An Experiment in the Design of Software Agents

Henry Kautz, Bart Selman,
Michael Coen, Steven Ketchpel, and Chris Ramming
AI Principles Research Department
AT&T Bell Laboratories
Murray Hill, NJ 07974
{kautz, selman, jcr}@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 principles of agent design.

Introduction

There is much recent interest in the creation of software agents. A range of different approaches and projects use the term "agents", ranging from adaptive user interfaces to systems that use planning algorithms to generate shell scripts (Maes 1993; Dent *et al.* 1992; Shoham 1993; Etzioni *et al.* 1992).

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 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 agents are reliable and predictable, and that the human user remains in ultimate control.

One of the most difficult 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 ordinary users; 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. The key feature of the framework is the use of personalized agents called "userbots" that mediate communication between users and task-specific agents. We are now in our third round of implementation and testing, in which we are further refining and generalizing our userbots so that they can communicate with software agents developed by other research groups, such as the "softbots" of Etzioni *et al.* (1992).

We believe that the bottom-up approach is crucial in identifying the necessary properties of a successful agent platform. 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.

Taskbots and Userbots

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 range of long-term 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

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

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. (We use the suffix "bot" for "software robot".) After examining various proposed agent architectures (Etzioni, Lesh, & Segal 1992; 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. (We considered various forms of natural language input instead of forms. However, the current state of the art in natural language processing 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.) Small editing errors by users often made it impossible to process the forms automatically, requiring human intervention. Moreover, users other than 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 that 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, more needs to be done. 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 only 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 commercial 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 Fig. 1. This design includes an agent for each individual user in addition to the visitorbot. For example, the agent for the user "kautz" is named "kautzbot", for "selman" is named "selmanbot", and so on. The userbots mediate communication between the visitorbot and their human owners.

The normal interaction between the visitorbot and the users proceeds as follows. The initial talk announcement is mailed by the visitorbot to each userbot. The userbot then determines the preferred mode of communication with its owner. In particular, if the user is logged in on an X-terminal, the userbot creates a pop-up window on the user's screen, containing the announcement and a button to press to request to meet with the visitor, as shown in the left-hand window in Fig. 2. If the user clicks on "yes", the userbot passes this information back to the visitorbot, which responds with a request to obtain the user's preferred meeting times. The userbot then creates a graphical menu of meeting times, as shown in the right-hand window in Fig. 2. The user simply clicks on buttons to indicate his or her preferences. The userbot then generates a message containing the preferences and mails it back to the visitorbot. If the userbot is unable to determine the display where the user is working, or if the user fails to respond to the pop-up displays, the userbot forwards the request from the visitorbot via email to the user as a plain text form.

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 with the "mhcoenbot" running at MIT.[2] 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

[2]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. Even if the identity of the machine is known, it may be impossible to serve X-windows directly due to "firewalls" or intermittent connections.

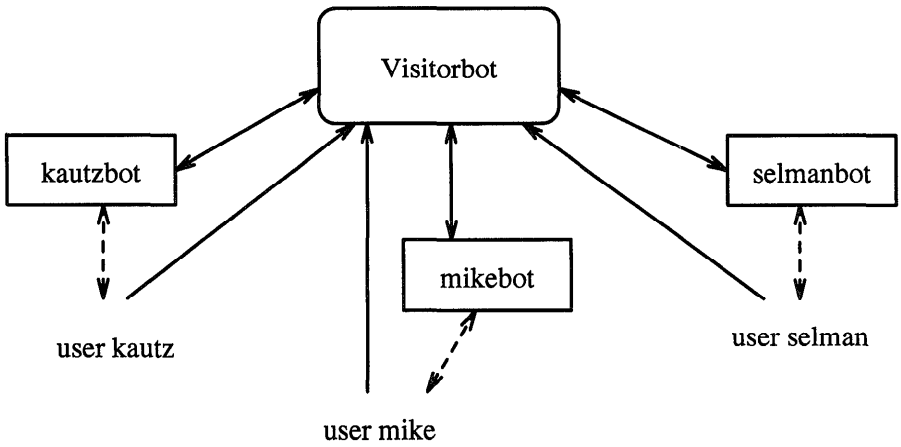

Figure 1: Current architecture of the agent system. Solid lines represent email communication; dashed lines represent both graphical and email communication.

her userbot.

Another advantage of this design is that different users, who may have access to different computing resources, can run different userbots, of varying levels of sophistication. Thus, everyone is not restricted to a "least common denominator" type interface.

Perhaps the most important benefit of the design is that a task-specific agent (such as the visitorbot) is not tied to any specific form of communication. The task-specific agent specifies *what* information is to be transmitted or obtained, but not *how* the communication should take place. The userbot can then employ a wide range of media, such as graphics, voice, FAX, email, etc. for interacting with its owner. The userbot also can take into account its owners preferences and such factors as the owners whereabouts and current computing environment in deciding on the mode of communication. For example, a userbot could incorporate a telephone interface with a speech synthesizer. This would enable a userbot to place a call to its owner (if the owner so desires), read the talk announcement, and collect the owner's preferences by touch-tone. Note that this extension would not require any modification to the visitorbot itself.

Refining the Userbot

Tests of the visitorbot/userbot system described in the previous section showed that users greatly preferred its ease of use and flexibility over our initial monolithic, email-based agent. Now that we had developed a good basic architecture, the logical next step was to incorporate new task-specific agents. In order to do so, we undertook a complete reimplementation of the system. In the new implementation, all visitorbot-specific code was eliminated from the userbots. We designed a simple set of protocols for communication between task-specific agents and userbots. Again, our approach was pragmatic, in that we tried to established a minimal set of conventions for the applications we were considering, rather than immediately trying to create a full-blown agent interlingua.

In brief, bots communicate by exchanging email which is tagged with a special header field, "XBot-message-type". The message type indicates the general way in which the body of the message (if any) should be processed by the receiver. For example, the message type "xchoices" means that the message is a request for the receiver to make a series of *choices* from among one or more sets of alternatives described in the body of the message. The inclusion of the field "XBot-return-note" in the message indicates that the result of processing the message should be mailed back to the sender. The communication protocol establishes the syntax for the data presented in the body of each message type, and the format of the data that results from processing the message. However, the protocol deliberately does not specify the exact method by which the processing is carried out. For example, a userbot may process an xchoices message by creating a pop-up menu, or calling the user on the telephone, or simply by consulting a database of defaults that the user has established.

When applications are developed that demand novel kinds of interactions with userbots, the communication protocols can be extended by adding new message types. This will require the creation of mechanisms for distributing "updates" to the userbots to handle the extensions (an issue we return to below). So far, however, only a very small number of message types (namely, ones for requesting choices, requesting help, and simply conveying a piece of information) have been needed. One question that more experience in building bots will answer is whether the number of basic message types is indeed bounded and small, or if new types are often needed with new applications.

In essence, then, the messages that task-specific bots and userbots exchange can be viewed as *intensions* – such a request to make a choice – rather than *extensions* – for example, if one were to mail a message containing a program that draws a menu on the screen when executed.[3] In this

[3]This description of messages as intensions versus extensions

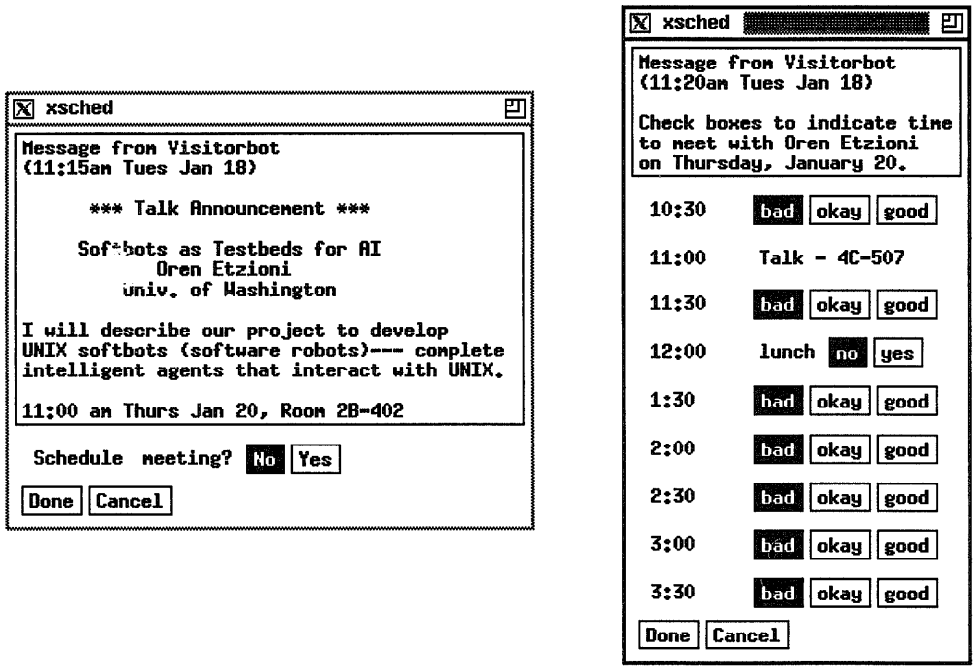

Figure 2: Graphical interfaces created by a userbot in response to messages from the visitorbot. The left window is created by processing a talk announcement; the right, by a request for the user's preferred meeting times.

regard it is informative to contrast our approach with that used in *Telescript*, the electronic messaging language created General Magic. In Telescript, messages are full-fledged programs, and are executed on the receiving machine by a fixed interpreter. Thus, Telescript messages are purely extensional. While the Telescript approach has the advantage that the reception of a message can initiate arbitrarily novel and complex processing, this must be weighed against the fact that any flexibility in the *way* in which the interaction takes place with the user must be built into each and every message.

One aspect of the preliminary userbot that some users found objectionable was the fact that various windows (such as those in Fig. 2) would pop-up whenever the userbot received mail, which could be disruptive. Therefore, in the new implementation messages are normally not displayed until the user makes an explicit request to interact with his or her userbot. This interaction is supported by a continuously-running "active" userbot, as shown in Fig. 3. The main userbot window indicates the number of outstanding messages waiting to be processed, the userbot's state (working or idle), and three buttons. Clicking on the "process message" button allows the userbot to process messages that require user interaction – for example, bringing up an xchoices window on behalf of the visitorbot. Note, however, that messages that are tagged as "urgent" are always immediately processed by the userbot.

The second button, "user preferences", brings up a window in which the user can set various options in the behavior of his or her userbot. For example, checking the "autopilot" box makes the userbot pop up windows without waiting to be explicitly told to do so. The "voice" checkbox causes the userbot to announce the receipt of new userbot mail using the speaker in a Sun workstation – a kind of audible "biff" for botmail. The "forward to" options are used to indicate that the userbot should choose try to communicate with its owner at a remote location – for example, by transmitting messages via a FAX-modem to the owner's home telephone. (Currently the code to support the "forward to" options has not yet been completed. The exact appearance and functionality of these options may differ in the final version.)

Finally, the third button in the main userbot window brings up the window labeled "taskbots". This window contains a button for each task-specific agent whose email address is known to the userbot. (This information is maintained in file that the user can easily customize.) Clicking on a button in this window initiates communication with the designed task-specific agent, by sending a message of type "help" to that agent. The communication protocol specifies that the agent should respond to a help message by sending back a menu of commands that the agent understands, together with some basic help information, typically in the form of an xchoices message. When the userbot processes this response, it creates a window containing the appropriate controls for interacting with that particular task-specific

---

is due to Mark Jones.

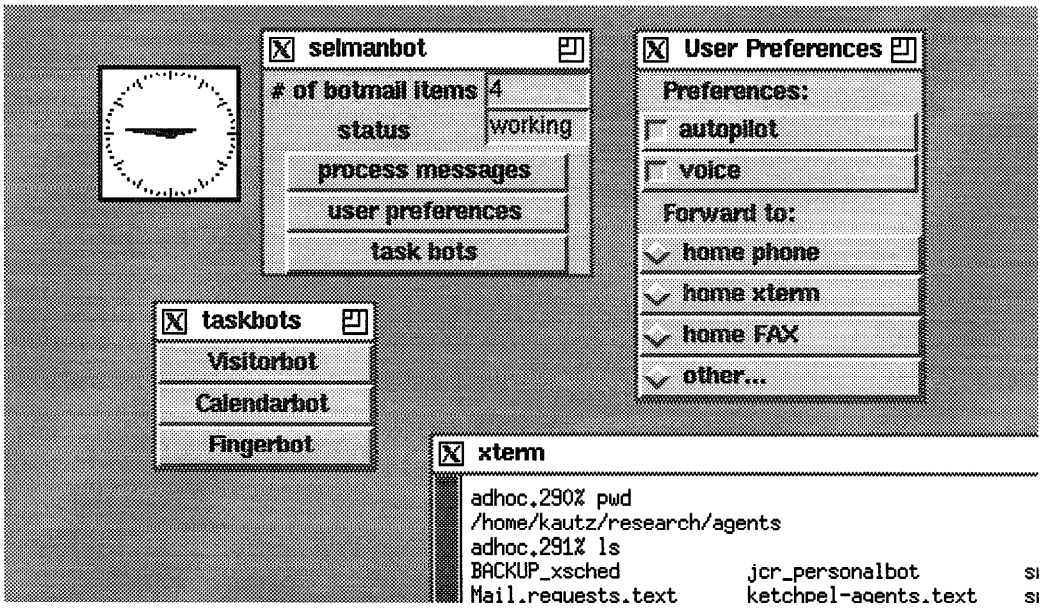

Figure 3: Graphical display of a userbot.

agent. For example, a user who is hosting a visitor to our lab starts the entire process by clicking on the visitorbot button. This leads to the creation of a window containing buttons for basic visitorbot commands, such as scheduling a new visitor, getting the status of a visit, ordering a schedule to be generated, and so on. Clicking on some of these buttons could lead to the creation of other windows, for example, one in which to type the text of the abstract of the visitor's talk.

At the time that this paper is being written, only the visitorbot button in the taskbots menu is active. Over the next few months we intend to establish communication with Oren Etzioni's "finger" agent (used to obtain information about people on the internet) (Etzioni, Lesh, & Segal 1992), and Tom Mitchell's "calendar" agent (used to schedule meetings among groups of people) (Dent *et al.* 1992). The fingerbot and calendarbot will not themselves be ported to our laboratory's computers; instead, those programs will run at their respective "homes" (University of Washington and CMU), and communication with userbots at various sites will take place using ordinary internet email. We hope that the idea of a userbot will provide a powerful and flexible framework for integrating many different kinds of software agents that run in different computing environments.

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 delete automatically 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.

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 to which the bot is not supposed to respond, unlisted home telephone numbers, the user's personal schedule, and so on.

Thus, userbots 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 id.

Earlier we described the alternative Telescript model in which messages are programs that are executed on the receiving machine. This model raises concerns of computer security, particularly if such programs are able to access the host's file system. (Security features in Telescript allow the user to disable file access, but this would appear to limit the kinds of tasks Telescript agents could perform.) Userbot systems are by nature secure, insofar as the routines for processing each message type within the userbot are secure. Although this is a non-trivial condition, it would appear to be easier to guarantee that the code of the userbot itself (that is presumably obtained from a trusted source) is secure, rather than to guarantee that every email program (that could come from anyone) does not contain a virus. Extensions and updates to userbots to handle new message types would have to be distributed through secure channels, perhaps by using cryptographic techniques (Rivest, Shamir, & Adleman 1978).

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.

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 may need general reasoning and planning capabilities, our experiments have shown that interesting agent behavior can already emerge from systems of relatively simple agents.

Conclusions

We have described 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 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 graphical user interface for communicating with users via the userbots. Furthermore, additional modalities of communication, such as speech and FAX, can be added to the userbots, without modifying the task-specific agents.

Perhaps the hardest problem we encountered was defining the initial task. More attention should be paid to identifying useful and compelling agent applications that blend unobtrusively into ordinary work environments. We believe that the empirical approach taken in this paper is essential for guiding research toward the truly central and difficult issues in agent design.

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, L.; Boticario, J.; McDermott, J.; Mitchell, T.; and Zabowski, D. 1992. A personal learning apprentice. In *Proceedings of AAAI-92*, 96–103. AAAI Press/The MIT Press.

Etzioni, O.; Hanks, S.; Weld, D.; Draper, D.; Lesh, N.; and Williamson, M. 1992. An approach to planning with incomplete information. In *Proceedings of KR-92*, 115–125. Morgan Kaufmann.

Etzioni, O.; Lesh, N.; and Segal, R. 1992. Building softbots for UNIX. Technical report, University of Washington, Seattle, WA.

Maes, P., and Kozierok, R. 1993. Learning interface agents. In *Proceedings of AAAI-93*, 459–464. AAAI Press/The MIT Press.

Maes, P., ed. 1993. *Designing Automomous Agents*. MIT/Elsevier.

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, Y. 1993. Agent-oriented programming. *Artificial Intelligence* 60:51–92.