A Component-Based Approach to Standardising Agent Communication

Frank Guerin  
Dept. of Computing Science, Univ. of Aberdeen  
Aberdeen AB24 3UE, United Kingdom  
fguerin@csd.abdn.ac.uk

Wamberto Vasconcelos  
Dept. of Computing Science, Univ. of Aberdeen  
Aberdeen AB24 3UE, United Kingdom  
wvasconcelos@acm.org

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 dialogues and normative relations. 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 the appropriate components.

Categories and Subject Descriptors
I.2.11 [Artificial Intelligence]: Distributed Artificial Intelligence—Multiagent systems

General Terms
Standardization, Languages

Keywords
Agent communication languages and protocols; Artificial social systems: Conventions, Norms, Institutions

1. INTRODUCTION

We aim at facilitating interoperability of 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. 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 have fared poorly regarding the semantics of languages used. Implementers who claim to use a particular standard ACL tend to ignore those aspects of the standard that pose difficulties for their implementation; 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 [4], which will not be understood by other systems. 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 ACLs which are universal in the sense that they could be used to simulate any other computable ACL. We propose 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. 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 other programming languages. We do not advocate that the components described here be adopted as a standard; we merely provide simple examples to demonstrate the feasibility of the component-based approach.
2. TOWARDS A GENERAL ACL

We define an ACL by specifying an institution. Institutions regulate the activities of their members by inventing institutional facts [3]. 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. 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. Institutional rules may be 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. 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.

A further point to note is that the institutional facts being modified by a rule could be states 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). Typical events include messages being transmitted, timer events and other non communicative actions of agents.

Let \( E \) be the set of possible events and let \( F \) be the set of possible institutional facts. Let \( \text{update} \) be a function which updates the institutional facts in response to an event, \( \text{update} : E \times 2^F \rightarrow 2^F \). Now in an institution \( I \), 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 \( I \) can then be fully specified by specifying the interpreter \( I \) and the facts \( F \subset F \). Recall that \( F \) is itself composed of the rule type of facts \( R \) and the state of affairs type of facts \( A \), so \( F = (R, A) \). Therefore institution \( I \) can be represented by a tuple \( (I, F) \). 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 \((I, F)\) 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 \((I, F_0)\) 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: \( F_i = \text{update}_{e_i}(F_{i-1}) \) where \( \text{update}_{e_i} = I(R_{e_i}) \) (and \( F_i = (R_{e_i}, A_{e_i}) \) for all \( i \)). The interpreter \( I \) remains fixed throughout all runs.

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.

Def. 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 \( \text{update} \). Any update function that is computable can be produced in this way. Any Turing complete programming language can be used as a universal agent communication machine.

3. SPECIFYING EXTENSIBLE ACLS

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

Def. 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 \( I = (\langle I, F \rangle) \) (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. 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 now define one such language. 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, 2]:

\[
\begin{align*}
\text{interpretEvent}(F, \text{Event}, \text{NewF}) : & = \text{if } F = \langle \text{Rules}, \text{Asserts} \rangle \text{ then } \langle \text{EventAsList} \rangle, \\
& \text{append(\text{EventAsList}, \text{NewF}, \text{NewEventAsList})}, \\
& \text{Pred=..\text{NewEventAsList},} \\
& \text{member([..\text{Pred}[\text{Tail}]], \text{Rules2})}, \\
& \text{callPred(Tail, Rules)}.
\end{align*}
\]

\[
\begin{align*}
\text{callPred}([..]) : & = \text{copy_term(Rules, Rules2)}, \\
& \text{member([..\text{Head}][\text{Tail}], Rules2)}, \\
& \text{callPred(Tail, Rules)}.
\end{align*}
\]

The \text{interpretEvent/3} predicate invokes \text{member/2} to find the appropriate predicate to match the event (i.e. find it.
in $R$). This is important 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 “\( \text{pred1}(A,B) \rightarrow \text{pred2}(A), \text{pred3}(B) \)" becomes \( \{1, \text{pred1}(A,B), \text{pred2}(A), \text{pred3}(B)\} \). This corresponds to the Horn clause \( \text{pred2}(A) \lor \text{pred3}(B) \rightarrow \text{pred1}(A,B) \). Some sample rules are:

\[
\begin{align*}
&1. \text{addRule}(\text{Rule},[R1,A1],[\text{NewR1},A1]), \\
&2. \text{append}([R1],[\text{Rule}],[\text{NewR1}]), \\
&3. \text{deleteRule}([\text{Index},[R2,A2],[\text{NewR2},A2]], \\
&\text{delete}([R2,[\text{Index}],[\text{NewR2}]]))
\end{align*}
\]

Let the above program be called prog. Let the interpreter machine $I = (\text{prog,Proglo})$. Let the assertions $A$ be initially empty and the rules $R$ containing only the two rules above.

**Theorem 1.** The ACL specified by institution \( (I, \{R, A\}) \) has universal expressive power.

The truth of this follows from the 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.

4. **A NORMATIVE COMPONENT**

We can now start to add components on top of the above core language. We first consider a normative relations component, which we implement by rules in $R$ which define how events affect normative relations, and how the normative relations influence the interpretation of events. The normative relations themselves are represented by predicates 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. Obligation additionally requires a rule which specifies a sanction if an action is not done. We want to test agents’ norm-compliance over finite models, hence we must always specify a time-limit for obligations: it is no good for an agent to promise something, if there is no upper bound on the time taken. 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 use roles and define the power of a role. This is because the occupants of roles often change, while the powers associated with the role to not. Relations can apply to agents directly or via roles; an agent occupies one or more roles (also stored in the assertions $A$).

The following algorithm effectively provides normative relations with an operational semantics:

**Algorithm handle-norms**

1. input: a speech act event with Sender, Receiver, Performative, Content
2. Check if there is an obligation requiring that Sender (or one of its roles) send this speech act. If so remove the obligation from $A$ and go 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 step 4; 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.

With this implementation we make obligation imply permission and power. We also need to add the following to the housekeeping rule which will be invoked on every clock tick (timer event): “For each obligation check if it has timed out. If so, apply the sanction to the agent and remove the obligation from $A$.”

5. **PROTOCOL COMPONENTS**

We view protocols as additional components of an agent communication language, encoded via their own rules. 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 (e.g. in an auction: making the auctioneer obliged to announce the highest bidder as winner when the auction times out). 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. Normative relations defined within the protocol’s “space” only apply to messages that are part of that protocol. Agents are free to enter multiple parallel protocols, each being a separate sub-conversation.

6. **CONCLUSIONS**

Even with the few components we described, we can see already that programming moves to a higher level as we add more components (e.g. defining protocols by making use of the normative relations). We expect that standardisation will need to proceed by means of evolving libraries and tools [5, e.g.] 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.

7. **REFERENCES**