Policies, Conversations, and Conversation Composition

– Authors withheld; references to the implementation system are obfuscated –

Abstract. This paper motivates the use of inter-agent conversations in agent-based systems and proposes a general design for agent conversations. It also describes an instance of that design implemented in our experimental agent-based framework. While the agent community is generally familiar with the concept of a conversation, this paper's contribution is a particular model in which flexible composition of conversations from smaller conversations is possible.

Keywords: Agents, Conversations, Multi-Agent Systems, Communication Protocols

1 Introduction

The ability to communicate is a prevalent feature of systems where agents rely on coordination and division of labor. Communication is achieved by externalizing meaning through an artifact, either by changing the state of an existing object or by creating a new artifact whose state can be perceived by the intended audience. In the absence of commonly accessible artifacts, software agents are left with messages as the innate conduit to support communication and coordination.

Through the years it has become understood that messages are not just isolated tokens of data but that meaning and purpose is also attached to a sequence of messages exchanged between participants. These sequences, called conversations, have properties that pertain to sequencing (which places in the conversation when messages can take place) and turn taking (the set of participants that could send allowed messages to advance a conversation).

Conversations are normally defined by developers at design time and can be encoded programmatically or through rules executed at runtime. An advantage of rules over program instructions is they could entail flexible (malleable, learnable, and blended) conversations given existing agent state. A flexible conversation would be one that is able to handle unexpected messages (which could either be erroneously sent or sent with the intent to refine the context of interaction) by leading to commonly understood conversation states from which agents could recover and reconvene to an originally intended conversation state.

In this paper we present our approach to encode flexible conversations using rules, and in particular policies as a higher abstraction to rules, in which policies can exist in the context of a conversation template. To this end, Section 2 presents our conceptualization of an agent as a software program that holds conversation policies reacting to message events, has the ability to send and receive messages, and contains an event queue in which events are scheduled and acted upon. Sections 3 and 4 present conversations and subconversations, respectively, and Section 5 describes how they are implemented in our experimental agent-based framework. Lastly, Section 6 presents related work, and Section 7 our conclusions and future work.

2 Policies

While agents may be anywhere on the continuum between reactive to contemplative, we focus on the reactive part because all useful agents need to be reactive to some extent. A *purely* contemplative agent would never produce any output, so would have no useful purpose from our perspective. While we don't rule out contemplation, we model an agent as reacting to events, including, importantly, communicative events. Thus, we will assume an agent has an *event queue* where it will look when it's ready to process the next event (when it's done with the current events and any contemplating it's doing). Following many existing systems, we use an event queue rather than react immediately to incoming events because it is not always possible to react immediately to every incoming event. In order to specify how an agent is to process an event, we choose to use a rule-based system, where a set of rules will dictate one or more behaviours based on the event at hand. We call these rules *policies* because, as the reader will see, they differ from typical rules seen in the literature.

Since agents are computational entities, we assume all perceivable events come to it through an event queue. An agent observing an event (really, dequeueing it from its event queue) will attempt to apply policies to determine what to do. In this paper, we won't go into details on how the agent deals with multiple applicable policies since this issue is independent of policy structure and application, and can be usefully implemented in a variety of different ways. Most rule implementations have an antecedent consisting of a boolean expression; however since the event is the key element which actually stimulates policy application, we feature the matching of an *event descriptor* and an event as the antecedent of a policy. However, we also include a *precondition* (which *is* a boolean expression) which must evaluate to *true* in order for the policy to be applied. We use the precondition to filter on the bases of history, agent state, environmental state, or some combination thereof in the same way as more traditional rules do. Thus, we can compare the boolean antecedent of a tradition rule to $match(eventDescriptor, event) \wedge precondition$. This use is similar to rules in COOL [1].

Traditional rules also have consequents, and policies do too. We model the consequent as a list of *actions* which the agent will execute if a policy is fired. Note that we do not assume that the actions are primitive actions, but may include conditional actions, angelic choice among actions, repeated actions, parameterized actions, etc. Actions may include anything the agent can do, such as sending messages, changing state, etc. As well as at the consequent, we include in policies a *postcondition* which resembles the precondition in that it is a boolean expression describing some state of the agent or the environment, but it is interpreted as the *expected* state immediately after the policy is fired. The postcondition is useful for planning and analysis, but there is no guarantee the postcondition holds after a policy fires, as unanticipated outcomes (such as errors and failures) can always happen. The postcondition is similar to FIPA's *rational effect* [2].

Formally, a policy consists of an antecedent, precondition, postcondition, and consequent:

$$\begin{aligned} \mathit{Policy} \equiv (\, & \mathit{antecedent} : \mathit{Antecedent}, \\ & \mathit{precondition} : \mathit{Precondition}, \\ & \mathit{postcondition} : \mathit{Postcondition}, \\ & \mathit{consequent} : \mathit{Consequent}) \end{aligned}$$

The next subsections will describe these four components of a policy in detail. However, we first give some general definitions of properties:

$$\begin{aligned} \mathit{Property} &\equiv Identifier \mapsto \top \\ \mathit{Properties} &\equiv Identifier \nrightarrow \top \end{aligned}$$

That is, we use *Property* to describe a single attribute/value pair, and *Properties* to describe a (possibly empty) set of attribute/value pairs.

Antecedents An antecedent to a policy is just a description of an event that may occur. If the event description matches the event that just occurred, then the policy *may*[1] be eligible to be fired.

An event is an abstract occurrence that is observable by the agent. We take events to have abstract types arranged in a type lattice. For example, in some application, both *horseRace* and *race* could be events with a subsumption relation between them which we describe as *horseRace* $\prec$ *race*. Formally, the subsumption relation is partially ordered set (poset) which we will call the event ontology:

$$\mathit{EventOnt} \equiv (\mathit{id} : Identifier, \prec)$$

such that EvenOnt is a lattice. That is, an event ontology is some set of identifiers and a subsumption relation among these identifiers.

An individual event is slightly more than that, as an individual event might have properties specific to the individual. Thus, the type of an event is *Event*:

$$\mathit{Event} \equiv (\mathit{type}{:}\mathit{EventOnt},\ \mathit{props}{:}\mathit{Properties})$$

That is, we describe an event as a type identifier (from an event ontology) and a set of attribute/value pairs. Unlike in most object-oriented programming languages, we choose *not* to constrain particular types to hold specific properties, but let the concept of type and the concept of properties float independently of one another.

We want to fire policies when we match the antecedent of the policy with an event that has occurred. Thus, we need an object to describe an event. We

[1] The policy *will* be eligible to be fired if the precondition is also true (see §2).

call that object an event descriptor, and it differs only slightly from an event itself. The event descriptor looks like an event, but for each property, it holds not only a value, but a comparator operator to use in comparing the values of the property to that of an event. Thus, we describe *event descriptors* as type *EventDescriptor*:

$$\begin{aligned} \mathit{EventDescriptor} \equiv \; & \big(\mathit{type}\!:\!\mathit{EventOnt},\ \mathit{props}\!:\!\mathit{Identifier} \nrightarrow (\top,\ \mathit{op}\!:\!\mathit{Operator})\big) \\ & \text{where } \mathit{Operator} \equiv \;\; \top \times \top \end{aligned}$$

That is, an Event Descriptor looks just like an event, except that its properties map identifiers not only to values, but also to operators, which are comparator predicates describing how the values are to be compared to corresponding values in an event. In practice, useful operators we have encountered are subsumption (via the ontology), equality, the simple math inequalities, and regular expression comparators [5].

An antecedent of a policy in nothing more than an event descriptor:

$$\mathit{Antecedent} \equiv \mathit{EventDescriptor}$$

More complex antecedents are possible, such as disjuncts of event descriptors, but we have not yet observed the need in practice.

The policy passes the antecedent if the event descriptor *matches*. An event descriptor matches an event if its type subsumes the type of the event and each of its properties' values is a match with the corresponding property value of the event according to operator associated with that property:

$$\begin{aligned} \mathit{matches}(\mathit{event}, \mathit{desc}) \equiv \; & \mathit{event.type} \prec \mathit{desc.type} \wedge \\ & \forall\, t\!: \mathit{dom}\, \mathit{desc.props} \bullet \\ & \quad \mathit{desc.props}(t).\mathit{op}\big(\mathit{event.props}(t),\ \mathit{first}\, \mathit{desc.props}(t)\big) \end{aligned}$$

It may seem odd that an antecedent is no more than an event descriptor because agents typically take into account the environment or history when making decisions. We choose to separate the matching of the event from decisions about the environment or history by using *both* the antecedent and requiring the *precondition* to hold as well. The precondition is described in the next section.

Preconditions A *precondition* describes the state (of the agent or the environment) that must be satisfied in order for the policy to be eligible to fire.

$$\mathit{Precondition} \equiv \text{boolean expression}$$

We interpret *state* broadly (to include history), and it is therefore no more constrained than any arbitrary boolean expression.

Consequents A *consequent* is a list of actions that will be executed if the policy fires. We do not describe actions in detail in this paper. Actions are application dependent, although there obviously include speech acts which are used by almost all muliti-agent systems, but vary in detail between systems.

Actions may be more than atomic or primitive, but may include composition of actions including conditional actions, angelic choice between actions, repeated actions, parameterized actions, etc. All we can say about actions is that they are of type *Action*. A consequent then, is merely a list of actions:

$$Consequent \equiv seq\, Action$$

Postconditions A postcondition is a partial description of the *expected* state of the environment immediately after the policy is executed. The word *expected* here is important as there is no guarantee that the state will actually conform to that state: errors may have occurred, or some other unforeseen process may have influenced the state. However, the postcondition is useful for planning and analysis. Since agents are purely computation entities, a partial state can be described by an arbitrary boolean expression, and therefore the type of the postcondition is merely a boolean expression:

$$Postcondition \equiv \text{boolean expression}$$

Applicability of Policies When an event happens, an agent must determine which of its policies is *applicable*. In a simple rule-based system, this is accomplished by merely evaluating the antecedents of the rules, and all those who's antecedents evaluate to *true* are applicable. This situation is only slightly more complex in policies in that we evaluate the precondition *and* whether the antecedent matches the current event:

$$\begin{aligned} applicable(policy, event) \equiv\ & matches(policy.antecedent, event) \\ & \wedge policy.precondition \end{aligned}$$

As already explained, multiple policies may be applicable an event, and this decision is application dependent. However, our default approach is to fire all applicable policies, basing the firing order on a measure of specialization of the antecedent match. Space limitations preclude a detailed discussion on this topic.

3 Conversations

Naively, a designer could write a policy-based agent in which the agent has a set of policies that it applies to each event it sees. However, the designer would quickly see the agent becoming increasingly complex as the agent has to account for multiple concurrent conversations, erroneous messages, maliciously construed messages from other agents, and data associated with individual conversations. The problem is one of history-state, similar to the states in a state diagram. For example, a simple conversation protocol could be that buyer may *request* an item from a supplier at a certain price, to which the supplier could *agree* or $refuse$ (see Figure 1). Given this simple protocol, a devious supplier might unilaterally send an *agree* to a buyer to sell a certain item at a certain (high) price. A naive buyer could be tricked into purchasing the unwanted item. The problem

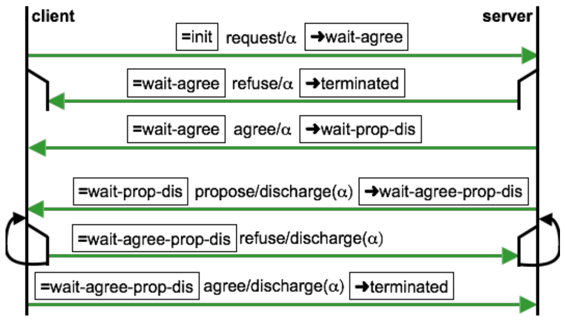

Fig. 1. A simple request conversation. Arrows are messages between agents. Messages are labeled with the performative of the message (e.g.: "request", followed by a slash and the content of the message (e.g.: "α"). Boxes with "=" represent preconditions of the state property of the policy handling the message. Boxes with "$\rightarrow$" represent setting the state property in the consequent of policies handling the message.

can be easily handled with state: in its *initial* state, the protocol would only entertain a *request*, after which it would only entertain an *agree* or a *refuse*. An out-of-the-blue *agree* would be handled as if the message was not understood (because, out of context, it isn't). The reader may have the impression this design is a finite state machine, however, conversation state is not *required*, merely a helpful concept that we have found convenient for conversational design. The problem stated here can also be solved by referring explicitly to past events, but modeling these with state tends to be easier for the conversation designer.

In addition to the simple history-based state described above, conversations usually have other state information associated with them such as "who is the debtor and and who is the creditor in the conversation?" Furthermore, an agent may carry on several conversations with several different agents concurrently. All this suggests that, although agents certainly need policies at the top level, conversations should be objects that contain conversation-specific policies and represent their own state (where that state might contain history).

With respect to events, policies, and conversations, we model an agent as containing an *event queue*, a set of *global policies*, a set of *conversation templates*, and a set of *current conversations*. When an agent dequeues an event from the event queue, it checks to see if it's a message event (sent, received, or just observed), and if it is, there should be a conversation id associated with it. The agent then checks to see if the conversation id matches the id of any of its current conversations. If it does, it checks to see if any of the conversation's policies are applicable, and if there are, it fires these policies. On the other hand, if either there is no matching conversation or the matching conversation has no applicable policies, the agent consults its global policies to determine a course of action. (If nothing matches, there is usually a catch-all policy which handles the error.)

Among the global policies are policies that will *create* a conversation. Obviously, in situations like the receipt of a *request* message, there is no current conversation, so there must be a global policy that recognizes a *request* message whose consequent includes creating a conversation to handle the request.

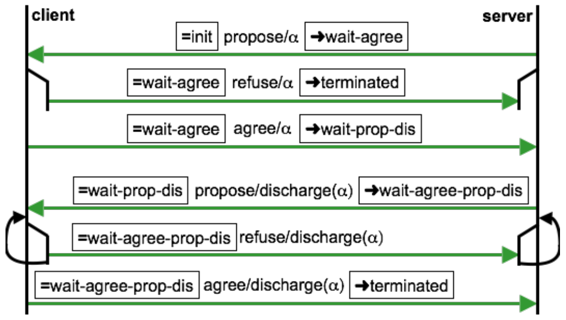

Fig. 2. A simple propose conversation.

Such policies create new conversations by cloning a new conversation from the conversation templates.

Under this model, a conversation is a set of policies, a state variable, and a set of properties:

$$\begin{aligned} SimpleConversation \equiv \big(\, & policies : (\mathbb{P}\, Policy), \\ & state : Identifier \mapsto Identifier \\ & props : Properties \,\big) \\ & \quad \text{s.t. } state \in props \wedge first\, state = \text{“}state'' \end{aligned}$$

Here, we have chosen to include state as just-another-property, however we distinguish it for reasons that will become apparent in §4.

Policies are no different from the global policies described in §2, except that they will be evaluated and fired within the context of the conversation. That is, the conversation object and its properties will be available to any code in the antecedent, precondition, consequent, or postcondition.

As already mentioned, the conversation state is modeled as just another conversation property who's value is a simple identifier. State is distinguished among the other properties only because it is so commonly used in conversation modeling. We use state to model the over-all state of the conversation, similar to the "current" state in a state diagram. While more complex models of state, such as those entailed in Petri nets, are possible, we choose to keep the state model simple.

Conversation properties are simple attribute/value pairs. Since we endeavor to keep our model as general as possible, we impose no restriction on properties.

4 Subconversations

Above, we have described atomic conversations as *SimpleConversation*. However, we want conversations to be compose-able; that is, to build up complex conversations from simpler conversations nested within. For example, if one looks at the request conversation (Figure 1) and the propose conversation (Figure 2), one can see that the second half of each conversation is identical. It would be better if we could write the discharge (second half) sub conversation once, and include

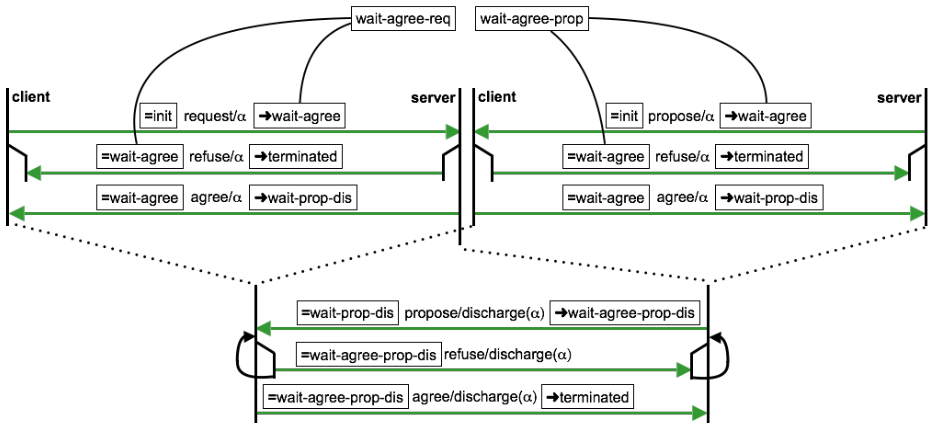

Fig. 3. A request/propose conversation.

it in both conversations. Even better, we could compose the first halves of both the request and propose conversations with the discharge sub conversation to create a request/propose conversation which represents a transaction that could be initiated by either the client or the server (see Figure 3).

Sub-conversations have many benefits analogous to composition and nesting of functions in programming languages. However, unlike programming language functions (which use control transfer and parameters to coordinate the components) sub-conversation coordination does not necessarily enjoy clean control transfer[2], and requires shared properties instead of parameters. We therefore *share* properties between super-conversations and their contained sub-conversations. However there is a problem here: The super- and sub-conversations may have been written independently, and their property names may not semantically align (i.e.: a super conversation's property may have a different name from a child conversation's property for the same semantic meaning). Therefore, we must include a way of *binding* property identifiers between super- and sub-conversations such that identifiers may be recognized as the same thing, but with different names. Binding two identifiers means that they share a memory location, and therefore bindings in super- and sub-conversations affect each other. These bindings will be defined as part of the super-conversation's definition (not the sub-conversation's) as developer of sub-conversations cannot be expected to know where their sub-conversations are going to be included.

A similar problem develops on a lower level with binding of the *value* of the state property: Sometimes a sub-conversations state value may hold a different semantics from the same state value of the super-conversation. For example, frequently the *terminated* state of a sub-conversation merely signals to the super-conversation a new (non-terminal) state.

Thus, we can define a full conversation as an extensions of the previously-defined simple conversation (additions bolded):

[2] Sometimes more than one sub-conversation will be operating concurrently.

$$\begin{aligned}
\mathit{Conversation} \equiv \big(\, & \mathit{policies}\!:\!\mathbb{P}\,\mathit{Policy},\\
& \mathit{state}\!:\!\mathit{Property},\\
& \mathit{props}\!:\!\mathit{Properties},\\
& \boldsymbol{children}\!:\!\mathbb{P}\,\boldsymbol{Conversation},\\
& \boldsymbol{binds}\!:\!\mathbb{P}\,\boldsymbol{Binding},\\
& \boldsymbol{bindState}\!:\!\mathbb{P}\,\boldsymbol{StateBinding}\big)
\end{aligned}$$

Property bindings bind a property identifier in the super conversation to a distinct property identifier in a particular sub-conversation:

$$\begin{aligned}
\mathit{Binding} \equiv (\, & \mathit{symbol}\!:\!\mathit{dom\,props},\\
& \mathit{child}\!:\!\mathit{children},\\
& \mathit{childSymbol}\!:\!\mathit{dom\,child.props})
\end{aligned}$$

If any particular property identifier appears in both a sub-conversation and its super conversation but is otherwise unbound, the two are assumed to be bound.

State bindings bind *value* identifiers of the sub-conversation's state property to different state value identifiers of the super-conversation:

$$\mathit{StateBinding} \equiv (\mathit{val}\!:\!\top, \mathit{child}\!:\!\mathit{children}, \mathit{childVal}\!:\!\top)$$

If identical state values appear in both a sub-conversation and its super-conversation, they area assumed bound. If a state value is assigned to the shared state property that doesn't appear in the sub-conversation, and is not bound to state value in the sub-conversation, then it cannot match any state expression in the sub-conversation.

The need for state value bindings is illustrated in the example in Figure 3. Note that the request sub-conversation and propose sub-conversation both change the state to *wait-agree* upon processing a request message or a propose message respectively. In this case, without state value bindings, the conversation would allow either the client or the server to send an agree or refuse message. This is not what we want. By binding the request sub-conversation's *wait-agree* state value to a different identifier in the super-conversation's state value than the propose sub-conversation's *wait-agee* state value, the two identifiers become distinct and the receipt of a request message allows only the server to respond with an agree message, and, respectively the receipt of a propose message allows only the client to respond with an agree message.

5 Implementation

Policies and conversations have been implemented as described here in out experimental agent based framework [5]. Our system is written in Java and uses Common Lisp as a run-time scripting language. Our system's main goal is to provide flexibility in agent development by using plug-in and scriptable components to implement all major agent functionality. For example, policies and conversation protocols are never hard-coded, but are read in at agent startup from script files. Since the scripting language is Lisp, and Lisp is a full programming language on its own, these scripts can be as simple or as complex as desired. For example, a definition of a policy to ignore outgoing messages with a performative of "not-understood" can be written as follows:

```
(policy
  `(msgevent-descriptor event_messageSent :=performative not-understood)
  `(nil))
```

Here, `policy` is a lisp function call which defines a policy object. It has two required parameters: an antecedent which is an *EventDescrptor* object (see §2) and a consequent which is executable code (a list of actions). Both these parameters are quoted (single backquote before the open parenthesis) so that they may be executed and instantiated in the context of the event at runtime, and not in the context of agent load time.

The conversation of Figure 3 can be defined as follows:

```
(conversation "request-propose-conversation"
  (list
    ask-client       ; previously defined
    offer-client     ; sub-conversations
    discharge-client )
  :bind-state `(
    ("wait-agree-req" ask-client "wait-agree")
    ("wait-discharge" ask-client "terminated-pending")
    ("wait-agree-prop" offer-client "wait-agree")
    ("wait-discharge" offer-client "terminated-pending")
    ("init" discharge-client "blocked-init")
    ("wait-request" discharge-client "blocked-request")
    ("wait-propose" discharge-client "blocked-propose")
    ("wait-discharge" discharge-client "init")
    ("wait-propose-discharge-reply" discharge-client "wait-propose") )
  :bind-var '(
   ("server"
     (if (agent.isa (event.get-msg 'performative) request)
       (new-url (event.get-msg 'receiver))
       (new-url (event.get-msg 'sender))))
   ("client"
     (if (agent.isa (event.get-msg 'performative) request)
       (new-url (event.get-msg 'sender))
       (new-url (event.get-msg 'receiver))))))
```

The `conversation` function constructs a conversation object. The first required parameter names the conversation. The second required parameter is a mixed list of sub-conversations and policies. In this case there are no top-level policies – they are all contained in the sub-conversations. The key parameters shown are as follows:

`:bind-state` is a list of triples of super-conversation state property values, sub-conversation names, and sub-conversation state property values that describe its binding between values as per §4.

`:bind-var` is a list of pairs of property identifiers and their values. These values illustrate how run-time Lisp code can be used to dynamically set a value: in this case the value of the client and server properties are dependent on the performative of the initial message in the conversation. The super- to sub-conversation bindings of property identifiers (§4) does not occur in this example.

In practice, our framework's conversations typically take advantage of Lisp's `defun` (function definition) to define conversations at the speech-act [7] level. `defun` parameters can be used to flexibly customize conversations to the actual conversation template level. Thus, one can define a basic *request* conversation at the speech act level, but customize it to data requests, specific service requests, etc that may follow different processes at the action level.

6 Relation to Other Work

The current work is most closely related to COOL [1] which Barbuceanu and Fox consider addressing the "conventions" (coordination) level of agent interaction. Their model includes a numerically-labeled *state* in a finite state machine (analogous to our state property), messages with *performatives* (almost identical to ours), *conversation rules* (analogous to our conversation policies), *error-recovery rules* (the concept is captured within our global policies), *continuation rules* (analogous to our global policies), *conversation classes* (analogous to our conversation templates), and *actual conversations* (analogous to our instantiated conversations). Furthermore, COOL's rules are similar to our policies in that they have a *received* field (analogous to our antecedent), a *such-that* field (analogous to our precondition), and collection of fields such as *next-state* and *transmit* describing actions (analogous to our consequent). COOL even uses conversation-scope variables similar to ours. What COOL does not have, however, is any mechanism to handle composition of conversations as described in §4. Of less import, we also consider our work to have greater flexibility in terms of actions: COOL handles a limited set of possible actions as part of its definition, whereas our work leaves actions open to the full expressibility of the scripting language (Lisp, in the case of our implementation).

The JADE project [8] is a very well known agent implementation project. Jade agents interact using messages as described here, but differs dramatically from the current work when it comes to policies and conversations. JADE is strictly based on the FIPA model [3], and implements FIPA's protocols directly as Java classes [6]. If one needs to diverge from the limited set of interactions offered by FIPA, then one must program the interactions in Java from scratch (or subclass one of the existing classes if that is feasible). It is the objective of the current research to minimize the work of implementing new or variant protocols. The current work focuses on the protocol (conversation) level rather than on the implementation level. JADE focuses on implementation using a pre-defined protocol level (FIPA).

Yet another big MAS project is Cougaar from US Defence Advanced Research Projects Agency (DARPA). Given its source, it's not surprising Cougaar has a main goal of a security and robustness. Cougaar's inter-agent communication is not based on any standard, but includes a choice of several build-in protocols [4]. Cougaar offers no particular support for design of new protocols or conversations.

7 Conclusions

This work extends previous work in explaining one way to specify inter-agent conversations. In particular, these conversations are *composable*: complex conversations can be built up from simpler conversations. Conversation composition is complicated by synchronization among the conversations, which cannot necessarily be handled by control transfer as is done in programming languages. Instead, our approach is to share state information. However, state variable names from different conversations may not always correspond, so our solution is to provide a mapping from the outer conversation namespace to the contained conversation's namespace. It turns out that a mapping between values (as opposed to variable names) is also sometimes required. One example is that of the state variable `state`, where the `terminated` value for the `state` variable in a sub-conversation signals the sub-conversation's termination. But in the larger context of the super-conversation this `terminated` value is just an intermediate step where the sub-conversation has completed but the over-all conversation continues.

References

1. Barbuceanu, M., Fox, M.S.: Cool: A language for describing coordination in multi agent systems. In: Proceedings of the First International Conference on Multi-Agent Systems. pp. 17–24. AAA Press/The MIT Press (June 1995)
2. Foundation for Intelligent Physical Agents (FIPA): FIPA communicative act library specification. document number SC00037J, FIPA TC communication. (Dec 2003), http://www.fipa.org/specs/fipa00037/SC00037J.html
3. Foundation for Intelligent Physical Agents (FIPA): Fipa web site. http://www.fipa.org (August 2012), http://www.fipa.org
4. Helsinger, A., Thome, M., Wright, T.: Cougaar: a scalable, distributed multi-agent architecture. In: Systems, Man and Cybernetics, 2004 IEEE International Conference on. vol. 2, pp. 1910–1917 vol.2 (Oct 2004)
5. Reference removed for double blind refereeing.
6. Nikraz, M., Caire, G., Bahri, P.: A methodology for the analysis and design of multi-agent systems using jade. International Journal of Computer SystemsScience and Engineering 21(2) (2006), (earlier version) http://jade.tilab.com/doc/tutorials/JADE_methodology_website_version.pdf
7. Searle, J.: Speech Acts. Cambridge University Press (1969)
8. Telecom Italia Lab: Jade (java agent development environment). http://jade.cselt.it/ (May 2008)