Implementing Social Norms using Policies

Rob Kremer
Department of Computer Science
University of Calgary
2500 University Dr.
Calgary, Alberta, Canada, T2N 1N4
Email: kremer@cpsc.ucalgary.ca

Abstract—Multi-agent systems are difficult to develop. One reason for this is that agents are embedded in a society where all agents must agree to obey certain social norms in order for the society to function. Thus, different programmers, writing different agents, must carefully obey certain agreed-upon protocols. This problem is difficult enough due to the complexity of the interactions, but it is exacerbated by the asynchronous and event-based nature of agent-based systems: agents must asynchronously respond to incoming conversational messages, and may carry on several simultaneous conversations.

Several large projects address these issues. Examples are Jade (Telecom Italia) and Cougaar (DARPA). Jade is strictly compliant with the well-known FIPA standard, which makes it useful for commercial agent development and research not directed at certain fundamental aspects of multi-agent systems. Cougaar was developed as a defense agent infrastructure, and while it is not tied to FIPA standards, it is quite prescriptive in both its inter-agent architecture, and its intra-agent architecture.

The contribution of CASA (Collaborative Agent System Architecture) is an agent infrastructure that seeks to support agent development, but as much as possible, avoids restricting the intra-agent architecture, and its intra-agent architecture. This paper describes aspects of the CASA tool that mitigate the aforementioned problems for the research-oriented developer who wants to investigate deviations from standards or alternative architectures. CASA provides a policy descriptor language that abstracts the complexities of conversational interactions away from the programming level, and allows sharing of policies among different agents, even at run time. Thus, an agent programmer is free to concentrate on the properties of the agent, and not on the intricate mechanics of conversational protocols. In addition, policies may be easily modified and distributed as the need arises. Thus, a protocol researcher can concentrate on protocols without having to re-write agent behaviour each time the protocol changes. The policy approach is very flexible, and we have developed policies to support the social commitment paradigm, the BDI paradigm, as well as simpler ad-hoc protocols.

I. INTRODUCTION

CASA (Collaborative Agent System Architecture) [1] is an agent development platform who's main contribution is its support of the flexible development of societies of agents such that researchers and others can experiment with various agent communication protocols and social organization. The flexibility described here is multi-faceted: At the intra-agent level, developers must be able to implement the inner workings of their agents in a variety of ways to implement specific behaviours. At the social (inter-agent) level, developers must be able to describe social norms among agents at a global scale, independent of individual agents. In addition, CASA endeavors to offer basic communication services, messaging services, command-line services, and plug-in logics and ontologies. All this flexibility can lead to an unwieldy system that could be difficult to use, so the flexibility must be carefully balanced against ease-of-use, both for the developer and the end-user of the agent system. CASA is available for download at http://pages.cpsc.ucalgary.ca/~kremer/CASA/.

This paper concentrates on the CASA's policies, which are the prime tools a developer uses to specify social norms. Policies are described in detail in section III, but suffice here to say that policies (just in like in human societies) are rules that all the members of a organization or society know and follow.

In CASA, these social norms are implemented using policy rules. For example, a social norm would be “if someone makes a requested of you, you should respond in some way.” This sort of approach to the social norm problem is called social commitment theory [2], [3] and the obligations agents incur due to social norms are called shared social commitments.

For an agent to function in a society of agents, she must both pursue her own goals and respond to the other agents in a socially responsible manner (as defined by the society’s social norms). How she goes about pursuing her goals is her private concern. However, her interactions with other agents in her environment are constrained by the social norms of the society, and, for artificial agents, these social norms must be specifically encoded in a way that can be shared among all the agents participating in a society. This encoding is implemented in CASA as policies (see section III).

An agent spends its time checking for messages on the event queue and, if there are no messages, executing its doIdle() method. Since it's not really the topic of this paper, suffice to say that the doIdle() method is where the agent does whatever proactive things the creator of the agent deems it should be doing, i.e.: this is where the agent pursues its own goals.

On the other hand, if a message is available on the event queue, the the agent will instead apply its policies to the message. In the case of a Social Commitment agent, the agent's applyPolicies() method will merely invoke the appropriate policies to either instantiate new social commit-

1or should follow
2The formal policy is specified in figure 2.
Map can be treated as a simple dictionary (request the attributes (fields) of a message to values. Figure 1 shows the relevant policies.

A policy contains two components: a description of the event it matches (if a policy matches an event, it is said to be ready to fire), and an action to be taken if the policy is fired.

All this seems rather simple, but it is a rather powerful and flexible mechanism. Unfortunately, power and flexibility usually imply complexity. In particular, the system needs to define what to do under the following circumstances:

1) We may want a single policy to apply to many different specific kinds of messages without having to enumerate every single possibility.

2) More than one policy may match a particular message, and we need to either choose a particular policy out of the set, or allow multiple policies to be fired, but in an orderly fashion.

The first situation can be handled by using a hierarchical type system, and CASA does this in a straightforward manner [5]. The second situation, choosing what policies to execute, and in what order, is not so straightforward. The next sections describe CASA’s choices for these and other related design decisions: the event and message descriptions (section II), policy descriptions, and policy application (section III). Section IV provides a simple example, and section V describes related work.

II. EVENTS AND MESSAGES

CASA messages are implemented as an abstract class which can be treated as a simple dictionary (Map in Java) mapping the attributes (fields) of a message to values. Figure 1 shows a typical request message that the agent Alice might use to register itself with an agent registry agent.

The message in figure 1 is a KQML-syntax message which has a performative field (unlabeled – it’s always the first field in KQML syntax) with value request. The performative describes the conversational act of the message, that is, the conversational move the message is attempting to perform. Other examples of performatives are inform, agree, and subscribe. The act field has value register_instance of type action and describes the conversational act in more detail – in this case, it describes that the sender is requesting to the receiver (a Local Agent Controller) that the receiver register the sender’s instantiation. The act field may also be a list of actions, for example a reply to figure 1’s message might have performative agree and act request/register_instance, indicating that the LAC is agreeing to “the request to register the instance”, not to “register the instance” – a small semantic difference, but significant. The sender and receiver fields are intuitively obvious, but one should note that these fields normally contain complete URLs, but these have been abbreviated in the examples here due to column width restrictions. The reply-by, reply-with, conversation-id, and priority fields are all book-keeping fields, that are of a particular significance to the topic of this paper. The content field describes the real data associated with the message, and the notation used in the content field is described by the language field (here, the language is “casa.*”).

Receiving, sending, or merely observing the exchange of messages between other agents are events, just like any other event that an agent might need to deal with. The events are matched against policies and if any policies are applicable to the event (sending, receiving, or observing a message) then those policies are fired and actions are taken according to the policies’ specifications. Policies are described next.

III. POLICIES

Policies are really just rules or if-then clauses. When an event, such as the receipt of a message, is handled, it is matched against the agent’s set of policies, and if one or more of them matches the antecedent part, then the corresponding consequent part is executed. However, it’s not quite that simple. The process of matching is somewhat complex as it takes into account (1) type subsumption on the event types (which includes the message performatives and acts in the case of message events), and (2) an event matching multiple policies, where the policies must be ordered and linked to allow interaction among the policies (in a similar manner to method overriding in some object-oriented programming languages).

An example petition policy is given in figure 2. The petition performative type is a supertype of a the more familiar request type. Policies are specified in lisp-like syntax. At the top level, the petition policy in figure 2 may be read as an lisp expression with operator policy that takes two arguments: an antecedent and a consequent. The antecedent is a descriptor describing the applicability of this policy. The consequent part is a list of operations to execute if this policy is fired.

The use of the antecedent part is described in section III-A and the use of the consequent part is described in III-B.
A. Matching Policies to Events

A policy matches an event if its event type matches and the policy object's isApplicable() method returns true. Thus, the details of a match are flexible and delegated to the class that implements any particular policy type. While policies can match any type of event, the most common event type is MsgEvent; other event types can be SensorEvent, for reacting to sensor input, DeferredEvent for executing code at a later time, etc.

A MsgEvent policy matches using the performative and act fields in the event’s message. It first matches by determining if the performative of the event’s message is subsumed by the performative given by the policy’s event type (as specified in the first argument of the antecedent of policy – in the case of figure 2, this is petition). If this matches, it determines if the act of the event’s message is subsumed by the act given by the policy’s event type (in the case of figure 2, this is “a”, which is equivalent to T, top, Object, or “any”, matching any act at all).

Sometimes more than one policy is applicable to a particular event. For example, in the protocol given by Kremer, Florez & LaFornie [5], a request message instantiates a social commitment to reply, and an inform message instantiates a social commitment to ack (acknowledge). But request is subsumed by the type inform, so a request message instantiates both a social commitment to reply and another to ack. These two requirements are best implemented by two distinct policies (antecedents (MsgEvent request *) and (MsgEvent inform *)), both of which would fire on receiving a MsgEvent with performative type request

In the case of multiple policies firing on the same event, there are cases where the code of these policies must interact. For example, in the example in the preceding paragraph, the ack message required by the inform policy can be avoided altogether if it can determine that the request policy has already sent (or is going to send) a reply message (ack subsumes reply). It is therefore important to specify at least a partial order in which policies are going to fire (the order is only necessary in cases where the policies’ types have a subsumption relation between them). This is quite easily accomplished by sorting in the least-specific-first order for event type. In the case of MsgEvents, the order is by least-specific-first order for performative type, and then filtering out all but the most specific acts for each performative.

B. Policy Execution

The consequent part of a policy is a list of Lisp expressions that indicate the actions to take when the policy is fired. The simplest of these actions merely specifies a method invocation on an agent. However, we generally choose to provide much more specific action descriptors here. In particular, social commitment-based agents deal with messages exclusively by interpreting them in terms of instantiating new social commitments (operator Add), and marking existing social commitments as no longer applicable (operators Fulfil or Cancel). In the example in figure 2, the consequent is a list of only a single operation, an Add operation. There are several other operators important for social commitment-based agents:

- (Add scAttr ...)
- (AddFulfil scAttr ...
- (Cancel scAttr ...)
- (MarkFulfil scAttr ...)

All of these operators take any or all of several named arguments (the scAttr terms above) that, taken together, describe a social commitment. These named arguments are the following:

- :Debtor agent-descriptor
- :Creditor agent-descriptor
- :Performative performative-subtype-name
- :Act act-subtype-name

Either a type or a list of types detailing the commitment. The type describing the commitment. Must be a sub-type of performative (roughly equivalent to Seale’s speech acts [6] and Austin’s illocutionary acts [7]).

The reader might be wondering about the quote term in figure 2. Lisp expressions normally pass evaluated arguments to functions, but the list of actions in the consequent part of the policy is meant to be executed at policy evaluation time, not at policy declaration time, so the quote term is the Lisp way of suppressing this premature evaluation.
mitment (may be empty). All elements must be a subtype of action.

::ActionClass
A quoted string describing the fully qualified java class from which to instantiate an action for this class. This class must be a subclass of casa.socialcommitments.Action.

::ActionData
Optional supplementary information which will be passed to the constructor of the action class.

::DependsOn event
A description of an event that must occur before the social commitment can be executed. In figure 2 the event refers to an existing social commitment that this social commitment “depends on”, ie: this social commitment cannot execute until the one specified here has been fulfilled. :DependsOn differs from :If in that the debtor is held to the commitment regardless of the status of the event specified by the :DependsOn clause.

::If event
A description of an event that must occur before the social commitment can be considered has holding. Until the described event happens, this social commitment is not executable. :If differs from :DependsOn in that the debtor is never held to the commitment unless the event specified by the :If clause actually occurs.

::Shared
This social commitment is considered a shared social commitment. A non-shared commitment is typically only recorded by the debtor, and ignored by all other agents.

Thus, policies specify when they match against a message and what the agent should do in response to the message if they match. The reader may be wondering why the explanation has glossed over the seemingly important issue of whether the message as been received, sent, or merely observed between two other agents. Note, however, that this is not necessary as the semantics of the policies includes marking the social commitments debtor and creditor fields in terms of the sender and receiver of the message, so it doesn’t matter how the message is observed. To give a concrete example, if Alice is the sender of a request message and Bob is the receiver, then any observer (including Alice and Bob) would record exactly the same thing: that Bob is the debtor of the ensuing reply commitment, and Alice is the creditor.

IV. Example

One of the simplest examples of agent conversations is the request-reply protocol. To simplify slightly, this conversation involves agent Alice making a request to agent Bob, who chooses among several possible replies (he could choose agree, refuse, or notunderstood – all subtypes of reply). Only if he agrees, the conversation will go on with Bob proposing to Alice that the job is done, and Alice either rejecting or accepting Bob’s proposal. To keep the example short, we will deal only with he first two message exchanges in the conversation: Alice’s request, and Bob’s reply.

The conversation starts off with Alice sending a message to Bob requesting he send her a list of the files in Bob’s current directory:

```casa
(policy (MsgEvent * (*))
quote {
  (Add
    :Debtor msg.receiver // = Bob
    :Creditor msg.sender // = Alice
    :Performative (considerer) // = consider
    :Act msg.act // = execute
    :ActionClass "casa...DefaultConsiderObject")
})
```

The (MsgEvent * (*)) policy matches any message event no matter what the message. This doesn’t seem useful, and indeed it isn’t in this case. This policy fulfills any commitment to send a message like the one just sent. (If Alice has a commitment to reply to Bob, then if she replies to Bob that commitment is fulfilled.) However, in this case, the debtor, Alice, has no commitments so nothing happens (silently).

The (MsgEvent inform *) policy responds to all inform messages (because type inform subsumes type request), and has the effect of adding a social commitment for the receiver, Bob, to consider Alice’s request to execute a bash command script. Note that this Add commitment operator is not tagged with the :Shared marker, so is not a shared social commitment, but merely taken up by Bob. In a sense, this is nothing more than a way to call Bob’s internal code (the DefaultConsiderObject class) so Bob can “decide” whether or not to act on Alice’s request.

The (MsgEvent petition *) policy (see figure 2) responds to all petition messages since petition subsumes request, and has the effect of adding a social commitment for the receiver, Bob, to reply to Alice’s request. Note that the :Act has the message’s performative pushed onto it because Bob is not replying to an execute, but a request to execute. In addition, this new social commitment is dependent on the one that was just instantiated by the (MsgEvent inform *)
policy because Bob can’t reply until Bob has considered how to reply.

There now exist two social commitments instantiated by the above policies:

1. :Debtor Bob
   :Creditor Alice
   :Performative consider
   :Act execute
   :ActionClass "casa...DefaultConsiderObject"
   :State ready

2. :Debtor Bob
   :Creditor Alice
   :Performative reply
   :Act (request execute)
   :ActionClass "casa.policy...ReplyAction"
   :Shared
   :DependsOn 1
   :State ready

Since Bob is the debtor of both of them, he must choose which to act on. However, he can’t act on the second one because it is dependent on another social commitment that is not fulfilled. So Bob executes his code of class DefaultConsiderObject and here is where Bob decides if he will actually do as Alice requested (and dispenses with his private commitment 1 in the process). If he does decide to do it, he will choose agree as the appropriate subtype of reply to use. Otherwise, if he didn’t understand the message, he would choose not understood; if he isn’t inclined to do it, he will choose refuse6.

If Bob decides to agree, he will dispatch the following message to Alice:

```
( agree
  :act request|execute
  :sender Bob
  :receiver Alice
  :reply-with Bob--7
  :in-reply-to Alice--0
  :language "bash shell script"
  :content "ls" )
```

When any observer matches applicable policies, they will match the first two policies as for Alice’s original request ((MsgEvent * (*)), and (MsgEvent inform *)), plus a (MsgEvent agree (request *)) policy:

```
(policy (MsgEvent * (*))
  {quote {
    (Fulfil
      :Debtor msg.sender // = Bob
      :Creditor msg.receiver // = Alice
      :Performative msg.performative // = agree
      :Act msg.act // = (request execute)
      }})
```

When this policy fires, it will mark social commitment 2 as fulfilled.

```
(policy (MsgEvent inform *)
  {quote {
    (Add
      :Debtor msg.receiver // = Alice
      :Creditor msg.sender // = Bob
      :Performative (considerer) // = verify
      :Act msg.act // = (request execute)
      :ActionClass "casa...DefaultConsiderObject")})
```

When this policy fires, it will instantiate a private policy for Alice to verify Bob’s reply (ie: it is this the policy that will cause Alice’s code to run for Alice to consider the information about Bob’s response and decide what action, if any, to take).

```
(policy (MsgEvent agree (request *))
  {quote {
    (Add
      :Debtor msg.sender // = Bob
      :Creditor msg.receiver // = Alice
      :Performative perform
      :Act (cdr msg.act) // = execute
      :ActionClass "casa.policy...PerformAction"
      :Shared)
    (Add
      :DependsOn
      (SCStateEvent
       (SCdescriptor
        :Performative perform
        :Act (cdr msg.act))
       fullfilled )
      :Debtor msg.sender // = Bob
      :Creditor msg.receiver // = Alice
      :Performative propose
      :Act (cons discharge (cons perform
         (cdr msg.act)))
       // = (discharge perform execute)
      :ActionClass "casa...ProposeDischargeAction"
      :Shared))}
```

The policy will add two shared social commitments: one for Bob to actually perform the act (he has agreed to it), and one for Bob to propose the discharge of Alices request. Thus, there are now 3 unfulfilled commitments:

3. :Debtor Alice
   :Creditor Bob
   :Performative verify
   :Act (request execute)
   :ActionClass "casa...DefaultConsiderObject"
   :State ready

4. :Debtor Bob
   :Creditor Alice
   :Performative perform
   :Act execute
   :ActionClass "casa.policy...PerformAction"
   :Shared
   :State ready

5. :Debtor Bob
   :Creditor Alice
   :Performative propose
   :Act (perform execute)
   :ActionClass "casa...ProposeDischargeAction"
   :Shared
   :DependsOn 4
   :State ready

Thus, at this point, Alice can consider Bob’s answer, and (because he had agreed to Alice’s proposal) Bob has social commitments both to actually perform Alice’s request...
and to propose the discharge of performing Alice’s request. The conversation continues, but the example so far should suffice to illustrate the concept. On the other hand, if Bob had sent a refuse message instead of an agree message, the (MsgEvent agree (request *)) policy would not have matched, and commitments 4 and 5 would not have been instantiated. Commitment 3 would have still been instantiated from the (MsgEvent inform *) policy, but this is a private commitment for Alice to examine Bob’s reply and serves only to inform Alice that Bob had refused.

V. RELATED WORK

Jade [8] is an open source agent ‘middleware’ project run by Telecom Italia. Jade is FIPA compliant and is aimed at solid support for commercial deployment of agent based systems [9]. CASA and Jade share many of of the same objectives and philosophies, however they differ in several respects. Jade is aimed at a wide range user community that supports a single standard, while CASA is aimed at providing a tool for researchers and others who want to experiment with and work with a variety of communication protocols and paradigms. While Jade is strictly FIPA compliant, CASA can be FIPA-compliant, but allows the policy writers and agent programmers to differ from FIPA in whatever way they see fit. The policies largely dictate agent communication protocols and can be freely modified by the application developers. Jade provides several services, as dictated by the FIPA standard, such as white- and yellow-pages services, while CASA is currently not attempting to provide all FIPA services.

COUGAAR [10] is an interesting agent infrastructure designed for DARPA, primarily by BBN Technologies. It is another Java based platform that is heavily bound to Java, using RMI, Java persistence, etc. COUGAAR deals with sophisticated inter-agent services [11], such as naming, message transport, QoS, and alarm services that CASA doesn’t provide. COUGAR also provides a detailed intra-agent blackboard-based architecture [12] whereas CASA specifically aims to avoid prescribing an intra-agent architecture. While COUGAAR has a very sophisticated message transport system, message envelopes are quiet primitive and consist primarily of a sender and receiver label and a Java-defined persistent object as the the content. Therefore, COUGAAR lacks CASA’s support for non-specialized agents accessing sufficient information about messages to deal with the message and conversational semantics described here.

VI. DISCUSSION

In this paper, we use CASA to illustrate the power and flexibility of using policies to support social norms by guiding agents in their conversational interactions as well as their response to external events. Specifically, conversational moves and events are treated uniformly as messages (receiving, sending, and observing) are merely treated as specialized events (speech acts are a subtype of events). One of CASAs main, unique contributions is showing that policies can be usefully specified, not by being hard-coded, but by being specified in a distributable lisp-like policy specification language. While the policies encode “social norms”, they do not direct the decision-making power of the agent: agents are “consulted” at conversational decision points by calls into their code specified in the policies.

External policies, such as described here, significantly simplify agent development by allowing the system developers separation of concerns: to consider social norms (policies) separately from the application layer (the actual task of coding the behaviour of the the agents)7.

External policies as described here also have the advantage of being pure text, and easily exchanged between agents at runtime if necessary. In an advanced system, cooperating agents could exchange, “learn”, and analyse other agents’ policies at run time.

CASA is available for download at http://pages.cpsc.ucalgary.ca/~kremer/CASA/.

ACKNOWLEDGMENTS

The author wishes to acknowledge the Canadian National Science and Engineering Research Council (NSERC) for financial support of the research. The author also thanks Dr. Roberto Flores of Christopher Newport University for ideas and editorial advice.

REFERENCES


7This is not to say that policies can’t also be used to specify certain aspects of an agent’s specific behaviour – policies can be used for this purpose.