Software Agents for Automating Multiple-Tool Tasks

Chris Toomey and **Ray Johnson**
Lockheed Artificial Intelligence Center
Palo Alto, California
toomey@aic.lockheed.com

Abstract

A key problem in computer-based information processing is the lack of tools addressing the automation of tasks that span multiple application programs. We are seeking to fill this void by developing software agent systems through which end users can delegate such tasks to autonomous software agents. We briefly discuss our approach to several key design issues that arise in building such systems: agent knowledge representation and reasoning techniques, effective human-agent interfaces, interoperation of external tools, and agent knowledge base extension and maintenance.

1 Introduction

The number of proposed roles for software agents in today's and tomorrow's computational environments seems to be almost as great as the number of people that are dreaming about or actively working on them, from intelligent "coworkers" that learn by observing the user at work and eventually begin automating certain user responses [4] to mobile "knowledge robots" that traverse networks in search of information requested directly by digital library users [3]. Though it is hoped that these various types of agents will have common technological needs (e.g., planning, learning, cooperation) that can be used to foster collaborative research efforts, a lot will depend on how people's ideas about the roles of software agents converge.

In this short paper we will discuss the kind of software agents we are developing at the Lockheed AI Center and our approaches to some of the key technological issues that must be addressed in realizing these agents.

2 Problem

While the number and sophistication of software application programs continues to grow at a remarkable rate, higher-level information-processing tasks requiring the capabilities of multiple applications remain frustratingly non-automatable and must therefore be performed manually by users. For example, to write a weekly sales report, one might have to pull sales data out of a database, import it into a spreadsheet to analyze and plot it, and finally employ a word processor to write the report and incorporate the appropriate data and charts from the spreadsheet and database. With the popularity of direct-manipulation application interfaces, even when the task requires use of only a single application, it is still usually the case that the user must sit down in front of the computer and manually operate the application to get the job done.

The performance of these tasks thus requires that users have intimate knowledge about the capabilities and usage of these tools, and that they spend the time to orchestrate the performance of the larger task by manually interacting with the tools. This manual labor requirement becomes a significant issue when these tasks must be frequently performed, for instance by personnel engaged in routine administrative activities and by personnel engaged in providing information products to customers. The tool knowledge requirement becomes a significant issue when new tasks arise and when new personnel come on-board.

There are two major barriers to providing cost-effective "task-level" information-processing automation (i.e., the end-to-end automation of user information-processing tasks) in today's computer environments. The first is simply the lack of powerful, meta-level tools capable of eliciting these high-level requests from users, determining an appropriate orchestration of external applications to satisfy such requests, and finally fulfilling the request by tasking the applications. As we will describe in this paper, this is the void that we feel can be filled by software agents.

The second barrier to automation is the lack of programmatic control availability for many software applications, especially those with direct-manipulation user interfaces. Though we obviously do not have direct control over this problem, we will touch on several developments that give hope that this barrier is starting to come down.

3 Software Agents at Lockheed

The particular problem we're addressing with our software agent program is the task-level automation one described above. Namely, we wish to build systems with which users can create and deploy software agents to autonomously perform tasks spanning multiple software applications. Thus we view the relationship between user and agent as being most similar to that between a manager and secretary, in that the manager explicitly tasks the secretary to perform tasks that would otherwise waste valuable cycles that the manager could better employ on more demanding activities. The metaphor is also apt in that we envision this type of agent being most useful when coming

pre-equipped with knowledge and capabilities, so that, like a secretary, it could perform tasks that the user (manager) either doesn't perform already or that are too unconstrained to be learned by watching the user (manager) perform them.

An example of the kind of task that we would like users to easily be able to delegate to a software agent, and for which we have built a prototype software agent system, is as follows. Many Lockheed managers and business-development personnel are actively engaged in monitoring the *Commerce Business Daily* (CBD), which the government uses to announce new procurements and awards made for previous procurements. This task requires users to 1) access and search the electronic CBD database for relevant announcements, 2) extract or download the relevant announcements from the database, 3) process the announcements in various ways (e.g., by formatting and breaking out embedded lists or tracking down the matching procurement announcements for award announcements) using an editor or word processor, and 4) distribute the processed announcements via the desired medium (e.g., e-mail, fax, or hardcopy) to appropriate coworkers, potential teammates, etc.

The overall goals of our software agent project are to make software agents that can handle such tasks technically feasible, palatable to end users, and cost-effective to build and maintain. The key issues we've identified that must be adequately addressed in order to achieve the above goals are 1) providing sufficiently expressive languages with which to represent user task requests and domain knowledge, 2) endowing agents with sufficient reasoning mechanisms to transform user requests into appropriate actions which fulfill them, 3) providing a sufficiently intuitive and flexible human-agent interfaces to enable unsophisticated users to easily convey their task requests, 4) developing adequate means for agents to interoperate other software applications, and 5) designing an open system architecture and support facilities to enable cost-effective maintenance and extension of agent capabilities.

3.1 Agent System Architecture

The agent system architecture we've developed for the Apple Macintosh provides an operating-system-like environment for software agents. It enables users to create, task, and monitor a personal collection of agents, and it provides the infrastructure within which the agents live and interact with the outside world. The system consists of two executables, an agent server process that runs as a background application and in which the agents execute, and a human-agent interface process with which the user manages their agent set and tasks the new agents they create. This two-piece architecture gives users the benefit of a high-level graphical user interface with which to define and periodically monitor their sets of agents, yet allows the defined agents to operate autonomously in the background within a lean, interface-less application.

The agent server module has an object-oriented design, with classes representing different types of agents, instances representing instantiated agents of a given type, and message-passing serving as the communication mechanism and also as the triggering mechanism for agent processing (the agents sleep after processing a message until the next message arrives). Associated with each agent class is a body of domain knowledge that encodes what the agent knows how to do (described in the next section) and a custom graphical user interface designed to assist the user in specifying tasks for that class of agents to perform.

Our original motivation for agent classes was to be able to create coarse, task-oriented classes for different user groups (e.g., a business information filtering class and a computer system administration class), with each class having knowledge about the all the tools needed to solve the user group's tasks and having an interface customized to task area and user group. For the CBD-processing software agent application, for example, most of the knowledge is stored in a single CBD-agent class (though some is inherited from the higher-level classes from which it originates). The CBD-agent class includes knowledge about accessing the CBD via WAIS, knowledge about how to create and delete files and directories, knowledge about sending documents via electronic mail, etc. This agent application also has a customized GUI that provides widgets through which users specify their CBD-processing requests, e.g., article keywords to look for and announcement delivery preferences.

While this "monolithic agent" approach facilitated the rapid development of our CBD-processing agent system, it has some fairly obvious modularity and scale-up shortcomings, which we are addressing in our current work on satellite data dissemination agents. Specifically, we are investigating a more modular, distributed-expertise approach in which a collection of specialist agents (e.g., one that provides information-retrieval services, another that provides electronic mail delivery services) are tasked as needed by "broker" agents servicing specific end-user requests. In addition to using broker agents to compose the capabilities needed to solve a particular task, we are also investigating the dynamic composition of task-specification user interfaces from capability-specific GUI components associated with the specialist agents (see Section 3.3).

3.2 Knowledge Representation and Reasoning

To realize the type of agents we are developing, namely explicitly tasked agents fulfilling user requests by dynamically composing sequences of appropriate actions, we have adopted a classic goal-driven, backward-chaining control strategy. User requests are represented by groups of immediate goals and *goal triggers*, which are data-driven rules that assert sets of immediate goals to be pursued in response to specified events (similar in spirit to the exogenous event handling scheme in Etzioni's UNIX softbot architecture [1]). Goal triggers enable agents to automate the kinds of tasks that users would most like to automate, namely those involving repetitive information-processing duties performed either at regular inter-

vals or in response to selected events. For example, goal triggers are used in our CBD-processing agents to specify requests such as performing daily WAIS searches of the CBD database and delivery of new CBD articles via electronic mail. Agent class-level goal triggers are also defined in order to provide the equivalent of object-oriented message-handling "methods:" messages invoke goal triggers which assert appropriate message-handling goals to drive the agent's response.

For our experimental CBD-processing agent system, we chose to build a simple rule-based reasoner in CLIPS rather than use a full-blown AI planner. Our motivations for this decision were that our search space branching factor was essentially one (each goal is addressed by a single operator), we wanted a C-based implementation, and we wanted to deliver the application quickly. We constructed a domain-specific planner/executer by using two types of rules: goal-decomposition rules provide a backward chaining effect (when used in the CLIPS forward-chaining inference engine) by matching on higher-level goals and asserting appropriate lower-level subgoals. These rules decompose the original goals into a set of primitive goals that can be satisfied by performing some action (e.g., sending a query to WAIS or tasking an e-mail server to deliver an article).

The second type of rule we used was the primitive action execution rule, which triggers off of primitive goals and executes the appropriate action(s). The execution order of the actions resulting from such inferencing is determined by the order in which the original goals and subsequent subgoals were asserted, as CLIPS uses a depth-first inferencing strategy. We used this knowledge to impose proper orderings on the goal and subgoal assertions to ensure that actions would fire in the proper order and thus not undo one another's effects.

In retrospect, we feel that the general knowledge representation and reasoning techniques we have used (goals and triggers for task specification, declarative action representation, and goal-driven reasoning) are the appropriate ones for building this type of explicitly tasked software agent. However, our rule-based implementation lacks the power, generality, and declarativeness of a more formal planning system, and thus presents maintenance and scalability difficulties. We are thus currently engaged in an effort to acquire a more formal reactive planning system for use in our next-generation agents.

3.3 Human-Agent Interface

As anyone who has ever demonstrated or delivered software to "real-world" users knows, the key to success is usually the user interface. The interface is especially crucial for enabling unsophisticated end users to do something as complex as specifying a task request to a software agent via a set of goals and goal triggers. On the other hand, given the historically high cost of developing sophisticated graphical user interfaces, techniques must be acquired or developed with which advanced GUIs can be produced at reasonable cost. A software agent development and management system actually requires three types of human-agent interfaces: the task-specification interface mentioned above, an interface for managing the collection of agents (e.g., creating new or duplicating existing agents, deleting agents), and an interface for receiving the results of the agents' processing. In this section we describe our approach to balancing efficacy and development cost for the most costly and complex of these, the task-specification interface.

While writing symbolic goal declarations directly might be an acceptable means for AI researchers to task their software agents, it is not a viable means for a wider community of end users to task their agents. Thus, one of our thrusts in building the CBD-processing agent application was to develop a sufficiently high-level GUI through which unsophisticated users could task their software agents. Our approach was to build a highly customized GUI having one or more widgets dedicated to each and every goal and goal trigger defined for the CBD domain. Users could thus select the goals and goal triggers they desired for a given CBD-processing task by checking off boxes, which would in turn activate additional widgets through which they would specify any associated parameters (e.g., the frequency and mode of article delivery). The resulting GUI inputs were then translated into appropriate goals and goal-triggers, in part by the GUI in constructing the task-specification messages to be sent to the agent and in part by the agent in processing those messages.

We have found this CBD-processing task-specification interface to enable our non-technical users to appropriately task their agents, and our previous user interface experience leads us to believe that this type of high-level task-specification interface will work for other agent applications. However, the cost of developing such specific, "hard-coded" interfaces has led us to begin investigating ways to deliver this kind of functionality with a much lower development cost. The primary cost-cutting approach we're investigating takes advantage of the fact that the task specification process involves users repeatedly 1) selecting a goal or goal trigger (e.g., find articles) and 2) specifying values for any parameters associated with the selected goal or trigger (e.g., find articles containing the keywords *software* and *agents* and occurring in the CBD database). We are thus looking to develop a generic task-specification interface that more closely mirrors this approach by querying the agent for the available goals and triggers and dynamically presenting the relevant interface components upon their selection by the user. Such an interface would automatically provide access to new agent capabilities as they are added to the system.

3.4 External Tool Interoperation

In order for agents to be able to automate the computer-based information-processing tasks that users would otherwise perform, they must be able to access and control the same tools a human user may control. Unfortunately, it is often difficult or even impossible to control some applications in a program-

matic way. Furthermore, the method by which an agent may control a tool or service is often wildly different between tools. These difficulties essentially limit the power of software agents by restricting the tools and services the agent may access. This section briefly discusses what we have done and what must still be done to make software agents what we call "First Class Citizens" [2].

The graphical user interface (GUI) revolution made applications easier to use for humans. At the same time, however, most of these new applications provided no way to control the application in a programmatic way. In order for a software agent to control such an application, it must either simulate key clicks and mouse movements or find some other method to fool the application into thinking a user was actually controlling the application. This is at best a difficult, if not impossible, task that would only work for certain applications.

Unfortunately, even the tools and services that are directly programmable utilize different methods of control, ranging from simple command line arguments to proprietary built-in scripting languages. Many require specific communications protocols such UNIX TCP/IP sockets or Macintosh AppleEvents. These two problems essentially make agents "Second Class Citizens" by preventing agent control of certain applications, either because programmatic control is impossible altogether, is restricted to a subset of tool functionality, or is too costly to utilize.

The long-term solution to such problems is for all application programs to 1) support a standard programmatic control interface, and 2) to provide complete access to their capabilities through such an interface. Such a solution would need to be driven both by encouragement from the platform provider and from market forces. In fact, movement towards such a seemingly far off solution has been taking place on the Apple Macintosh and PC platforms. The System 7 operating system introduced AppleEvents, which enable applications to send one another low-level commands and which Apple has been encouraging application developers to support. Apple has also recently introduced AppleScript, which enables interoperation of applications through a higher-level scripting language. HP's NewWave, an extension of Microsoft Windows, goes even further by mandating an application architecture supporting complete programmatic control of applications.

In the near term before such solutions become fully established, our agents use a platform-independent scripting language known as Tool Command Language (TCL) [5] to communicate with external tools (an AppleEvent exists to send a script to another application, and TCL provides an equivalent communication mechanism on UNIX). In order for our agents to interoperate non-TCL-compliant compliant tools, we build TCL-compliant "wrappers" that translate the agents' TCL commands into equivalent commands understood by the applications.

3.5 Agent Knowledge Extension and Maintenance

There are two dimensions along which a system's knowledge-base extension efficiency can be measured: (1) the support the system provides for acquiring new knowledge, and (2) the extent of downstream system modifications that must be made to enable access to and use of the new knowledge. Systems like Maes' learning interface agents [4] provide fully automated support along both dimensions, but only with respect to the pre-selected task addressed by the system (e.g., scheduling meetings). On the other hand, a declarative, planning-oriented system like Etzioni's UNIX softbots [1] may not provide any support for knowledge acquisition, but could presumably employ new hand-coded knowledge about other UNIX tools to perform entirely new information-processing tasks without further modification.

In our efforts to date we have focused on the second of the above dimensions. We have designed our agent system architecture to enable the modular addition of new agent capabilities without requiring additional system modifications before agents can begin utilizing these new capabilities. Two system design elements discussed above enable this modular-extension ability. First, we have selected a declarative, planning-operator knowledge representation and corresponding inferencing mechanism that enable new capabilities to be "added to the soup" and utilized as necessary by the agents to satisfy user requests. Second, we are in the process of developing a flexible, generic task-specification interface that would immediately enable users to request new agent capabilities after they are added to the knowledge base and their goal/trigger parameter-specification interfaces are created.

References

[1] Oren Etzioni, Neal Lesh, and Richard Segal. Building softbots for unix. Technical report, Department of Computer Science and Engineering, University of Washington, 1992.

[2] Raymond W. Johnson. Autonomous knowledge agents. In *Tcl/Tk Workshop Proceedings*. University of California, Berkeley, June 1993.

[3] Robert E. Kahn and Vinton G. Cerf. *The Digital Library Project*, volume 1: The World of Knowbots. Corporation for National Research Initiatives, 1988.

[4] Pattie Maes and Robyn Kozierok. Learning interface agents. In *Proceedings of AAAI-93*, 1993.

[5] John K. Ousterhout. Tcl: An embeddable command language. In *Proceedings of USENIX 1990*, 1990.