An Experiment in the Design of Software Agents

Henry A. Kautz, Bart Selman, Michael Coen, and Stephen Ketchpel
AI Principles Research Department
AT&T Bell Laboratories
Murray Hill, NJ 07974
{kautz, selman}@research.att.com
mhcoen@ai.mit.edu
ketchpel@cs.stanford.edu

Abstract

We describe a bottom-up approach to the design of software agents. We built and tested an agent system that addresses the real-world problem of handling the activities involved in scheduling a visitor to our laboratory. The system employs both task-specific and user-centered agents, and communicates with users using both email and a graphical interface. This experiment has helped us to identify crucial requirements in the successful deployment of software agents, including issues of reliability, security, and ease of use. The architecture we developed to meet these requirements is flexible and extensible, and is guiding our current research on fundamental principles of agent design.

1 Introduction

There is much recent interest in the creation of software agents [Etzioni *et al.*, 1992a; Maes, 1993; Dent *et al.*, 1992; Shoham, 1993]. A range of different approach and projects use the term "agents", ranging from simple shell programming program systems (such as the Hewlett-Packard agent interface) to agents incorporating sophisticated real-time planning (such as Etzioni's softbots).

In our own approach, agents assist users in a range of daily, mundane activities, such as setting up meetings, sending out papers, locating information in multiple databases, tracking the whereabouts of people, and so on. Our objective is to design agents ("bots") that blend transparently into normal work environments, while relieving users of low-level administrative and clerical tasks. We take the practical aspect of software agents seriously: users should feel that the bots are reliable and predictable, and that the human user remains in ultimate control.

One of the most challenging aspects of agent design is to define specific tasks that are both feasible using current technology, and are truly useful to the everyday user. Furthermore, it became clear during the testing of our initial prototype that users have little patience when it it comes to interacting with software agents. We therefore paid special attention to the user interface aspects of our system. In particular, whenever possible, we opted for graphically-oriented interfaces over pure text-based interfaces. In addition, reliability and error-handling is crucial in all parts of a software agent system. The real world is an unpredictable place: messages between agents may be lost or delayed, people may respond inappropriately to requests, and so forth.

Our approach has been bottom-up. We began by identifying possible useful and feasible tasks for a software agent. The first such task we choose involved the activities surrounding the scheduling of a visitor to our lab. We designed and implemented a set of software agents to handle this task. We deliberately made no commitment in advance to a particular agent architecture. We then tested the system with an actual visitor. The feedback from this test led to many improvements in the design of the agents and the human/agent interfaces, as well as the development of a general and flexible framework for agent interaction. We are currently in the process of the next round of testing.

In this paper we describe the design of our software agents, and the lessons we have learned during the construction and testing of the system. Our agents currently run at and communicate between Bell Laboratories, M.I.T., and, soon, the University of Toronto.

We believe that the bottom-up approach is crucial in identifying the necessary properties of a successful agent platform. As we will see in this paper, our initial experiments have already led us to formulate some key properties. Examples include the separation of task-specific agents from user-centered agents, the need to handle issues of security and privacy, and as mentioned above, the need for good human interfaces and high reliability. We believe these kinds of issues should also be addressed by theoretical work in agent design. For example, ultimately one would like to be able to prove that no communication deadlocks can occur in an agent system.

2 The Visitorbot

As noted in the introduction, selecting an appropriate task for software agents to perform is itself a challenge. Agents must provide solutions to real problems that are important to real users. The whole raison d'être for software agents is lost if they are restricted to handling toy examples. On the other hand, more complex tasks frequently include a whole range of long-range research issues, such as understanding unrestricted natural language.

After considering a number of possible agent tasks, we settled on the problem of scheduling a visitor to our lab.[1] This job is quite routine, but consumes a substantial amount of the host's time. The normal sequence of tasks consists of announcing the upcoming visit by email; collecting responses from people who would like to meet with the visitor, along with their preferred meeting times; putting together a schedule that satisfies as many constraints as possible (taking into account social issues, such as not bumping the lab director from the schedule); sending out the schedule to the participants, together with appropriate information about room and telephone numbers; and, of course, often rescheduling people at the last minute because of unforeseen events.

We decided to implement a specialized software agent called the "Visitorbot" to handle these tasks. After examining various proposed agent architectures (e.g. [Etzioni *et al.*, 1992b; Shoham, 1993]), we decided that it was necessary to first obtain practical experience in building and a using a concrete basic agent, before committing to any particular theoretical framework. Our initial implementation was a monolithic agent, that communicated directly with users via email. The program was given its own login account ("Visitorbot"), and was activated upon receiving email at that account. (Mail was piped into the Visitorbot program using the ".forward" facility of the Unix mail system.)

Our experience in using the Visitorbot in scheduling a visit led to the following observations:

1. Email communication between the Visitorbot and humans was cumbersome and error-prone. The users had to fill in a pre-defined form to specify their preferred meeting times.[2] Small editing errors by users often made it impossible to automatically process the forms, requiring human intervention. Moreover, users other than

[1] See also [Dent *et al.*, 1992; Maes and Kozierok, 1993], that describe the design of a software agents that *learn* how to assist users in scheduling meetings and managing their personal calendars.

[2] We considered various forms of natural language input instead of forms. However, the current state of the art in NLP cannot parse or even skim unrestricted natural language with sufficient reliability. On the other hand, the use of restricted "pseudo"-natural language has little or no advantage over the use of forms.

the host objected to using the Visitorbot at all; from their point of view, the system simply made their life harder.

Based on this observation, we realized that an easy to use interface was crucial. We decided the next version of the Visitorbot would employ a graphical interface, so that users could simply click on buttons to specify their preferred meeting times. This approach practically eliminated communication errors between people and the Visitorbot, and was viewed by users as an improvement over the pre-Bot environment.

2. There is a need for redundancy in error-handling. For example, one early version of the Visitorbot could become confused by bounced email, or email responses by "vacation" programs. Although our platform has improved over the initial prototype, there is still much room for improvement. The agents must react more or less predictably to both foreseen errors (e.g., mangled email), and unforeseen errors (e.g., a subprocess invoked by the bot terminates unexpectedly). Techniques from the area of software reliability and real-time systems design could well be applicable to this problem. (For example, modern telephone switching systems have a down-time of a few minutes per year, because they continuously run sophisticated error-detection and recovery mechanisms.)

3. The final task of creating a good schedule from a set of constraints did not require advanced planning or scheduling techniques. The Visitorbot translated the scheduling problem into an integer programming problem, and solved it using a general integer programming package (CPLEX). An interesting advantage of this approach is that was easy to incorporate soft constraints (such as the difference between an "okay" time slot and a "good" time slot for a user).

This experience led us to the design shown in Figure 1. This design includes an agent for each individual user in addition to the Visitorbot. For example, the agent for the user "selman" is named "selmanbot", for "kautz" is named "kautzbot", and so on. The userbots mediate communication between the Visitorbot and their human owners. For example, when the Visitorbot wishes to obtain a set of preferences for meetings times from a user, it mails the request to the corresponding userbot, rather than directly to the user. The userbot then determines the preferred mode of communication with the user. In particular, if the user is logged in on an X-terminal, the userbot will create a pop-up window on the user's screen, containing a menu of meeting times. The user simply clicks on buttons to indicate his or her preferences, as shown in Figure 2. After obtaining the preferences, the userbot generates a message containing the preferences and mails it back to the Visitorbot. If the user does not respond, or the userbot is unable to determine the display where the user is working, the Visitorbot forwards the request to the user by email as a text form.

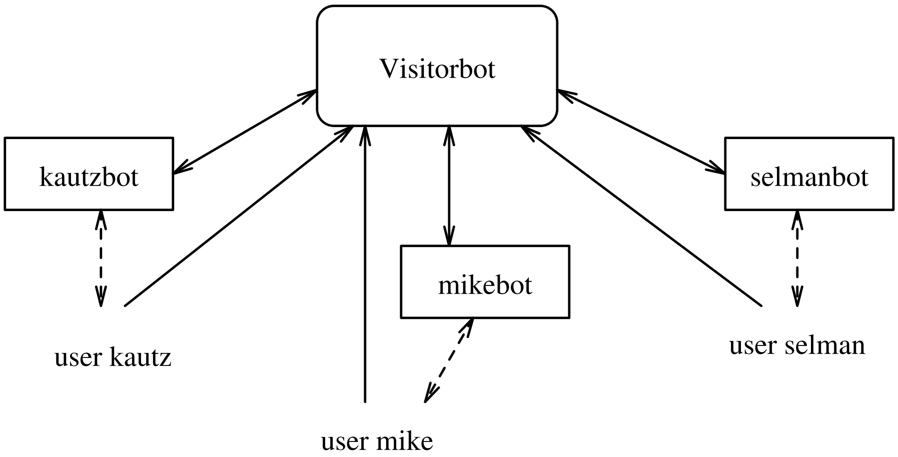

Figure 1: Current architecture of the agent system. Solid lines represent email communication; dashed lines represent both graphical and email communication.

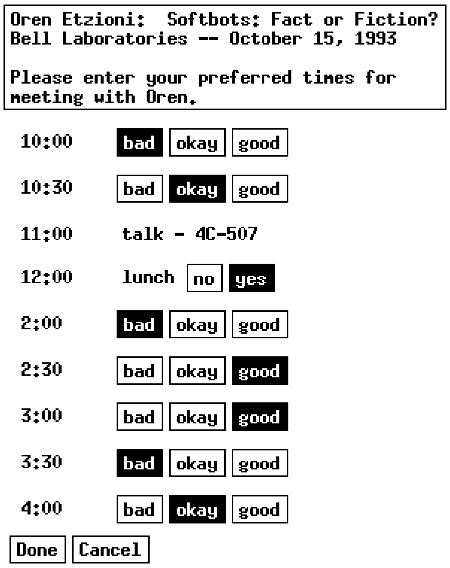

Figure 2: Graphical interface created by a userbot, in response to a query from the Visitorbot, asking for the user's preferred meeting times.

There are several important advantages of this design. First, the Visitorbot does not need to know about the user's display, and does not need permission to create windows on that display. This means, for example, that a Visitorbot at Bell Labs can create a graphical window at any site that is reachable by email where there are userbots. This was successfully tested using the mhcoenbot at M.I.T.[3] The separation of the Visitorbot from the userbot also simplifies the design of the former, since the userbots handle the peculiarities of addressing the users' displays. Even more importantly, the particular information about the user's location and work habits does not have to be centrally available. This information can be kept private to the user and his or her userbot.

The X-windows interface created by the userbots is quite flexible. The specific appearance of the pop-up window is not hard-wired, but instead is created by interpreting the message sent by the Visitorbot, which is written in a simple form description language. This makes it easy for the Visitorbot (and eventually other kinds of agents) to ask a userbot to display different kinds of information.

This design facilitates the addition of new modalities for communication between users and bots. For example, we are currently working on adding a telephone interface with a speech synthesizer to userbots. This will enable a userbot to place a call to its owner (if the user so desires), read the talk announcement, and collect the user's preferences by touch-tone. Note that this extension would not require any modification to the Visitorbot itself. Note too that users at different locations, using different kinds of machines, can run different userbots, of different levels of sophistication.

3 Privacy and Security

Early discussions with potential users made it clear that privacy and security are central issues in the successful deployment of software agents. Some proposed agent systems would filter through all of the user's email, pulling out and deleting messages that the agent would like to handle. We found that users generally objected to giving a program permission to automatically delete any of their incoming mail. An alternative approach would give the bot authority to read but not modify the user's mail. The problem with this is that the user's mail quickly becomes polluted with the many messages sent between the various bots.

[3]Sometimes it is possible to create a remote X-window over the internet, but this is prone to failure. Among other problems, the user would first have to grant permission to ("xhost") the machine running the Visitorbot program; but note that the user may not even know the identity of machine on which the Visitorbot program is running.

Our solution to this problem has been to create a pseudo-account for each userbot, with its own mail alias. Mail sent to this alias is piped into a userbot program, that is executed under the corresponding user's id. This gives the instantiated userbot the authority, for example, to create a window on the user's display. Any "bot mail" sent to this alias is not seen by the user, unless the userbot explicitly decides to forward it.

Each user has a special ".bot" directory, which contains information customized to the particular user. These files specify the particular program that instantiates the userbot, a log of the userbot mail, and the user's default display id. In general, this directory contains user-specific information for the userbot. It is important to note that this directory does not need to be publicly readable, and can thus contain sensitive information for use by the userbot. Examples of such information include the names of people the bot is not supposed to respond to, unlisted home telephone numbers, the user's personal schedule, and so on.

Thus, userbot provide a general mechanism for the distribution and protection of information. For a concrete example, consider the information you get by running the "finger" command. Right now, you have to decide whether your home phone number will be available to everyone on the internet, or no one at all. A straightforward task of your userbot would be to give out your phone number via email on request from (say) faculty members at your department and people listed in your address book, but not to every person who knows your login.[4]

4 Bots vs. Programs

An issue that is often raised is what exactly distinguishes software agents from ordinary programs. In our view, software agents are simply a special class of programs. Perhaps the best way to characterize these programs is by a list of distinguishing properties:

Communication: Agents engage in complex and frequent patterns of two-way communication with users and each other.

Temporal continuity: Agents are most naturally viewed as continuously running processes, rather than as functions that map a single input to a single output.

Responsibility: Agents are expected to handle private information in a responsible and secure manner.

[4] Of course, this setup can only be as secure as the underlying email system. Cryptographic techniques can be used to reach any desired level of security [Rivest *et al.*, 1978].

Robustness: Agents should be designed to deal with unexpected changes in the environment. They should include mechanisms to recover both from system errors and human errors. If errors prevent completion of their given tasks, they still must report the problem back to their users.

Multi-platform: Agents should be able to communicate across different computer system architectures and platforms. For example, very sophisticated agents running on a high-end platform should be able to carry out tasks in cooperation with relatively basic agents running on low-end systems.

Autonomy: Advanced agents should have some degree of decision-making capability, and the ability to choose among different strategies for performing a given task.

Note that our list does not commit to the use of any particular form of reasoning or planning. Although advanced agents will certainly need general reasoning and planning capabilities, our experiments have shown that interesting agent behavior can already emerge from systems of relatively simple agents.

5 Conclusions

We have described a bottom-up approach to the design of software agents. We have built and tested a bot system that addresses a real-world problem, namely handling the communication involved in scheduling a visitor to our laboratory.

Our experiment helped us to identify crucial factors in the successful deployment of agents. These include issues of reliability, security, and ease of use. Security and ease of use were obtained by separating task-specific agents from personal userbots. This architecture provides an extensible and flexible platform for the further development of practical software agents. New task-specific agents immediately obtain a flexible and reliable medium for communicating with users via the userbots. Furthermore, additional modalities of communication, such as speech or FAX, can be added to the userbots, without modifying the task-specific bots.

Our experiments have convinced us that software agents must blend unobtrusively into the ordinary work environment in order to be accepted by real users. Graphical interfaces greatly facilitate the ease of use and reliability of software agents.

We believe that the empirical approach taken in this paper is essential for guiding theoretical research toward the truly central and difficult issues in agent design. This paper has described the first major phase in our overall agent research project. We are currently using the experience we have obtained to guide us in the design of a higher-level bot programming language, and in the creation of tools for formally specifying reliable software agents. The higher-level programming language should greatly simplify the implementation of our next generation of software agents. Moreover, formal specification tools will help us guarantee (at least in part) that our agents are robust and reliable.

6 Acknowledgements

We thank Oren Etzioni for stimulating discussions about softbots during his visit to Bell Labs, leading us to initiate our own softbot project. We also thank Ron Brachman, Mark Jones, David Lewis Chris Ramming, Eric Sumner, and other members of our center for useful suggestions and feedback.

References

[Dent *et al.*, 1992] Dent, Lisa; Boticario, Jesus; McDermott, John; Mitchell, Tom; and Zabowski, David 1992. A personal learning apprentice. In *Proceedings of the Tenth National Conference on Artificial Intelligence*. AAAI Press/The MIT Press. 96–103.

[Etzioni *et al.*, 1992a] Etzioni, Oren; Hanks, Steve; Weld, Daniel; Draper, Denise; Lesh, Neal; and Williamson, Mike 1992a. An approach to planning with incomplete information. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*. 115–125.

[Etzioni *et al.*, 1992b] Etzioni, Oren; Lesh, Neal; and Segal, Richard 1992b. Building softbots for unix. Technical Report.

[Maes and Kozierok, 1993] Maes, Pattie and Kozierok, Robyn 1993. Learning interface agents. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*. AAAI Press/The MIT Press. 459–464.

[Maes, 1993] Maes, Pattie, editor 1993. *Designing Automomous Agents*. MIT/Elsevier.

[Rivest *et al.*, 1978] Rivest, R. L.; Shamir, A.; and Adleman, L. 1978. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM* 21(2):120–126.

[Shoham, 1993] Shoham, Yoav 1993. Agent-oriented programming. *Artificial Intelligence* 60:51–92.