive, as nestings of $\mathcal{U}$ can be used, as well as the Boolean connectives.

Using this simple language we have constructed a model checking component which keeps track of the temporal formulae which an agent is obliged to keep true; i.e. the checker "carries forward" the pending formulae and checks each new state as timer events are processed. This allows the formulae to be used in normative specifications, and sanctions to be triggered automatically when the formulae are falsified. We use the method of particle tableau from [18] to check the formulae. This allows an efficient incremental construction of the relevant portion of the tableaux.

7 Related Work

There are few recent works which address standardising agent communication semantics. It appears that the effort has been abandoned since the attempts of FIPA and KQML [8,15,7] in the 90's. However, in terms of technical ideas, there are some recent proposals which are moving in directions similar to what we propose.

In [10] it is demonstrated that a system of production rules can be used to implement many agent institutions that had originally been specified with very diverse formalisms. This is similar to our proposal as it is given a common computational grounding to proposals which were previously hard to compare. It also shows that if we are considering computational implementations of agent communication, then one simple language will be sufficient to implement whatever diverse notions we choose to employ to govern the agents.

In [20] the possibility of agents modifying the rules of the institution is mentioned; it is stated that this would require "interpretable code or some form of dynamic compilation". In [2] the event calculus formalism has been implemented to animate a specification of a rule governed agent society, but it is also stated that features of the underlying programming language could be made accessible to complement the event calculus formalism; this comes closer to the flavour of our proposal. In [11] normative relations are implemented in the Jess production rule system. The authors mention the possibility of "societal change", where societies may "evolve over time by altering, eliminating or incorporating rules". This societal change facility is not actually implemented in [11], but the authors do specify norms in a computationally grounded language based on observable phenomena.

In [6] there is a proposed development methodology which is similar to the "component based" aspect of our approach; generic protocols are specified, and then *transformers* can be applied to them to capture variations of the protocol for specific contexts. The work of [26] advocates the need for tools to assist developers in protocol design, while also showing how protocols can be built on the social commitments approach to agent communication semantics; this type of tool support and structured development is exactly what we expect will be needed to take our standardisation approach forward.

8 Conclusions and Future Work

There are two requirements which should be fulfilled as a precondition to making a standard for agent communication which has a reasonable prospect of actually being adopted. One is the expressive power to allow developers to do what they want, and the second is the ease of use (for which tools are required). The first aspect is easy, as we have shown, the second will take more effort. Even with the few components we described, we can see already that programming moves to a higher level as we add more components. We expect that standardisation will need to proceed by means of evolving libraries and tools which make the agent developers job easier. In this process the role of a standards body would be to accredit components and publish them, and to standardise the form of their documentation.

One avenue for future work is to explore the possibility of creating a component which defines a more intuitive semantics for common speech acts. The auction example above is a very mechanistic protocol, in that there is little room for flexibility and innovation among the agents. The social facts we created were solely concerned with describing sufficient information for the execution of the auction. Thus the semantics of these messages was purely procedural. In more flexible protocols we could create social facts which capture more of a natural human style of communication. For example, a "request for information" message sent by an agent could create a social fact which describes that the sender has expressed a desire to know something. Other participants in the society could then be creative in how they respond, because they can recognise the need of the speaker, rather than being constrained to only reply according to some rigid protocol, as would be the case for procedural semantics.

Another direction we are currently experimenting with is our temporal logic component which model checks temporal logic formulae. We plan to extend the expressiveness of its language. Argumentation is yet more interesting as it typically requires the use of nonmonotonic logics: an agent may undercut another agent's argument, and so force a conclusion to be retracted. Here we would code the rules defining acceptability of an argument. The ability of a meta-interpreter to specify a depth limit on proofs is particularly useful for this purpose; in order to have a common consensus on what arguments are accepted we need to specify the limits on the resource bounded reasoning [17]. Argumentation also introduces the possibility of negotiating changes to the rules of the institution itself. There will also no doubt be considerable interest in developing components for various

logics such as the C+ action language (which is gaining popularity [2,6]) and various modal logics.

Eventually it is hoped that different electronic institutions could publish the components which comprise their communication language in a machine readable format, so that a roaming agent could come and join the institution without needing to be programmed to use that particular language in advance. This is an ambitious goal, as the agent would need not to just know the rules, but also its strategy for participation. However, if we restrict our attention to certain types of dialog, and their variants (e.g. auctions) then it does seem feasible.

References

1. Agerri, R., Alonso, E.: Semantics and Pragmatics for Agent Communication. In: Bento, C., Cardoso, A., Dias, G. (eds.) EPIA 2005. LNCS (LNAI), vol. 3808, Springer, Heidelberg (2005)
2. Artikis, A., Sergot, M., Pitt, J.V.: Specifying Norm-Governed Computational Societies. Technical Report 06-5, Dept. of Computing, Imperial College, London, UK (2005)
3. Austin, J.L.: How To Do Things With Words. Oxford University Press, Oxford (1962)
4. Bentahar, J., Moulin, B., Meyer, J.-J.C., Chaib-Draa, B.: A Computational Model for Conversation Policies for Agent Communication. In: Leite, J.A., Torroni, P. (eds.) Computational Logic in Multi-Agent Systems. LNCS (LNAI), vol. 3487, Springer, Heidelberg (2005)
5. Chaib-Draa, B., Labrie, M.-A., Bergeron, M., Pasquier, P.: An Agent Communication Language Based on Dialogue Games and Sustained by Social Commitments. Autonomous Agents and Multi-Agent Systems 13(1), 61–95 (2006)
6. Chopra, A.K., Singh, M.P.: Contextualizing Commitment Protocols. In: AAMAS. Procs. 5th. Int'l Conf. on Autonomous Agents and Multi-Agent Systems, ACM Press, New York (2006)
7. Cohen, P.R., Levesque, H.J.: Communicative Actions for Artificial Agents. In: Int'l Conf. on MASs, pp. 65–72. MIT Press, Cambridge (1995)
8. FIPA. [FIPA OC00003] FIPA 97 Part 2 Version 2.0: Agent Communication Language Specification. In: Website of the Foundation for Intelligent Physical Agents (1997), `http://www.fipa.org/specs/fipa2000.tar.gz`
9. Fornara, N., Vigano, F., Colombetti, M.: Agent communication and artificial institutions. Autonomous Agents and Multi-Agent Systems (2006), doi:10.1007/s10458-006-0017-8
10. Garcia-Camino, A., Rodriguez-Aguilar, J., Sierra, C., Vasconcelos, W.: A Rule-based Approach to Norm-Oriented Programming of Electronic Institutions. SIGEcomm Exchanges 5(5) (2006)
11. Garcia-Camino, A., Rodriguez-Aguilar, J.-A., Noriega, P.: Implementing Norms in Electronic Institutions. In: AAMAS. Procs. 4th Int'l Conf. on Autonomous Agents & Multiagent Systems, ACM Press, New York (2005)
12. Guerin, F., Pitt, J.V.: A semantic framework for specifying agent communication languages. In: ICMAS 2000. Fourth International Conference on Multi-Agent Systems, pp. 395–396. IEEE Computer Society, Los Alamitos (2000)
13. Jones, A.J.I., Sergot, M.J.: A formal characterisation of institutionalised power. Journal of the IGPL 4(3), 429–445 (1996)

14. Labrou, Y.: Semantics for an agent communication language. PhD thesis, Baltimore, MD: University of Maryland Graduate School (1996)
15. Labrou, Y., Finin, T.: A semantics approach for kqml – a general purpose communication language for software agents. In: CIKM 1994. Third International Conference on Information and Knowledge Management, pp. 447–455 (1994)
16. Labrou, Y., Finin, T., Peng, Y.: The current landscape of agent communication languages (1999)
17. Loui, R.P.: Process and policy: Resource-bounded nondemonstrative reasoning. Computational Intelligence 14(1), 1 (1998)
18. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems (Safety), vol. 2. Springer, New York (1995)
19. Mayfield, J., Labrou, Y., Finin, T.: Desiderata for agent communication languages. In: AAAI 1995 Spring Symposium. Proceedings of the AAAI Symposium on Information Gathering from Heterogeneous, Distributed Environments, pp. 347–360. Stanford University, Stanford (1995)
20. Pitt, J., Kamara, L., Sergot, M., Artikis, A.: Formalization of a voting protocol for virtual organizations. In: AAMAS 2005. Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multi-Agent Systems, Utrecht, ACM Press, New York (2005)
21. Searle, J.R.: What is a speech act? In: Martinich, A.P. (ed.) Philosophy of Language, 3rd edn., Oxford University Press, Oxford (1965)
22. Singh, M.: Agent communication languages: Rethinking the principles. IEEE Computer 31(12), 40–47 (1998)
23. Singh, M.: A social semantics for agent communication languages. In: IJCAI Workshop on Agent Communication Languages, Springer, Berlin (2000)
24. Venkatraman, M., Singh, M.P.: Verifying compliance with commitment protocols: Enabling open web-based multiagent systems. Autonomous Agents and Multi-Agent Systems 2(3), 217–236 (1999)
25. Verdicchio, M., Colombetti, M.: A logical model of social commitment for agent communication. In: Proceedings of the second international joint conference on Autonomous agents and multiagent systems table of contents, Melbourne, Australia, pp. 528–535 (2003)
26. Yolum, P.: Towards design tools for protocol development. In: AAMAS 2005. Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems, pp. 99–105. ACM Press, New York (2005)