Intelligent Management of Questions

Andrew Faulring and Brad A. Myers
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA
{faulring, bam}@cs.cmu.edu

ABSTRACT
We are part of a large project that is building a suite of intelligent agents that will help users more easily complete routine tasks such as scheduling multi-person meetings, updating project websites, and finding and reserving conference rooms. Each of these agents needs to communicate with users to clarify its interpretation of natural language requests, to request approval to perform actions, or to report on the status of tasks. Rather than have each agent implement a custom user interface for communicating with the user and require that the user deal with interruptions from each agent, we are building a unified dialog management architecture in which all the agents make requests to the central dialog manager, and the dialog manager interacts with the user. The important innovations in our dialog manager are (1) a high-level XML-based language with which agents specify the desired communication, (2) interaction techniques for repairing inferences that provide rich feedback to the agent to improve its performance through learning, (3) a method for delaying the presentation of questions to avoid interrupting users at inappropriate times, and (4) the coordination and reordering of questions from multiple agents so that users can address the questions in the most efficient manner.

KEYWORDS: dialog manager, intelligent agents, interruptions, clarification dialogs, anchors, learning, RADAR

INTRODUCTION
The goals for our dialog manager are to ask fewer questions, and to ask questions at the right time, in the right order, in the right place, and in the right way. We are building the dialog manager as part of RADAR (Reflective Agent with Distributed Adaptive Reasoning), a large interdisciplinary project building a suite of intelligent agents that will help users more easily complete routine tasks [11]. We are providing the dialog manager as a service to agent developers. Other researchers are constructing agents such as a Calendar Manager, which uses an advanced constraint-based scheduler [12], a Webmaster [14], and a Space Planner.

As a subordinate to the user, each agent will need to seek clarification, request approval, and report its status. While a group of interacting agents would prefer to communicate with each other in a machine-understandable format, an agent cannot assume that all users have agents, and so an agent may need to interpret natural language such as an email written by a user. Should the agent encounter difficulties, it may need to ask clarifying questions of the user. Often existing AI systems ask questions by proposing an action and offering the user the option of only responding yes or no. Early research showed that users usually need to adjust a few parameters to the action [8]. For instance, when asked by the agent to confirm a 10:00 meeting, the user might want to respond that the proposal is almost right but would prefer to start a half hour earlier. If the user bypasses the agent by adjusting their calendar independently, the agent is deprived of a potential learning opportunity. If instead the agent receives feedback about what parameters the user modifies, it has the opportunity to improve its performance through learning. Our dialog manager enables this kind of learning.

Users may not trust their agents enough to grant them complete autonomy. Agents may attempt to model the user’s preferences, which are often not well defined. Unfortunately, users may be unwilling to spend the time to fully describe their preferences. A user may be willing to allow an agent to autonomously negotiate a meeting time, but then may want to check the selected meeting time before the meeting can be confirmed. Also, if at some point the negotiation stalls, the agent may need to notify the user.

CENTRALIZED DIALOG MANAGER
A centralized dialog manager provides benefits to both the agent developer and to the user that would be difficult to achieve if each agent had a separate custom user interface. In our system, the agents describe their communication needs in a high-level XML-based language rather than implement an interface at the widget level, which is notoriously difficult and time consuming [9]. Prior efforts at building UIMS and model-based systems also followed this basic approach (see Szekely’s survey [13]), but attacked the much more general domain of full graphical user interfaces. Our language supports a more restricted domain. The RADAR agents communicate among themselves by using a machine-understandable representation called a template. In general, a template encodes a hierarchical collection of fields. We designed the messaging system to allow the agents to use their existing templates to communicate with the dialog manager. The agents add meta-data to the message that tells the dialog manager how to interpret the template’s fields.

SUBMITTED TO UIST’04
Agents and the dialog manager communicate using an asynchronous exchange of XML messages. After sending a message to the dialog manager, an agent may not receive a response from the user for hours, days, or ever. When a response is ready, the dialog manager attempts to transmit it to the agent. This push-model for the messaging protocol works well given the assumption that agents are always available. In the future, we could easily extend the model to allow agents to poll the dialog manager for responses.

The dialog manager provides interaction techniques that let the user repair inferences and that provide rich feedback to the agents to improve their performance through learning. We now turn our attention to the benefits for the user.

Ask Fewer Questions
The first way to help the user is to simply reduce the number of questions that the user sees. The messaging protocol allows the agent to add a new question, replace an existing question, or withdraw an existing question.

Consider the following example. One evening after you have gone home, a colleague sends an email to you and one of your students asking to meet on the following day at either 10am, 1pm, 2pm, or 4pm. Unbeknownst to your colleague, a personal appointment the following morning will keep you away from your email until about noon. Your calendar agent intercepts the email message, interprets its contents, checks your calendar, and responds that the 10am meeting is bad, but the others work. Your calendar agent sends a question to the dialog manager asking which of the remaining times you prefer. Since you are not available, the message sits in a queue. Later that evening your student responds that he has an appointment at 4pm. When your calendar agent receives that message, it instructs the dialog manager to replace the previous question with one that asks you to choose either the 1pm or 2pm time. Early the next morning, your colleague wakes up with the flu and sends emails canceling all her appointments for the day. Your calendar agent receives that message and instructs the dialog manager to withdraw the previous question regarding the meeting with your colleague. When you arrive at your workstation at noon, you do not see any messages related to the now non-existent meeting.

Suppose that your colleague had not become ill that morning. In this case, at an appropriate time after you had arrived at your workstation the dialog manager would ask you which meeting time you preferred and would send your response back to your calendar agent. Then your calendar agent would respond to the original email request in both natural language and in a machine-understandable format in case any of the recipients also had a calendar agent.

Ask at the Right Time
Many agents operate in the background awaiting the receipt of external messages that trigger a need to work on the user’s behalf. The agent’s question to the user will often be unrelated to the user’s current task. If the question is presented immediately to the user, the resulting interruption could cost the user time and negatively impact the user’s performance on the current task [10, 15]. Users will quickly turn off a system of agents that continually interrupts them at seemingly random times. McFarlane identified four known methods for deciding when to interrupt a user: immediate, negotiated, mediated, and scheduled [7]. Our dialog management architecture incorporates a mediator, called the user interruptibility sensor, which monitors the user’s activity to determine when to present questions. Existing research systems that make inferences about the user’s state include Coordinate [4], Interruption Workbench [3], AmIBusy [1], and Krause et al.’s wearable system [6]. Prior to building the AmIBusy system, Fogarty et al. completed feasibility studies to establish that it would be able to infer a user’s degree of interruptibility in real-time [2]. We are collaborating with the AmIBusy group, and our dialog manager uses AmIBusy to determine when to present questions to the user in a desktop environment. In a mobile computing environment, we are collaborating with Krause’s group, which is building “a wearable system which can determine typical user context and context transitions probabilities…” [6]. Knowledge of the user’s context may help the dialog manager choose an appropriate modality in which to present questions.

We are also considering how to incorporate a notion of urgency into the question presentation scheduling algorithm. Perhaps if a question’s urgency exceeds a threshold, it would override the user interruptibility sensor, in which case the dialog manager would immediately present the question.

Ask in the Right Order
The dialog manager cannot always present a question to the user, either because the user interruptibility sensor says not to or because the user is away from the computer. In this case, questions accumulate in a queue. Messages in that queue can be replaced or withdrawn by the issuing agent. When the time arrives to begin presenting questions to the user, the dialog manager reorders the questions from the multiple agents so that the user can address them in the most efficient manner. The agent provides a task descriptor and a priority for each question. The task descriptor, which follows a format such as “calendar/9-Apr-2004,” “web/radar.html,” or “space/WeanHall/4thFloor,” allows the dialog manager to group questions by task, for example grouping together all questions related to a particular webpage. Within a group of related tasks, the priority determines the order in which tasks are presented. The groups are then sorted based upon the highest priority of any question in the group. The dialog manager selects the group with the highest priority and presents all the questions in that group before moving on to the next group. This strategy reduces the number of task switches compared to a strategy that just presented questions in delivered order or purely in decreasing order of priority. Horvitz’s Priorities system attempts to infer the criticality of email messages and considers whether the benefit of notifying the user immediately outweighs the cost of the interruption [4]. Our agents could benefit from a Priorities-like system for determining the importance of a request. Like our dialog manager, Priorities considers the benefit of batching
notifications. Horvitz’s Notification Manager accepts notifications from multiple sources, consults user models of attention and interruptibility, and then chooses the best modality to in which to deliver the notification [4]. Both of these system focus on delivering notifications to the user, whereas our dialog manager presents interaction questions to the user.

Ask in the Right Place and the Right Way
When the time comes to ask the user a question, our dialog manager asks the question within the context of the user’s existing applications and in a manner that provides feedback to the agent for improving its performance through learning. Consider the following example. A calendar agent intercepts an email containing a meeting request and encounters difficulty interpreting the natural language request. In this example, the agent is required to check with the user before confirming a meeting.

Paula sends an email to her colleague Jason asking to meet to discuss an upcoming open house for their lab. Jason’s calendar agent intercepts the message and converts the natural language request into an “add-meeting” template, which contains all the information necessary to add a meeting to a calendar including the subject, location, start time, and duration. The calendar agent sends a question to the dialog manager requesting that Jason repair any incorrect inferences and confirm the meeting. The dialog manager presents the dialog box seen in Figure 1a. The agent’s question appears at the top followed by the fields of the add-meeting template. The agent’s natural language parsing infers values for the four template fields based upon the original text of the email request, which appears at the bottom.

Based upon its parsing of the email, the agent proposes a subject of “meeting with Paula about Lab Open House” that contains two phrases extracted from the source text: “Paula” and “Lab Open House.” To emphasize the connection between phrases in the template fields and the source text, our message protocol allows the agent to explicitly specify the location in the source text where it found a phrase. We call the referenced text within the source an anchor. The system visualizes text anchors by rendering them with a blue dotted underline, for example “Paula” in Figure 1b. When the mouse or text cursor moves over the anchor, the system highlights the anchor and any connected anchors by rendering them with a thicker blue underline, for example “Lab Open House” in Figure 1b.

ANCHORS FOR APPLICATION OBJECTS
Users will often want to answer questions in the context of their existing applications so they can see the information relevant to the question. To ask a question effectively, an agent may need to refer to an object in a legacy application. Questions about appointments are best addressed within a calendaring program such as Microsoft Outlook, Palm Desktop, or iCal. The RADAR architecture provides wrappers around common desktop productivity applications so they can interoperate with the agents and the dialog manager. The dialog manager supports anchors for objects within any existing application that supports a programmable interface capable of accessing the user visible objects. Examples include the add-in interfaces for Microsoft Office and Palm Desktop. The wrappers register themselves as anchor containers with the dialog manager. For instance, an anchor can denote an appointment in a calendar, a graphical object in a drawing program, an email message header in an email client, or a range of text within a web browser. The syntax for describing anchors contains two parts: an identifier for the anchor container, and an anchor container specific part that denotes the particular object within the container. Jason’s calendar program, Microsoft Outlook, appears behind the dialog box in Figure 1c. When Jason hovers over the first choice in the popup menu for the start time field, the blue temporary appointment appears in his calendar allowing him to see how that choice relates to other appointments in his calendar. Other systems have also provided support for referencing objects within an application. For example, the Apple Guide help system provides a mechanism for help document authors to refer to interface widgets [5].

REPAIRING INFERENCES
The dialog manager offers the user several methods for repairing the values proposed by the agent. First, the protocol allows the agent to provide multiple choices of values for a template field. When interpreting the email message, the calendar agent recognizes three time phrases, “Wed, Jul
2004 11:32:31 -0400,” “noon on Thursday,” and “Friday at 10am.” For each phrase, it calculates a confidence value that indicates the likelihood that the phrase specifies the start time of the meeting. In Figure 1c the user has activated a popup menu that shows the three choices for the start time field sorted by confidence value. As the user hovers over the choices in the popup menu, the system highlights the anchors in the email referenced by the current choice. If the correct value appears in the list of choices the user can select it, and the agent will be told which choice was selected. Sometimes the agent will not have high confidence in any of its choices for a template field. If a template field has no choices above a minimum threshold, the dialog manager draws the user’s attention to the problematic field by setting the field’s background to pink, for example the duration field of Figure 1a. In this case, the original email contained nothing to indicate the length of the meeting, so the agent provided “1 hour” as a low-confidence default value.

The user can also correct an erroneous inference by copy-and-pasting or drag-and-dropping from the source text (the email message in this example) into a template field. In this case the dialog manager creates an anchor in the source text and links the copied text to that anchor. The dialog manager returns the location of the user-created anchor to the agent as feedback to improve its learning algorithms.

Finally, the user could just edit the text field. However, this method deprives the agent of feedback to improve its learning algorithms. When presented with detailed information such as phone numbers, dates, and hard-to-spell names, we believe that the user would prefer to select choices from a popup menu or to use copy-and-paste or drag-and-drop rather than risk mistyping the information. Compared to retyping, the other options are easier for the user and provide the agent with feedback to improve its performance through learning—a win-win situation.

Who repairs inferences?
Since Paula has no agents, Jason’s agent has to interpret her natural language email request. Should the agent encounter difficulties, Jason has to repair the inferences. However if Paula had a calendar agent, it might intercept her outgoing email to Jason and attempt to add a machine-understandable add-meeting template to the message. In this case, Paula would repair any problematic inferences. Having the sender repair the inference may be preferred, since the sender knows more about the request and has an incentive that the request be understood correctly. Additionally, the work of the one sender benefits all the recipients whose agents receive a machine-understandable version of the request.

FUTURE WORK
We have several ideas for improving the dialog manager. First, we plan to extend the wrapper design to allow users to answer questions fully within a legacy application without a need for custom dialog boxes. Second, agents may need to ask multi-step questions in which the later steps depend upon prior ones. When the inter-step constraints are simple, the agent might specify the constraints explicitly. However, some constraints may require that the agent performs arbitrary computation, in which case we might extend the protocol to require the agent to be available while the question is presented to the user. Finally, we would like to extend our implementation to work in a mobile computing environment.

CONCLUSION
We have described a dialog manager that provides benefits to both the user and the agent developer. Our implementation has focused on supporting intelligent agents that operate in the background on a user’s computer, though it may be useful for any program that needs to communicate with the user. For users, our dialog manager asks fewer questions, and asks questions at the right time, in the right order, in the right place, and in the right way. For agent developers, our dialog manager allows them to describe their communication needs rather than implement a separate custom user interface. Furthermore, our dialog manager provides interaction techniques that allow the user to easily correct inference errors made by the agent and returns rich feedback about such corrections to the agent.

ACKNOWLEDGEMENTS
The authors thank James Fogarty, Jay Modi, Jeffrey Nichols, Jean Oh, Bill Scherlis, Asim Smailagic, Desney Tan, Anthony Tomasic, and John Zimmerman. Andrew Faulring is supported by a NSF Graduate Research Fellowship. This material is based upon work supported by DARPA under Contract No. NBCHC030029.

REFERENCES