Dynamic conversation structures

Scott A. Moore
University of Michigan Business School
samoore@umich.edu

Abstract

Models of communication that emphasize convention and those that emphasize inference each have their uses as a theoretical basis for a system of formal communication. Systems based on the former are less computationally intensive at the expense of being more complex to maintain and modify. Those based on the latter result in the opposite. Certainly, what is desired is a system that combines the best of both types. This paper explores the issues involved in building such a system.

1 Introduction

Recent research in agent communication languages (ACLs) has focused on the semantics of messages (FIPA's ACL (Foundation for Intelligent Physical Agents 1997), FLBC (Moore 1998b), KQML (Labrou & Finin 1997)). Much of this research has focused on languages whose semantics are expressed in terms of the beliefs, desires, and intentions (BDIs) of the participants in the exchange—or at least the computerized simulation of such states. Neither researchers nor the marketplace have come to any kind of agreement about what messages should mean nor how they should be interpreted. The reasons for this failure are deep and will not be going away soon; for example, one basis for this disagreement is the diversity of agent models. Agents are built for different purposes and need correspondingly simpler or more complex agent models.

In this paper I present a means of modeling conversations composed of messages in an ACL. This is an area fraught with trade-offs because of competing criteria. On the one hand, companies want to have models that are not complex to interpret (to ease maintenance) or to execute (to increase response time and decrease technology investment). On the other hand, they also want to have flexible conversation models that can both handle many alternative responses and integrate unexpected responses. If it were possible to implement such a system, it would benefit those who use and deploy agents in several ways. First, the ability to share models would make it easier to get started. Organizations could build on the expertise and experiences of others instead of having to start from scratch when building communicating agents. Second, focusing on and explicitly specifying how messages are being used to *do* things would make it clearer than current practice allows how a message should be interpreted and responded to. Finally, changing from an inference-based interpretation system—such as that used by FLBC and FIPA's ACL—to one based more directly on convention should simplify and speed up the interpretation of frequently used messages. However, care will have to be taken to keep the system flexible so that it can meaningfully respond to messages from a wide variety of partners. This is something that, I propose, KQML has not been able to do.

In this paper I define what I mean by "conversation policy." I then develop a solution to a general, but relatively simple, communication problem. Through this analysis I progressively reveal an approach to implementing conversation policies. Next, through

an extended example I demonstrate how this system of managing message exchange by conversation policy (a method that relies heavily on convention) contrasts with a system based on inference. I also show how these two can be integrated to gain some of the advantages of both. I conclude by discussing the benefits and drawbacks of the proposal.

2 Conversation policies

In this section I present my views on conversation policies. As a basis for understanding that presentation, I first discuss my assumptions regarding agents, ACLs, and conversations.

Researchers have deployed many types of agents. These agents have been built to accomplish many different tasks and have widely differing capabilities. This situation shows no signs of stabilizing. One of the main differences relates to mobility. As discussed by Labrou et al. (Labrou, Finin, & Peng 1999), the current situation is so bad that researchers of mobile and (non-mobile) agents even mean different things when referring to agent interoperability. For mobile agents it means being able to make remote procedure calls on its current host. For (non-mobile) agents it generally means being able to exchange messages in some ACL. Other differences, while less dramatic, offer similarly difficult obstacles to overcome if agents of all types are to ever to cooperate or interoperate. The point of this work is to describe a means for allowing—in the sense used by the (non-mobile) agent community—agents, mobile or not, to communicate with each other so that one can get information from another or get the other agent to do something for it.

In this paper I assume as little as possible about agent architecture and communication language so as to increase the chances that the framework might be widely applied. Later in the paper I narrow the focus a bit and show the benefits that accrue when this approach is applied to the FLBC. An underlying assumption to minimizing assumptions is that if a fully formal specification for both responding to messages and handling conversations in a heterogeneous environment were specified, it would be difficult, if not impossible, to completely verify the conformity of any random agent to that specification. One important impediment would be the political reality of the verification process: Does a third party handle the verification? Do they have to look at the code that makes up the agent? A second impediment would be to determine what is meant by verification. Would it mean that an agent is able to reply to a message in that ACL or all messages in the ACL? Would it mean that the agent is able to generate a response to a message no matter how simplistic that message might be (e.g., replying "yes" to all messages)? Another impediment is determining what the specification looks like. Would it be a static document? Would it change every time a new type of message were defined, every time a new message were defined, or not at all? To summarize, I assume that if all agents were able to conform, then that would imply that either 1) the specification was so simple as to be of limited benefit to those who are deploying systems, or 2) the specification was so complex and had so many exceptions that it could not be usefully, easily, or meaningfully deployed.

What I am going to assume is that developers can understand a formal specification and can implement it *in whatever way they see fit* given the needs of their application. For example, in FLBC the standard effects of a **request** message are that 1) the hearer believes that the speaker wants the hearer to do something, and 2) the hearer believes that the speaker wants the hearer to want to do that same something (Moore 1998b, Appendix B, D). Some agents might implement these effects within a defeasible belief system that models its own and others' beliefs. Other agents might not have such sophisticated technology, might not model the beliefs of speakers, and might focus on the second standard effect and simply do what it is that the requesting agent asked. I assume that agents can have any capabilities (e.g., use a belief system, have deontic reasoning) but they have to be able to implement procedures that can follow the conversation policy specifications. I base the examples in this paper on a specific language but the principles concerning conversation policies should be applicable to a wide range of languages.

In order to communicate with other agents, some

particular ACL must be used. For the purposes of discussing conversation policies in this paper, the specifics of any one language are mostly irrelevant. It should be possible to add these facilities for describing and managing conversations to almost any ACL, though the benefits of such an addition may vary from one language to the next. That said, in this paper I refer periodically to FLBC (see (Moore 1993; 1998a; 1998b), among others; more information can be found at the Web site `http://www-personal.umich.edu/~samoore/research/flbc/`). FLBC is a language similar to KQML. Its message types are based on speech act theory; in contrast with KQML, this set of types has not needed frequent additions.

An FLBC message is an XML document (Cover 1998; Light 1997) and has deep foundations in speech act theory. The `flbcMsg` XML DTD is located at `http://www-personal.umich.edu/~samoore/research/flbcMsg.dtd`. An FLBC message has the form

```
<?xml version="1.0" encoding="UTF-8"
  standalone="no"?>
<!DOCTYPE flbcMsg SYSTEM
  "http://www-personal.umich.edu/~samoore/-
     research/flbcMsg.dtd">
<flbcMsg msgID="M">
  <simpleAct speaker="A"  hearer="B">
    <illocAct force="F"  vocab="N"
              language="L">
      X
    </illocAct>
   </simpleAct>
  <context>
   <resources>
    <actors>
     <person id="A"/><person id="B"/>
    </actors>
   </resources>
   <respondingTo message="V"/>
  </context>
</flbcMsg>
```

The interpretation of this message is fairly straightforward. This is a message, identified by `M`, from `A` to `B` in reply to a previous message identified by `V`. Speech act theory hypothesizes that all utterances have the form `F(P)` where `F` is the *illocutionary force* of the message (e.g., `inform`, `request`, `predict`) and `P` is the propositional content of the message (what is being informed, requested, or predicted). In conformance with the theory, the structure of this and all FLBC messages is `F(X)`. The content `X` is written using the language `L` and the ontology `N`.

When using FLBC, a message's sender constructs and sends a message knowing that he cannot know, but can only predict, how it will be interpreted. The message's recipient receives the message, interprets it, and then uses it as a basis for inferring how he should respond. The "standard effects" of a message are those specific to the message type; for example, the standard effects of all `request` messages are those defined above. The "extended effects" are those effects that depend on the message's content (the *propositional content* in speech act terms) and context. Part of the context is the conversation to which the message belongs. This separation increases the probability that an agent will be able to usefully interpret a message it has not received before since 1) each agent developer knows the definition of the standard effects of each message type, 2) the standard effects contain a significant portion of the meaning of any particular message, and 3) the set of currently-defined message types covers a significant portion of the messages that agents need to say.

A conversation among agents is an exchange of messages. This exchange is generally done toward the accomplishment of some task or the achievement of some goal. In its simplest form it is a sequence of messages in which, after the initial message, each is a direct response to the previous one. More complicated structures occur when subdialogs are needed. Linguistics and philosophers have not come to any consensus about subdialogs, but several types consistently appear (Litman & Allen 1987; Moore 1998b; Polanyi 1988): subordinations, corrections, and interruptions. A message begins a *subordinate* conversation when it elaborates on a point made in a previous message. This message should be about the previous message, probably as a clarification of some fact. A message that begins a *correction* subdialog indicates that the message somehow corrects a previous message in the conversation. A message *interrupts* the current conversation when it is neither an expected nor the standard reply to the previous message. This

should be used only if the other two do not apply.

A conversation policy (CP) defines 1) how one or more conversation partners respond to messages they receive, 2) what messages one partner expects in response to a message it sends, and 3) the rules for choosing among competing courses of action. These policies can be used both to describe how a partner will respond to a message (or series of messages) it receives and to specify the procedure the partner actually executes in response to that message (or those messages). The policy is a template describing an agent's reactions to an incoming message, its expectations about upcoming messages, and the rules it applies to determine its own reactions. In a linguistic sense, it moves the conversation from an inference-based process to a convention-based one. Inference acts now as a backup, providing a more computationally expensive way of understanding unfamiliar messages for which CPs have not been defined. The means by which this backup can be implemented is demonstrated in §4.

A CP is said to be *well-formed* if it does not contain contradictory directions for what a partner should do. These rules may assume any underlying agent model but, clearly, cannot assume more capabilities than the associated agent actually has. A specific CP is invoked upon the receipt of a message and specifies a series of actions that usually, but not always, includes sending messages related to the received message. Two dimensions of this approach differ a bit when a CP specifies actions for all the partners in a conversation (instead of simply one of them), specifically well-formedness and differing agent models. In addition to the requirement above, a well-formed multi-agent CP specifies a coherent set of actions that does not lead the conversation to a *deadlock*—a situation in which each agent is waiting for the other. As for agent models, just as no specific one is required for a single-agent CP, a multi-agent CP does not require that all agents share the same model; thus, the rules for picking alternate courses of actions for one agent might be based on that agent's beliefs, another agent might have rules that automatically reply to all incoming messages (the degenerate case), and another might look at the status of environmental variables (such as indications of whether a machine is on).

The ACL actually used should not much matter; however, there is a specific, and to the point of this paper, quite relevant way in which FLBC and KQML differ—how each treats a response to a message. (This difference applies to the version described in (DARPA 1993) but not in Labrou (Labrou 1996).) As I have previously argued (Moore 1998a, p. 216), information about discourse structure should not define a speech act. If we were to go down this path, agents might need, in addition to `reply`, acts for `reply-request` (if a message is replying to a question but is asking for more information) and `reply-request-reply` (if a message is replying to a question that was asked in response to a question) and a `reply-predict` (if a message is a prediction replying to a question) and so on. These are not what we want. I propose that discourse information, as much as possible, be represented in a message's context term, separate from the speech act (illocutionary force and content) itself.

3 Proposal

In the following I describe my proposal for how agents should handle and how ACLs should represent conversation policies.

Agents should work with conversation policies that are represented using the statechart formalism defined by Harel (Harel 1987). This is a graphical language which developers would find easier to work with than the underlying XML representation which the agents themselves use. Harel states, more clearly than I could, why statecharts were developed and why I think they are appropriate:

> "A reactive system... is characterized by being, to a large extent, event-driven, continuously having to react to external and internal stimuli. ... The problem [with specifying and designing large and complex reactive systems] is rooted in the difficulty of describing reactive behavior in ways that are clear and realistic, and at the same time formal and rigorous, sufficiently so to be amenable to detailed computerized simula-

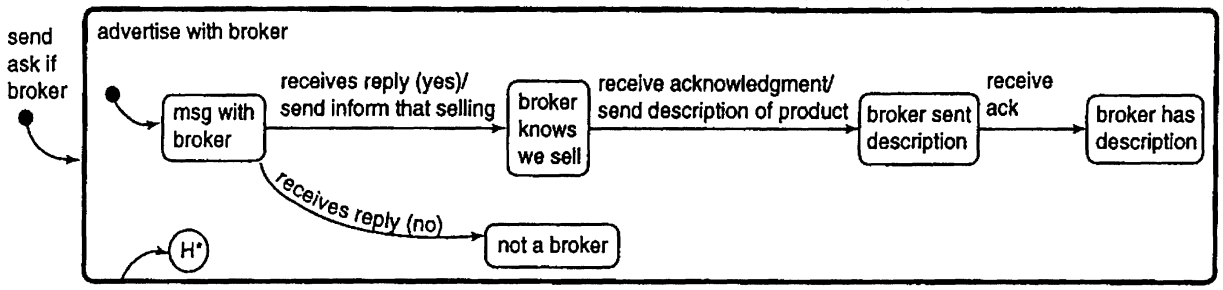

Figure 1: A sample statechart representation of a conversation policy

> tion. The behavior of a reactive system is really the set of allowed sequences of input and output events, conditions, and actions, perhaps with some additional information such as timing constraints." (Harel 1987, pp. 231–2). "Statecharts constitute a *visual formalism* for describing states and transitions in a modular fashion, enabling clustering, orthogonality (i.e., concurrency) and refinement, and encouraging 'zoom' capabilities for moving easily back and forth between levels of abstraction." (Harel 1987, p. 233)

Everything that Harel states about reactive systems applies to agent communication systems; thus, statecharts are an appropriate technology to investigate for modeling CPs.

Every CP begins with a message being either sent or received. An agent maintains a database of both CPs it can use and a set of currently executing CPs. Figure 1 contains a sample statechart representation of a conversation policy. The whole statechart is labeled advertise with broker. It has six states: advertise with broker, msg with broker, broker knows we sell, broker sent description, not a broker, and broker has description. Transition labels have the form t(c)/g, each part of which is optional. When the event represented within the trigger t occurs, then, if the condition c is met, the transition between states is made. If the g is present on the transition, then the event g is generated as the transition is taken.

To interpret the statechart in Figure 1, start at the left, think of the broker as some external agent, and think of these figures as representing our agent's CPs. When the send ask if broker event is generated somewhere within the system (that is, when a message asking if someone is a broker is sent), then this statechart is invoked. No condition (c) or generated event (/g) is on this transition. The transition is taken and this statechart is invoked because that event is the trigger for this statechart. The system's decision rules will favor specificity so it will choose the CP whose trigger provides the most specific match with the event that occurred. Specificity of triggers grouped by the type of the message provides one way of hierarchically organizing CPs. The unlabeled arrow with a dot on the end signifies that the system should start in the msg with broker state by default. The system stays in this state until a reply to this message is received. If the reply is "no", then the system takes the transition to the not a broker state. If the reply is "yes", then the transition to broker knows we sell is taken. When this is taken, the send inform that selling event is generated (that is, the system sends a message informing the broker that we sell some particular product). The system stays in this state until it receives acknowledgment. As it takes the transition to the broker sent description state, it generates the event send description of product. The system stays in this state until it receives an acknowledgment of this message. At that point it takes the transition to the broker has description state.

In the figures and body of this paper I informally represent the triggers, conditions, and generated events. I think of this as being the business

Event	Occurs when
entered(S)	State S is entered
exited(S)	State S is exited
true(C)	The value of condition C is set to true
false(C)	The value of condition C is set to false

Table 1: A sample of the operators defined by The Open Group

analyst's view of the statecharts. This could be universally shared. Again, an XML DTD for statecharts could define the interchange vocabulary for describing statecharts that agents would be able to send, receive, and process. I envision that the tool supporting the definition of these statecharts would also contain more formal (even executable) specifications more appropriate for the system designer's needs. These formal descriptions could provide a more detailed and clear definition of the statecharts for all who see them; however, I do not believe that it is realistic to expect that this formal specification would be universally executable. The most that might be expected is to have a certain core set of operators (see those defined by The Open Group (The Open Group 1999) and Table 1) that are shared by all and that have a shared semantic meaning.

I expect that CPs will be stored in a public repository that people can browse. The process of publishing a CP would assign it a unique identifier. After finding an appropriate CP in such a repository, the person could download it and subsequently adapt it or its implementation (depending on the programming language the agent is written in, etc.) so that his or her agent could use it. The more modular it is, the smaller it is, and the more it uses just the core statechart operators, then the simpler this adaptation process will be.

As for using this CP, when the agent is following a publicly-available CP, it would include the CP's name in any related message's context. This would have varying effects on the message recipient. First, if the recipient is not aware of that CP and does not want to change, then this information would simply be ignored. Because of the default standard effects in FLBC, an agent can usefully reply to many messages without being aware of the structure of the sending agent's CP. Second, the recipient may not have this CP implemented but may have noted that it has received many messages that use this CP. This could lead the agent designer to add this CP—or, more appropriately, a CP that could be used to interact with or respond to messages under that CP—to the agent's set of CPs. Third, if the recipient is aware of the CP, it may have already implemented a CP that it uses to interact with messages from that CP. If so, the receiving agent could simply invoke the appropriate CP, knowing that this is the most efficient and effective way both to handle the incoming message and to respond to the reason or goal underlying that message.

The statechart in Figure 1 describes a process in which an agent confirms that another agent is a broker, tells the broker that it sells a product, and then describes that product. This process could be represented by different statecharts—for example, the statecharts in Figure 2. This pair of statecharts is equivalent to Figure 1 except that two separate statecharts are composed to accomplish what, previously, one statechart accomplished. In the top statechart in Figure 2, when the system generates the send inform that selling event, the inform broker about product we sell state is entered. The bottom statechart in this figure has this same name and is invoked with this same event. Benefits of the representation in Figure 2 are that 1) the function of the advertise with broker statechart is more apparent because of the new, descriptive state, and 2) the inform broker about product we sell statechart can be used separately from the advertise with broker statechart. This modularization should be applicable to a wide variety of procedures.

An alternate representation of the bottom statechart in Figure 2 is the statechart shown in Figure 3. Though it is close, it is *not* equivalent to the original representation. The system no longer waits for the receipt of the acknowledgment message before transitioning to the broker sent description state. It also eliminated the need for the broker has description state

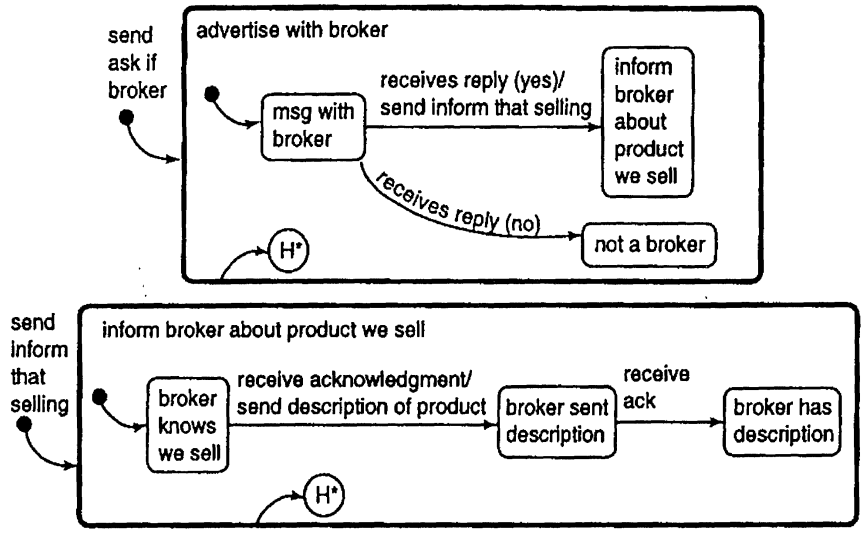

Figure 2: A sample statechart representation of a conversation policy using composition (this is equivalent to Figure 1)

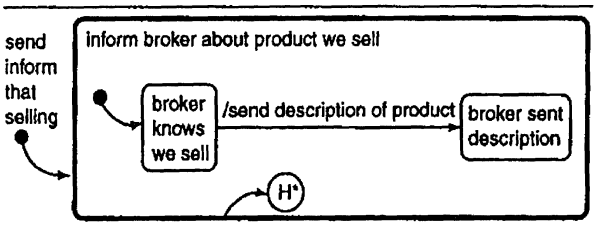

Figure 3: A statechart representation of a conversation policy that leaves a few messages out compared with the bottom statechart in Figure 2

by dropping the last receive ack trigger.

This raises one of the general difficulties with specifying conversation policies that are not shared by all conversants: each conversation partner may handle the same message in different ways. In this instance, the bottom CP in Figure 2 assumes that the partner returns an acknowledgment. If the partner does not return this message, then the conversation never progresses without some sort of external intervention (e.g., sending an email to the partner and asking him or her to send an acknowledgment—clearly not the solution we're looking for). The alternate representation shown in Figure 3 removes the requirement that the partner send acknowledgment messages but also no longer specifies how these messages would fit into the conversation were they actually sent. This is where the proposed method really shines. The following shows how this would work.

To this point I have ignored the process of how a statechart used to describe a CP becomes a statechart used both to define a set of executable activities and to monitor and control their execution. I describe that process here. Suppose both that the agent has just sent a message to another agent asking if that agent is a broker for a certain kind of product (i.e., send inform that selling) and that our agent uses the CP depicted in Figure 3. When the system determines that it is going to use this statechart, it puts it inside a state that has a unique (random and meaningless) conversation identifier. The original statechart (in Figure 3) is a normative specification for how the conversation should proceed. This new outer state (in Figure 4) is used to individuate this conversation among the set of all other conversations (not shown) this agent is currently (or has been) involved in. This allows the agent to carry on simultaneously multiple conversations that use the same CP.

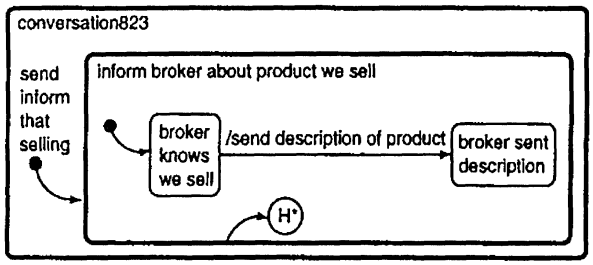

Figure 4: A statechart representation of a current conversation

Now, suppose that the broker agent informs our agent (via an inform message) that it received our message that we sent informing it that we sell a product (i.e., send inform that selling). The message our agent receives contains a term that indicates the message is a reply to the original inform message our agent sent. This term tells our system that the reply should be part of this conversation (conversation823 in Figure 4); thus, the CP should specify how the system should respond to it. Given all this, the figure clearly *does not* specify how to handle this message since there is no trigger receive inform in the figure. However, there is more to the system than has been described and there actually *is* a defined procedure for responding to this message. It simply is not shown. To that we now turn. Recall that the agent has a database of CPs. For FLBC the database contains a CP for the standard effects of each message type; these CPs are invoked only when other, more specific CPs do not apply. Thus, for every incoming message, a procedure can unambiguously choose one appropriate statechart.

Suppose our agent has just received this message—informing it that the broker agent received the original message—that the CP does not specify how to handle. Further, suppose that our agent has the statechart depicted in Figure 3 in its database of normative CPs. The system follows a procedure such as the following:

1. It looks for an appropriate CP. In this case it chooses standard effects for inform. See the bottom portion of the statechart in Figure 5.

2. The system draws a transition from the state im-

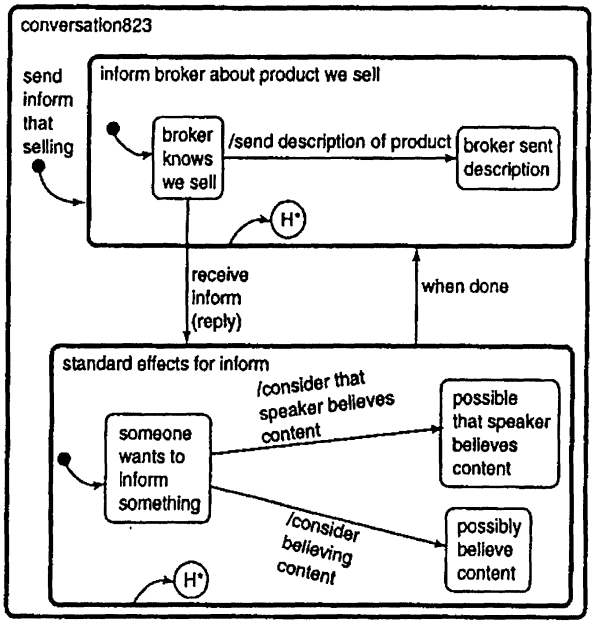

Figure 5: A statechart representation of a conversation policy with an unplanned-for subdialog

mediately following the transition containing the message that the incoming message says it is a reply to. In this case, the message is a reply to the send inform that selling message on the left of the inform broker about product we sell state. The state immediately following this one is the broker knows we sell state. The transition is drawn from this state to the statechart representing the CP identified in the previous step (standard effects for inform). This transition should be labeled with a description of the reply. This label is descriptive—so as to show what caused the system to take this transition—and restrictive—so as to keep the system from taking it again after the system returns to the inform broker about product we sell state (described below). In Figure 5 this is represented by the transition labeled receive inform (reply).

3. The system now draws a transition from the CP chosen in the first step. The transition is drawn from this state back to the original CP. This

transition is labeled when done to indicate that this transition should be taken when the standard effects for inform state is completed. Not shown is that each statechart contains a definition of the conditions that must hold for it to be completed.

The new statechart is now joined to the old statechart within the context of this conversation. In terms of the statechart, this means that the system has finished joining the original statechart to the statechart for the standard effects; this is all shown in Figure 5. Now that they are joined, the system can execute the appropriate steps: The send inform that selling message was sent (before these two statecharts were joined) and conversation823 was begun. The inform broker about product we sell state was entered. The default state broker knows we sell was entered. Because it was entered, the transition to its immediate right is taken and the event send description of product is generated. This takes it to the broker sent description state. At some later time (or even concurrently with the last transition) the system receives the reply to the first message, joins the statecharts as described above, and enters the standard effects for inform state. The default state is entered which immediately causes the system to take both transitions and generate their associated events. The system then determines that the statechart standard effects for inform is done and takes the when done transition back to the original statechart.

This is where the H* (history) node is useful. This signals that the statechart should restart in the node it last finished. In this case, since the transition was immediately taken into the last state, this statechart has already been completed so the system does not reenter it. If the transition had not been taken because a trigger had not been fired or because a condition had not been met, then the system would go back into the broker knows we sell state.

Figure 5 is possibly incomplete in two ways. First, later messages might add further subconversations. At some later time the broker might send something back in response to the send description of product message. Second, the CP does not plan for all contingencies. For example, if the agent had been informed that the broker does not sell the product, then a message (the send description of product message) would be erroneously sent. This might happen if the broker receiving our inquiries employed the CP depicted in Figure 6. The inform broker about product we sell CP, as it is currently constructed, essentially assumes that the broker is not going to say anything that would convince the agent to not send the product description. If this were a common occurrence and if the sending of the product description were expensive, the CP inform broker about product we sell should certainly be refined.

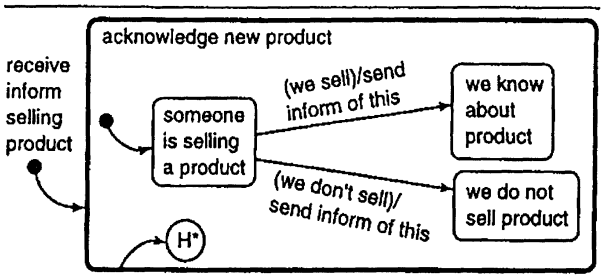

Figure 6: A statechart that describes the process of acknowledging a new product

The above discussion has mostly applied to the actions, messages, decisions, and conversation policies for one participant in a conversation. Using FLBC as a communication language, this agent cannot know what the other does with this message. The sending agent can only know that the receiving agent knows what the sender wants the hearer to consider believing as a result of the message. Consider the message send inform that selling that the agent sends to the broker. The broker agent—not our agent, but the broker agent—may have a CP defined like the one shown in Figure 6 or it may simply use the standard effects for inform. The sending agent simply does not know which one the broker uses. This incomplete knowledge is highly applicable to many circumstances since agents need not be cooperative, friendly, or trustworthy. It is unreasonable to expect that these types of agents should be required to share specifications of the effects of messages or that they would even have to believe this information were they told it. However, even under these circumstances it should

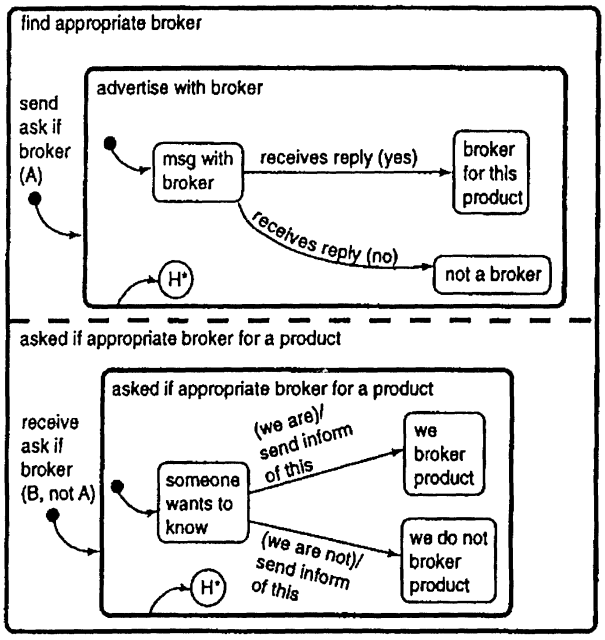

Figure 7: A statechart that describes the activities of both parties in a conversation

be possible for one agent to effectively communicate with another. Further, even if a CP is not defined, the FLBC standard effects provide a "fail soft" mechanism. If a CP is not defined, then the system would be able to do something useful with the message and not always require manual intervention.

Some situations call for less uncertainty. In these situations communication partners can get together and define multi-agent CPs. Figure 7 contains an example of such a policy. The dotted line is the statechart feature that expresses concurrency and, for CPs, also separates the actions taken by separate conversation partners. The idea behind these multi-agent CPs is that each participant is committing to making public the messages it *generally* sends in a conversation and the rules it employs in choosing among alternatives. The exposure of these messages and rules are expected to result in CPs that have fewer unexpected subdialogs (as shown in Figure 5). This would certainly be more effort than defining a single-agent CP and would, therefore, only be attempted with conversation partners with whom many messages are exchanged and who have special needs.

Defining a multi-agent conversation policy does not bind the partners to never sending a message that does not conform to the CP. In fact, it opens up new ways in which the partners can deviate from the CP—for example, with subordinations, corrections, and interruptions. Consider the following situation in which two agents, A and B, use the CP shown in Figure 7. Agent A uses the top of the CP while Agent B uses the bottom.

1. Agent A has started the CP by sending a message asking Agent B if it is a broker for disk drives (top half of the figure). It asserts that this message is part of a new conversation identified by conv874.

2. Agent B receives this message (bottom half of the figure) and tries to determine an answer to the question. Now, Agent B is only a broker for disk drives of a certain price range. In order to answer Agent A's question, Agent B sends a question back to A starting a subordinate conversation (identified by conv921) to determine the price of the disk.

3. Agent A receives this message that is a question and which states that it is related to conversation conv874 in that it is starting a conversation subordinate to it. Agent A composes a response that states the price of the disk drive, adds the information that it is a reply to another message and is part of conversation conv921, and sends it back to Agent B.

4. Agent B receives this message and determines that it has completed the subordinate conversation. It returns to executing the original conversation. In this CP it is in state someone wants to know. It attempts to make a transition out of it. Agent B's knowledge base now contains the information necessary to determine that it does sell the product so it takes the top transition and generates the event send inform of this.

5. And so on...

The question Agent B sent in step 2 is not part of the CP but was sent as part of the process of answering Agent A's question. The way the CP is defined suggests that Agent A's question is not a common one. If it were, then these two agents should probably get together and refine this CP.

4 Extended example

In this section I explore in-depth an example adapted from Greaves, et al. (Greaves, Holmback, & Bradshaw 1999, Figure 1). Their example is specified for the KAoS system (Bradshaw *et al.* 1997) using a form of finite state machine. In Figure 8 this conversation policy is shown as a statechart; the sequencing and policies covering the conversation are essentially equivalent to the specification by Greaves et al. The top half of this figure describes the contracting process for the contractor; the bottom half is for the supplier. The purpose of the exposition of this example is to demonstrate how a multi-agent conversation policy can be used to control the flow of messages, contrast this with how messages are handled via an inference-based process, and show how the inference-based processing can be integrated with the policy-based handling in order to deal with exceptions to the policy.

In the following I list the messages that might be sent in fulfillment of this conversation policy. I use a Prolog-based representation for the FLBC messages for space considerations. The `send(A, B, C)` predicate should be interpreted as Agent A sends message Agent C to the agents listed in B. Further, `msg(S, R, F, C, X)` should be interpreted as speaker S sending a message to recipient R with illocutionary force F, content C, and with relevant context X.

4.1 Basic processing

The following message is sent from Agent C to Agent S1 (line #1). This is a request from the sender to the recipient that the recipient send back to Agent C an offer in which Agent S1 reserves a hotel room for Agent C in Delft from June 6 to June 10.

```
send(c, [s1],                                    [Message #1]
  msg(S, R, request,
      send(R, S,
        msg(R, S, offer,
         [reserve(z),
          Agent(z, R),
          Object(z, x),
          room(x),
          beginDate(z, time(1999, 6, 6)),
          endDate(z, time(1999, 6, 10)),
          location(x, delft)])),
     [cp(contractProcess),
      convID(v423),
      ackPolicy([S, R], [parse], yes),
      timeSent(time(1999, 5, 11)),
      msgID(c45)]))
```

Of course, predicates in this term are simplified versions of what actual applications would require; for instance, information about the room and the location would have to be markedly more detailed.

When this message is sent by Agent C, the "contract process" conversation policy is invoked since both the `cp()` term in the context matches the process's name and the message itself matches the process's trigger (see the transition at the far left of Figure 8). The default transition in the top half is taken to put the contractor in the "supplier knows" state.

When the supplier receives this message, it determines from the `ackPolicy()` term that it must acknowledge to the contractor when it successfully parses this message. This requirement of "parsing" is more restrictive than simply receiving the message but is less so than fully processing the message. This acknowledgment should indicate to Agent C that the sender has been able to parse the message and knows what each of the terms mean. This acknowledgment does *not* tell Agent C either that the sender has evaluated the original message or that the sender is going to respond to the message. The purpose of this message is to confirm that the receiving agent is on-line and that the agent communication system is successfully routing messages.

```
send(s1, c,                                      [Message #2]
  msg(S, R, inform,
      parse(S, msgID(c45)),
```

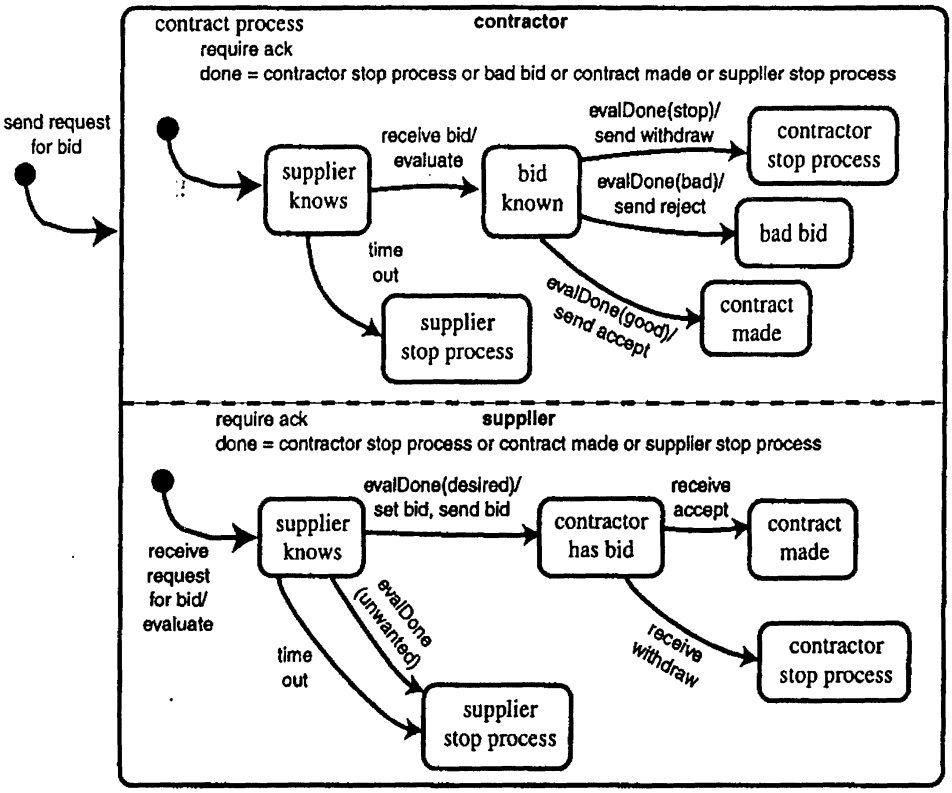

Figure 8: Contract conversation policy

```
[cp(contractProcess),
 convID(v423),
 msgID(s56)]))
```

Agent S1 sends Message #2 back to Agent C. Having received this acknowledgment, Agent C can reasonably assume that Agent S1 received the message. This takes one level of uncertainty out of the processes of interpreting messages and managing conversations.

For the most part these acknowledgment messages lie outside the part of the message processing framework that I want to concentrate on. Notice, however, that the sequencing of the contract process conversation policy remains the same whether or not an acknowledgment is requested. The decision about acknowledgment could be defined as a variable to be agreed upon by the conversants before the conversation begins. It could even be agreed upon for all conversations between the two parties when they first begin to exchange messages. This provides one means of reducing the frequency of the specific problem addressed in §3. For the rest of this explication I assume that these acknowledgment messages are being properly sent, received, and processed.

The supplier, having received the request for bid (Message #1), takes the transition to the left of the "supplier knows" state in the bottom half of the statechart. Taking this transition fires Agent S1's "evaluate" event. The agent evaluates the request to send a bid. How this evaluation is to proceed is not specified by the conversation policy. It is not observable by Agent C; its inner workings do not affect Agent C; and it does not need to be known by Agent C. The evaluation function would have to accept arguments

with the following form:

```
evaluate(request,
  send(s1, c, msg(s1, c, offer, [...])))
```

Thus, Agent S1 will evaluate a request that Agent S1 send an offer to Agent C on a certain item (described above in Message #1). Assume that the output of this evaluation is a term whose internal representation is

```
determination(msgID(c45), desired)
```

This assumption does not affect the generality of this discussion; it simply provides specifics we can use to proceed.

Whenever an evaluation is completed, it sends the event `evalDone`. This triggers the statechart to attempt to take the two transitions leading out of "supplier knows" in the bottom half of the statechart. Since the value was determined to be "desired," the system takes the top transition. In doing this the system sends the events that cause the system to set the bid and then send it back to Agent C.

```
send(s1, c,                                [Message #3]
  msg(S, R, offer,
      [reserve(z),
       Agent(z, s1),
       Object(z, x),
       room(x),
       beginDate(z, time(1999, 6, 6)),
       endDate(z, time(1999, 6, 10)),
       location(x, delft),
       price(x, $200)],
      [convID(v423),
       msgID(s57)]))
```

This message contains the information from the request for an offer message plus the `price()` predicate.

The contractor receives this offer. It determines that this message belongs to conversation v423 so it invokes the already-begun "contract process" conversation policy. It currently is in the "supplier knows" state. Having received the offer, it takes the transition to the "bid known" state and simultaneously fires the `evaluate` function. The evaluate function would have to accept arguments with the following form:

```
evaluate(offer, [...])
```

Thus, Agent C will evaluate an offer for a reservation of a room at $200. Again, when this evaluation is done, it sends the event `evalDone`. This triggers the statechart to attempt to take the three transitions leading out of the "bid known" state in the top half of the statechart. Assume the value was determined to be acceptable. This directs the system to take the transition to the "contract made" state. In doing this, the system generates the event that causes the message accepting the bid to be sent:

```
send(c, s1,                                [Message #4]
  msg(S, R, accept, msgID(s57),
      [convID(v423),
       msgID(c48)]))
```

The supplier receives this acceptance. It determines that this message belongs to conversation v423 so it, again, invokes the "contract process" policy. It currently is in the "contractor has bid" state. Having received the acceptance, it takes the transition to the "contract made" state.

The above example demonstrates how a conversation policy can provide fairly straight-forward message processing. The complexity of message handling is dramatically lessened by the pre-defined structure. One cost of this simplification is the added requirement that the conversation partners have to agree beforehand that they will use this policy. Further, for them to agree to use this policy it must already be defined and they must both have access to it. This is a difficulty that grows increasingly difficult with the number of conversation partners and with the types of conversations the agent engages in.

4.2 Inference-based message handling

For a moment let us consider how the system can process messages without defining conversation policies. Assume that Agent B receives the following message:

```
msg(a, b, request,                         [Message #5]
  send(b, a,
    msg(b, a, inform, [x]:name(b, x))),
```

```
[ackPolicy([a, b], [parse], yes),
 timeSent(time(1999, 5, 12)),
 msgID(a63)])
```

This is a request from Agent A to Agent B that Agent B inform Agent A of Agent B's name (more directly, "x such that x is the name of Agent B"). According to the definition of the standard effects of request (Moore 1998b), the following are the results of this message.

```
considerForKB a wants do(b, send(b, a,
    msg(b, a, inform, [x]:name(b, x)))
considerForKB a wants (b wants do(b, send(b,
    a, msg(b, a, inform, [x]:name(b, x))))
```

This specification indicates to the developer of Agent B that Agent A wants the effects on Agent B of this message to be the following two items: 1) Agent B should consider adding to its knowledge base that Agent A wants Agent B to send a message to Agent A informing it of Agent B's name; 2) Agent B should consider adding to its knowledge base that Agent A wants Agent B to want to send that message. Because of the diversity of agent models, it is basically impossible to provide a general accounting of what the agent would do in response to this message and these two items in particular. What I provide below is an accounting of two reasonable approaches to defining a general means of handling these demands.

4.2.1 Simple agent

What I am describing here is how a simple agent might implement the `receive` function when this "request to inform" message is received:

```
receive(msg(a, b, request,       [Message #6]
  send(b, a,
    msg(b, a, inform, [x]:name(b, x))),
  [ackPolicy([a, b], [parse], yes),
   timeSent(time(1999, 5, 12)),
   msgID(a63)]))
```

Little about the agent model can be assumed other than the agent's ability 1) to receive FLBC messages, 2) to interpret them, and 3) to use the FL-SAS (Moore 1998b) to initially process the message. In the FL-SAS process, the agent verifies that the message is valid and well-formed, composed of terms it knows the meaning of. It then ensures that it knows all the referents in the content of the message. In this message, the only referent is "b", the recipient of the message. Agent B next verifies that the message's content is compatible with its force; that is, it ensures that it is possible to "request" to "inform about the name of b." Finally, it ensures that the message as a whole makes sense.

```
receive(msg(From, b, request,
      send(b, From,
           msg(b, From, inform, WhatInfo),
      Context))) :-
  rightToKnow(From, WhatInfo),
  determineAnswer(From, WhatInfo, Answer),
  createNewContextTerm(Context,
       responseTo, NewContext),
  send(b, From,
       msg(b, From, inform, Answer),
       NewContext).
```

Figure 9: Processing a request to inform with a simple agent

After the above steps are completed for this message (as they are for *all* incoming messages), Agent B is reasonably sure that it can process this message. This should simplify later processing of this message in something like the same way that compiling a Java program simplifies the process of executing that program—it removes, early in the process, whole groups of errors. The agent can now get on with the process of completing that part of the process defined by the `considerForKB` items above. These are not meant to be directly executed because so little can be assumed about the agent model. Thus, these two `considerForKB` statements tell the developer of the receiving agent what the message sender meant to convey with the message but does not tell him or her how to implement the `considerForKB` function nor how to respond to the message. In this simple agent the developer has chosen 1) to not maintain a model of the sending agent's intentions, and 2) to do what

```
receive(msg(From, b, request, send(b, From, msg(b, From, inform, WhatInfo), Context))) :-
  considerForKB(wants(From, do(b, send(b, From, msg(b, From, inform, [x]:name(b, x))))),
  considerForKB(wants(From, wants(b, do(b, send(b, From,
                                          msg(b, From, inform, [x]:name(b, x))))))).

considerForKB(wants(Other, do(b, Action))) :-
  /* do some processing, for example, the following */
  isBelievable(Other),
  worthRemembering(Action),
  addToKB(wants(Other, do(b, Action))).
considerForKB(wants(Other, wants(b, do(b, Action)))) :-
  /* do some processing, for example, the following */
  isBelievable(Other),
  worthRemembering(Action),
  addToKB(wants(Other, wants(b, do(b, Action)))),
  considerForIntentions(wants(b, do(b, Action))).

considerForIntentions(wants(b, do(b,
                                  send(b, From,
                                       msg(b, From, inform, WhatInfo),
                                       Context)))) :-
  /* do some processing */
  rightToKnow(From, WhatInfo),
  determineAnswer(From, WhatInfo, _),
  addToIntentions(do(b, Action)).

fulfillIntentions(do(b, send(b, From, msg(b, From, inform, WhatInfo), Context))) :-
  determineAnswer(From, WhatInfo, Answer),
  createNewContextTerm(Context, NewContext),
  send(b, From,
       msg(b, From, inform, Answer),
       NewContext).
```

Figure 10: Processing by a complex agent

the sending agent wants the receiving agent to do. Figure 9 shows a simplified Prolog representation of how this might be handled. The agent determines the recipient's right to know what it's asking, determines the answer to the question, creates a new context term (that might, for example, retain the conversation identifier and stack information, add a term indicating this is a response to a message, and add a new message identifier), and sends the reply.

4.2.2 More complex agent

While the above addressed how a simple agent would respond to a request to inform, the following looks at how a more complex agent might handle this same message. The process for this agent begins as it did for the simple agent: the FL-SAS is applied with exactly the same steps. After this the actions of the agents diverge. In this agent the developer has chosen to implement a BDI (belief, desire, intention) agent model, to maintain a model of the sending agent's intentions, and to implement an agent that can make deductions about these intentions. A preferred agent could reason defeasibly about time and obligation (see (Kimbrough & Moore 1993) for a discussion of what is needed for such a system and why it might be needed).

Figure 10 shows some snippets of Prolog code that might be used to reason about this message and how the agent would handle it. The `receive()` predicate is a fairly direct mapping from the terms shown in the beginning of §4.2. The first `considerForKB()` term specifies the restrictions on adding information about another agent's beliefs to the knowledge base. The second one does the same but also begins the process of determining (by invoking `considerForIntentions`) if the receiving agent's intentions should be affected by this incoming request. The agent's intentions are only affected if the requesting agent has a right to know the information and if the receiving agent can actually answer the question. (Of course, all these predicates could be more complicated and effective. For example, maybe the receiving agent cannot determine the answer but knows someone who might know the answer—in some cases this might be sufficient and appropriate.) Finally, the `fulfillIntentions` term would be called by some process when the agent is attempting to do what it desires to do.

4.2.3 Summary

I have described agents with two different agent models. Each uses the FL-SAS to handle the initial message processing, and each interprets FLBC messages. Other than these similarities, the underlying structure of the agents differ. However, as the sketchy existence proofs I have given in these two sections indicate, each agent will create a similar response to the incoming message, and each was able to process the message without a predefined conversation policy.

4.3 Exception handling

Another cost of the use of conversation policies (expressed as statecharts) versus inference-based methods is that—this is not going to be a surprise—they limit what can be said in a conversation. The conversation policy limits what messages can be said and when. Consider again Message #1 (at the beginning of §4.1), in which the contractor requests that the supplier send an offer for a room reservation. In response to this message the supplier begins to evaluate what it should do. The form of this predicate is shown on page 13. Suppose that the `evaluate` predicate requires more information to reach a conclusion than is provided in the incoming request. The full predicate might look something like that shown in Figure 11. This predicate has three main clauses. The first does some processing if it is able to determine that the agent desires to make an offer. The second does something else if it is able to determine that it does not want to make an offer. The last clause, and the one that is elaborated on, does some basic checking and determines that the description of the offer does not contain enough information to determine the attractiveness of the request. In this case the predicate specifies that the agent should send a message back to Agent C requesting that it inform the supplier about the number of beds in the room the agent wants.

Agent C receives this message, examines it, and determines that it is part of the `contract process` it

```
evaluate(request, send(s1, c, msg(s1, c, offer, Description))) :-
  (((/* some processing if desireable */)
   ;
   (/* some processing if unwanted */)
   ;
   (/* some processing if not enough information */
    not contains([beginDate, endDate, beds], Description),
    send(s1, [c], msg(S, R, request,
          send(R, S, msg(R, S, inform,
                  [b]:[reserve(z), Agent(z, s1), Object(z, x), room(x),
                       beginDate(z, time(1999, 6, 6)), endDate(z, time(1999, 6, 10)),
                       location(x, delft), beds(x, b)])),
          [cp(contractProcess), convID([v441, v423]), subordinate,
           timeSent(time(1999, 5, 11)), msgID(s62)])))).
```

Figure 11: Evaluation that sends unexpected message

is already involved with. It also notes (from the incoming message's context term) that this message is part of a subordinate conversation and that this message does not match any of the triggers coming out of the "supplier knows" state in the top half of the contract process statechart. Using the technique shown in Figure 5, the system is able to link the conversation policy described in §4.1 with the inferential processing described in §4.2.1 and §4.2.2. Either of these methods should result in Agent C sending back a message informing Agent S1 about the number of beds it prefers in the room it wants to have reserved. Having received this information, Agent S1 would re-start the `evaluate` predicate shown in Figure 11. From the additional information it received in this last message, the agent should be able to come to a determination as to whether or not it wants to make an offer and, thus, continue with the conversation policy.

The above demonstration provides reason for believing that the system for managing conversation policies described in this paper can be integrated with an inference-based system for interpreting messages. Certainly, the contract process conversation policy could have been re-defined to handle the request for information discussed in §4.3; however, that is not really the point. If that message were the only one that might be sent in violation of the conversation policy, then of course the policy should be revised. Unfortunately, many such messages might occur and it is not at all certain that all these messages might be foreseen. Further, if these exceptions were built into the policy then the policy would become ever-more complex and difficult to implement. And all for the benefit of exceptions that may not ever be seen by the agent. The ability to gracefully handle exceptions allows conversation policy definitions to describe a flow of conversation directly without needlessly focusing on the myriad strange twists and turns it might take.

5 Conclusion

In this paper I have described at a moderate level of detail the needed capabilities of an agent communication system that implements conversation policies. In developing this approach I guarded against several pitfalls that would limit its applicability. First, conversation partners do not have to specify down to the last detail every possible outcome that might result from sending a message. This is unreasonable and would discourage people from undertaking the application of the technology. Second, these CPs are not inflexible. They should allow different agents to implement features in different ways and should allow deviations (in specified ways that are not overly re-

strictive) from defined CPs. Third, agents have only minimal requirements placed on their architecture in order to use these CPs. Fourth, and finally, these CPs are effective under various levels of cooperation. Agents are able to apply a CP without another agent's cooperation but, if that other agent chooses to cooperate in the use of a multi-agent CP, agents gain benefits from cooperation.

I envision agents that use this system of managing conversations as a system that will evolve. The core of the system for agents that use the FLBC is the set of CPs for the standard effects associated with each message type. These CPs are the only context-free portion of the system and are, thus, the most widely applicable. This core is shared by every agent. What will happen next as the agents are used more is that the developers for each agent will develop CPs for those messages that the agent receives frequently. If it is a generally useful CP, then the developer would post it to a public repository so that others could use it. Other CPs will develop that are domain specific and still others will develop that are conversation partner specific. Each of these will have their place in the agent's repertoire of conversation management tools. Further, not only will the population of CPs grow but CPs themselves will evolve. Nuances and special cases will be added to the CPs that weren't foreseen when the CP was originally defined. Thus, when initially deployed, an agent will have a small set of simple CPs. Over time the agent will gain more CPs, and those CPs will be more complicated and more specialized for specific tasks. If all goes well, they will still be composed of reusable modules that can be shared among different agents and different agent developers.

There is much that needs to be tested and refined concerning this system. The XML DTD for statecharts must be defined. The set of core operators on statecharts must be defined. A set of scenarios must be defined that would test the capabilities of and demonstrate the effectiveness of this approach—and then a system must be built to actually operate within those scenarios. The usefulness of a single-agent CP and default standard effect CPs (both single- and multi-agent) must be demonstrated. A system for publishing and naming CPs must be developed. Further, this system must allow CPs to evolve but at the same time must synchronize the version of a CP among conversation partners. The usefulness of allowing communication among agents of differing capabilities must be carefully examined to determine how those differences effect what is said and how it is interpreted.

Even given all these needs, the promise of this system is encouraging, pointing to a future in which cooperation and knowledge-sharing might be more easily realized.

End notes

Thanks to Prof. M.S. Krishnan for his helpful suggestions for improving this paper.

For the "Workshop on Formal Models for Electronic Commerce," organized by Yao-Hua Tan and held at the Erasmus University Research Center for Electronic Commerce, Rotterdam, the Netherlands, June 2–3, 1999.

A previous version was published in the "Workshop on specifying and implementing conversation policies," held preceding *Autonomous Agents '99*, Seattle, Washington, May 1, 1999, a workshop organized by Jeff Bradshaw, Phil Cohen, Tim Finin, Mark Greaves, and Munindar Singh. File: dynamic-conv-policies.tex.

References

Bradshaw, J. M.; Dutfield, S.; Benoit, P.; and Woolley, J. D. 1997. KAoS: Toward an industrial-strength open agent architecture. In Bradshaw, J. M., ed., *Software agents*. AAAI Press. chapter 17, 375–418.

Cover, R. 1998. Extensible markup language (XML). Accessed at `http://www.sil.org/sgml/xml-.html`.

DARPA Knowledge Sharing Initiative External Interfaces Working Group. 1993. Specification of the KQML Agent-Communication Language. `http://www.cs.umbc.edu/kqml/kqmlspec.ps`. Downloaded on July 14, 1997.

Foundation for Intelligent Physical Agents. 1997. FIPA 97 specification part 2: Agent communication language. Geneva, Switzerland.

Greaves, M.; Holmback, H.; and Bradshaw, J. 1999. What is a conversation policy? In Greaves, M., and Bradshaw, J., eds., *Proceedings for the Workshop on Specifying and Implementing Conversation Policies*, 1–9. Seattle, WA: Autonomous Agents '99.

Harel, D. 1987. Statecharts: A visual formalism for complex systems. *Science of Computer Programming* 8:231–274.

Kimbrough, S. O., and Moore, S. A. 1993. On obligation, time, and defeasibility in systems for electronic commerce. In Nunamaker, Jr., J., ed., *Proceedings of the Hawaii International Conference on System Sciences*, volume III, 493–502. University of Hawaii.

Labrou, Y., and Finin, T. 1997. A proposal for a new KQML specification. Downloaded from `http://www.cs.umbc.edu/kqml/` in January 1998 (Technical Report CS-97-03).

Labrou, Y.; Finin, T.; and Peng, Y. 1999. The interoperability problem: Bringing together mobile agents and agent communication languages. In Ralph Sprague, J., ed., *Proceedings of the 32nd Hawaii International Conference on System Sciences*. Maui, Hawaii: IEEE Computer Society.

Labrou, Y. 1996. *Semantics for an Agent Communication Language*. Ph.D. Dissertation, University of Maryland, Computer Science and Electrical Engineering Department. UMI #9700710.

Light, R. 1997. *Presenting XML*. SAMS.net Publishing, 1st edition.

Litman, D. J., and Allen, J. F. 1987. A plan recognition model for subdialogues in conversations. *Cognitive Science* 11:163–200.

Moore, S. A. 1993. *Saying and Doing: Uses of Formal Languages in the Conduct of Business*. Ph.D. Dissertation, University of Pennsylvania, Philadelphia, PA.

Moore, S. A. 1998a. Categorizing automated messages. *Decision Support Systems* 22(3):213–241.

Moore, S. A. 1998b. A foundation for flexible automated electronic commerce. Working paper at the University of Michigan Business School.

Polanyi, L. 1988. A formal model of the structure of discourse. *Journal of Pragmatics* 12:601–638.

The Open Group. 1999. Statechart specification language semantics. `http://www.opengroup.org/onlinepubs/9629399/chap8.htm`. Chapter 8 of *CDE 1.1: Remote Procedure Call*, 1997.