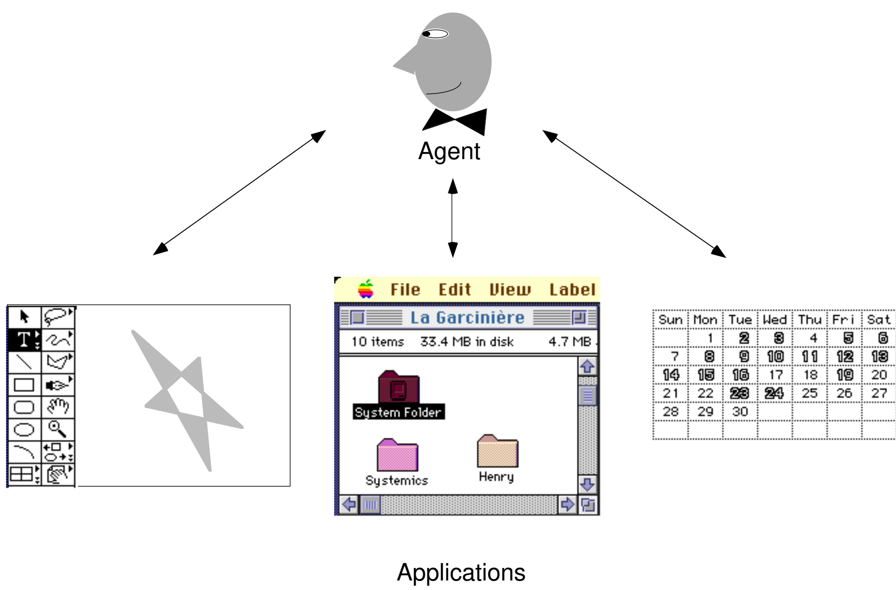
Proceedings of the ACM Conference on Intelligent User Interfaces, San Francisco, January 1998

Integrating User Interface Agents with Conventional Applications

Henry Lieberman
Media Laboratory
Massachusetts Institute of Technology
Cambridge, MA 02139 USA
(1-617) 253-0315
lieber@media.mit.edu



ABSTRACT
In most experiments with user interface agents to date, it has been necessary either to implement both the agent and the application from scratch, or to modify the code of an existing application to enable the necessary communication. Instead, we would like to be able to "attach" an agent to an existing application, while requiring only a minimum of advance planning on the part of the application developer. Commercial applications are increasingly supporting the use of "application programmers' interfaces" and scripting languages as mean of achieving external control of applications. Are these mechanisms sufficient for software agents to achieve communication with applications?

This paper reports some preliminary experiments in developing agent software that works with existing, unmodified commercial applications and agents that work across multiple applications. We describe a programming by example agent, *ScriptAgent*, that uses a scripting language, Applescript, to record example procedures that are generalized by the agent. Another approach is *examinability*, where the application grants to the agent the right to examine internal data structures. We present another kind of learning agent, Tatlin, that compares successive application states to infer interface operations. Finally, we discuss broader systems issues such as parallelism, interface sharing between agent and application, and access to objects.

KEYWORDS: Agents, scripting languages, programming by example, programming by example, programming by demonstration, machine learning, user interface.

FROM "APPLICATIONS" TO AGENT ENVIRONMENTS

The dominant paradigm in commercial software for personal computers has been that of a so-called "application", a self-contained, shrink-wrapped product bought from a single supplier to do a well-defined task. The user of a computer chooses among the available applications on the machine, "enters" an application, does some focused work on a set of documents manipulated by that application, then "leaves" the application and may enter a new application. Each application has its own interface, and moving between applications means interacting with a different [but possibly overlapping] interface. Recent operating systems have improved upon this only slightly by allowing several applications to be "open" at once, or to let a document produced by one application be "contained" in a document produced by another application [Open Doc, OLE].

The paradigm of thinking of software in terms of self-contained and isolated applications or documents is becoming rapidly obsolete. Computing environments are getting more complex, and users are getting tired of the artificial barrier between applications. Users want to work with text, graphics, communications, programming, etc. seamlessly in an integrated environment. Agents can be seen as a way of supplying software that acts as the representative of the user's goals in the application environment. Agent software can provide glue between the applications, freeing the user from the complexity of dealing with the separate application environments.

This paper reports on several experiments to use current commercial inter-application communication mechanisms to implement application-independent agents. We focus on agents that learn from observing user actions in the interface and produce generalized procedures capable of automating tasks for the user. These communication mechanisms fall short of full agent-application cooperation, but have supported some interesting experiments. We present these results, not as the definitive solution to these problems, but as lessons from experience that will help those grappling with similar problems.

The first, *ScriptAgent*, uses a so-called *scripting* language to record procedures and control applications. The second, *Tatlin*, uses a concept we call *examinability*, to learn from comparing successive states of the application. The point is not to provide complete descriptions of these systems, but to focus on the aspect of how these systems communicate with their target applications.

The rest of the paper surveys conceptual issues in agent-application communication. These divide into systems-oriented issues, such as access to objects and parallelism, and more high-level issues such as the sharing of the user interface between the agent and the application, how the agent stores knowledge about the application, and the division of problem-solving effort between the agent and the application in solving the user's task. Complete answers to this higher level issues remain an open research question.

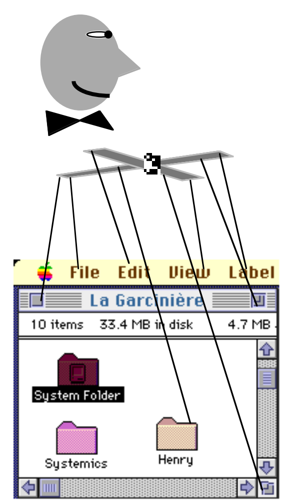
ATTACHING AGENTS THROUGH THE USER INTERFACE: "MARIONETTE STRINGS"

There are efforts now underway in the commercial arena at putting together some sort of inter-application communication. The intent of these efforts is primarily to support inter-application data-sharing, scripting [manually writing external programs to control an application], and simple macro-recording. However, developers of such languages also envision that they might serve as support for future interface agent software as well. Examples of such efforts include AppleEvents and AppleScript, Microsoft OLE, Active X, and Visual Basic, TCL, Java, Javascript, and others.

Many applications such as Excel, Emacs, Director, Netscape, and others have an embedded programming language in the application. Such languages have primitives that allow direct access to the objects and functions of the application, but unfortunately, do not provide a direct way of interacting with other such applications.

Scripting languages like AppleScript and Visual Basic take advantage of the following observation. Although most commercial applications are not written so that their functionality is accessible by another program, they must provide some way for the user to access that functionality through input, typically by selecting some item from a menu, clicking on some icon, or typing some command. Thus, if another program can "fool" the application into thinking that the user typed or clicked something, it too could access the functionality of the application.

I call this approach *"marionette strings"* because the agent is given a set of "strings" corresponding one-to-one with user actions in the interface, and can "tug" on the strings to make the program perform.



"Marionette strings"

Regardless of programming language or implementation, most modern direct manipulation interfaces are structured around an interpreter called an "event loop" that accepts mouse and keyboard input, tries to determine what functionality the input is requesting, changes the application data structures, and updates the screen display. Many UIMS systems supply at least a rudimentary form of this interpreter. The marionette strings are implemented by stuffing an object representing user input into the application's input buffer. The application then processes it "as if" the input was produced by user interaction.

This works, after a fashion. Because there is some degree of standardization in interfaces for invocation of user-level commands, such as pull-down menus, or double-clicking on icons, it is easier to get different applications to agree on a standard format for input than for how to invoke the underlying functions. [If there were no interface standardization, the users would rebel!]

But it is less effective for returning values from operations, for accessing application data. From the user's point of view, the result of the operation is simply visible in the graphical interface. An agent program cannot "see" what the user sees [though see [Potter 93] for an approach that parses the screen display]. Other operations that are necessary for a program but do not have direct correlates in the interface are also difficult.

One approach is to provide more marionette strings by permitting "virtual events" [e.g. Kosbie 93] that have no direct counterpart in the interface, but are treated as events in the input buffer. This introduces some interpretation overhead, and may complicate the operation of the interpreter.

SCRIPTAGENT: A LEARNING AGENT BASED ON A SCRIPTING LANGUAGE

As an example of how we might use the marionette string approach to implement an agent, we implemented ScriptAgent, a programming by example system based on Apple's AppleScript inter-application scripting language. ScriptAgent uses techniques from the author's Mondrian system [Lieberman 93], which learns procedures in a graphical editor. ScriptAgent's initial domain is Apple's Scriptable Finder, but it is targeted for use with any AppleScript scriptable and recordable application.

ScriptAgent is one of the few attempts to build a learning agent that works with standard operating system software and unmodified commercial applications. This should greatly extend the scope and practicality of agent learning systems. ScriptAgent's use of AppleScript also enables it to be one of the few agent systems that could operate across multiple applications. Macro recorders like Apple Macromaker or CE Software's Quickeys do provide recording and replay of unmodified input events across applications, but no generalization of the recorded procedures. ScriptAgent also serves as one of the first serious tests of AppleScript's original intention to provide monitoring of user interface actions and effective control of applications by an external agent.

INTEGRATING THE AGENT"S INTERFACE WITH THE APPLICATION'S INTERFACE

For the Finder, we have tried to integrate the agent's interface into the Finder's interface itself. We use AppleScript-generated applications to represent the interface with the agent. Procedures recorded by the agent are stored as AppleScript applications that are invoked by double-clicking or dragging as other applications are. ScriptAgent's generalization operation is represented by an application Make Example.

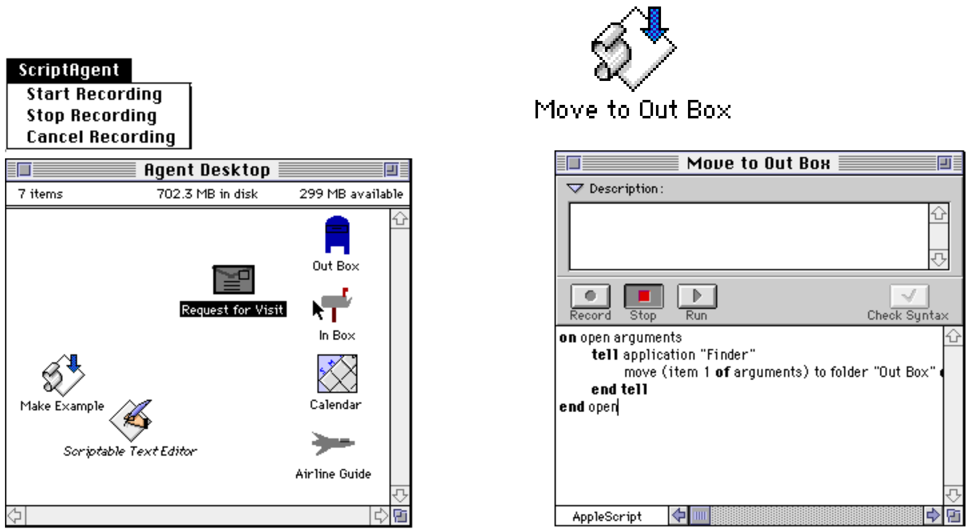
Dragging a Finder object onto the Make Example icon says that that object is to be thought of as an example for the procedure that is being taught to the system. All subsequent user operations are recorded as relative to that example. If the example object is used as an argument to further operations, then those operations will depend upon the example as well, and the next time the procedure is executed, the new objects will be substituted.

An example

As a very simple example, we show a simple file manipulation procedure in the Finder. We pick a specific file named "Request for Visit" to serve as an example. We will demonstrate the procedure on "Request for Visit" and the system will generate a procedure that can be applied to any file. We turn on recording, drop the file onto the Make Example application, then demonstrate the procedure. On the right of the illustration is the AppleScript code as generalized by ScriptAgent. This generates an AppleScript application which, when a new file is dropped on it, performs the analogous Move and Duplicate actions.

The Make Example operation is duly recorded in the AppleScript code recorded by the Scriptable Finder, and is therefore integrated into the procedure being recorded. It is noticed by ScriptAgent's generalization code, which designates the argument to Make Example as a generalization.

Scriptability



ScriptAgent: The Finder interface [left] and generated application and script [right]

The term *scriptable* [following Apple terminology] is used if the application provides a means [either through a scripting language or through a so-called *application program interface* [API]] for an external agent to invoke the commands of the application. To be fully scriptable, the application must allow the external program to invoke *any* function that the user can initiate by selecting from menus, clicking on icons, or typing. An application will be called *recordable* if it is capable of reporting to an external agent when the user asks the application to perform a function, by menu or icon selections, or by typing.

Making an application fully scriptable and recordable is a very strict requirement. Several years after Apple introduced AppleScript scripting, and asked application developers to comply, sadly, few developers have done so. Many applications are at least partially scriptable, but few provide either a complete scripting interface or a usable recording capability. The situation on other platforms is similar. A commercial interface agent using neural network learning techniques [Caglayan, et. al. 96] has had to resort to patching system routines in order to observe and affect user interface actions, and was ultimately unsuccessful. However, we do hope that the number of scriptable and recordable applications will increase, and the existence of intelligent interface agents will certainly increase the incentive for developers to provide external control. Though "scriptable" is in its name, it is actually recordability which is of more interest for programming by example, and is the reason why ScriptAgent operates in the Finder's domain.

Enabling the agent to work with AppleScript

Understanding and generalizing the AppleScript code as it comes from the Scriptable Finder is done via a parser and unparser from AppleScript to CLOS objects, using Joachim Laubsch's Zebu [Laubsch 92], a LALR parser compiler.

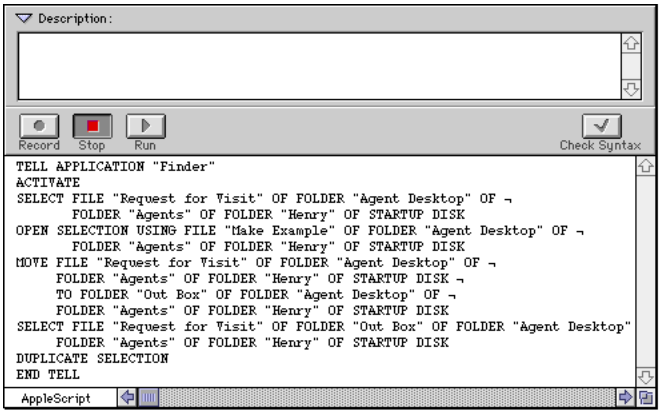
The approach of reusable parsing/unparsing technology is important. We are entering an era where the computer environment will have many co-existing scripting languages or other text-based application specific languages. AppleScript, Java, JavaScript, TeleScript, Visual Basic, and TCL, are all examples of this trend. No one of these languages is computationally general enough to single-handedly support the kinds of agent applications we are envisioning, and agent programs may have to deal with code written in several of them. Parsing the various representations into a single dynamic environment suited for symbolic computation seems like a strategy that will cope with a multi-language world.

ScriptAgent, like many agent programs, is a program-writing program, so must be able to analyze, generate and execute data representing program code. Generation of code for agent actions is accomplished by methods that act as code walkers or partial meta-interpreters on the parsed representations. For ScriptAgent, these methods do the explanation-based generalization of recorded procedures.

Describing actions and objects

The basic mechanisms for recording and generalizing procedures in ScriptAgent are taken from Mondrian [Lieberman 93] and consist of an explanation-based learning mechanism that generalizes on user-designated example objects, and propagates generalizations through structures that record the dependencies between operations.

The illustration below shows the user interface actions as originally recorded by AppleScript's default action recorder. Note that specific file paths appear where ScriptAgent had generalized an argument to the resulting application.



How the user interface recording mechanism describes objects involved in user interface operations is known in the field of programming by example as the *data description problem* [Cypher 93]. This problem is central to how the agent and the application will cooperate. Different choices in how to describe objects will result the agent having different ideas about "what the user did".

In AppleScript, objects are described using AppleScript expressions which designate a "path" to the object of interest, e.g. "Window 2 of Document 1". The choice of describing the object in this way [as opposed to, say, the window named "Foo", or the last window the user selected] is made by the recording mechanism rather than by the agent. Nothing in the agent can affect the way this choice is made.

Unfortunately, AppleScript does not give the agent direct access to the objects that are the arguments to and the results of operations themselves. Were it to do so, the agent could construct its own data description of the object, as occurs in most programming by example systems [Cypher 93]. However, AppleScript cannot do this, because it does not require the application to provide a full object model for all its internal data. The applications may be hard coded C programs which may not be able to deliver pointers to particular internal data structures. If the user selects a word in a word processor, the word processor application may or may not be able to return "Word 3 of Line 20 of Document 2" as a value that can be relied on in subsequent operations. So-called AppleScript Objects are not required to have a lifetime beyond a single AppleEvent transaction.

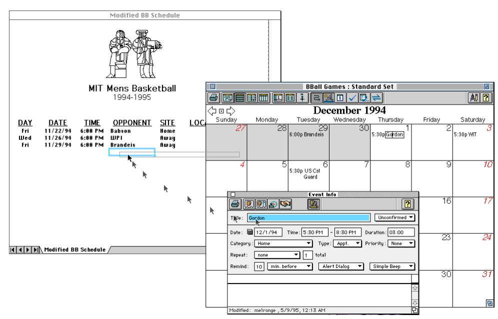
For the Scriptable Finder, ScriptAgent deals with this problem by using the file system to dereference expressions, since all the objects in the Scriptable Finder's domain [files, applications, etc.] are accessible directly to ScriptAgent through the file system. However, for an arbitrary application, it may be necessary for the agent to keep "shadow" objects to model the application's data. In the worst case, the agent may need to mimic the application's functionality in order to access the result of an operation.

EXAMINABILITY

Because scriptability is so much more prevalent than recordability, we are also exploring an alternative approach which we call *examinability* to allow an agent to operate when full recordability is not available. We notice, that as part of scriptability, many applications do give an external agent the right to examine internal data structures. Thus we can have an agent poll the application data structures and try to infer the user interface actions by comparison of successive application states.

This approach is taken in Tatlin, a system by my student David Gaxiola [Gaxiola 95], which integrates a commercial spreadsheet [Microsoft Excel] with a calendar program [Now Up-to-Date]. The user interactively transfers an example entry from the calendar to the spreadsheet, and the agent learns a procedure that can perform an analogous transformation on similar entries, using similarity-based learning.

In this scenario, a user has a series of sports events noted in a calendar, and wishes to construct a spreadsheet summarizing information about the events. The user demonstrates a single example of how to transfer the information from an appointment in the calendar to appropriate columns of the spreadsheet, using Cut and Paste operations. The agent is then asked to generalize a program that can be given a set of events in the calendar, and have them automatically transferred to the spreadsheet.

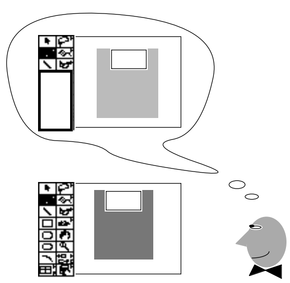


The agent examines the internal state of the calendar application, starting from the appointment window currently open and selected. Tatlin has an object model for the calendar application, and reads the various fields of the appointment data structure. The user must then select the destination data structure in the spreadsheet [in this case, a row] and Tatlin tries to match up the fields. Tatlin can accept advice to pay attention [or not] to certain attributes of data, such as fonts, case, or numeric format.

Tatlin assumes that fields that have similar names or contents are causally connected by user interface actions, such as cutting the Date field from the event and placing it in the Date column of the spreadsheet entry. Such an assumption can be erroneous, of course, but the user is assumed to be providing "good" examples that do not have collisions, in order to clearly teach the agent. A practical problem that arises in such systems is making these inferences robust in the face of minor details such as differences in time and date formats, and differences in terminology between applications [one may refer to a document's "Title" while another may refer to its "Name"].

AN AGENT"S "MENTAL MODEL" OF THE APPLICATION

Both ScriptAgent and Tatlin bring up the issue that it is necessary for the agent to have an object-oriented model of the data in the application it is dealing with. Since the agent is supposed to act as a proxy for the user, one could think of this model as the agent's "mental model" of the application. As in a user's mental model of the application, the model need not consist of all the objects in the application, but at least those that are directly visible and of concern to the user for specific purposes in the interface, and the minimal set of internal objects that affect the application.



A problem for programming an agent is that the objects internal to the application are not accessible directly to the agent. The agent must thus deal with *foreign objects*. Foreign objects may be written in a language different from the language of the agent, they may be in another process, or they may be stored on another machine on a network. Several proposals are currently under consideration for how Java might interact with foreign objects stored in other locations on the Web.

We have implemented a *foreign object interface* in CLOS [Common Lisp Object System]. This is a facility that allows a calling program to access foreign objects more or less transparently, as if the objects were local to the calling program. It works by keeping a table of objects in each local program that are referenced by foreigners, managing that table on an as-needed basis. The foreign object interface is intended to be complementary to so-called *foreign function interfaces*, such as remote procedure calls or inter-language function calls. Some foreign function interfaces do provide translation of basic data types such as numbers and strings, but complex objects that may point to other objects are not typically provided for. Leaving object management up to the applications programmer is typically too great a burden, and effectively discourages routine use of multi-language or multi-process programs. The return value of each foreign procedure call is a *foreign object,* a representative of an object in another application. Functions applied to the foreign object, and requests to access the components of the foreign object result in forwarding the request to the server application.

It is important that foreign objects are created in a lazy or on-demand manner. The client program need not have a foreign object corresponding to every object in the server. Foreign objects for compound objects on the server are created on the client only to the depth to which they are referenced, and not in their entirety. This property is essential for making the foreign object interface efficient enough to be practical. It is also important that access to the object be transparent. If it is not, the caller must be prepared to deal with the possibility that any object might, in fact, be foreign, which complicates calling interfaces.

A request to create a new object on the server is handled by a process analogous to what is called *interning* in interpretive systems. Interning is a process that translates strings into references to objects. It involves keeping a table of objects, usually in the form of a hash table. The first time a string is encountered, a new object is created and stored in the table. The next time a string containing the same sequence of characters is encountered, the original object corresponding to that string will be returned.

Each server maintains a *registry*, a table of objects that are referenced from outside. When a server receives a request to create a new object, it makes a new entry into the registry, and returns an index into the registry. The foreign object on the client contains the server connection and the index into the registry. The metaobject protocol [Kiczales and Bobrow 92] can be used to achieve transparency between the use of foreign and local objects. The system exchanges objects with a Lisp running on another machine using the Apple Events [Apple 93] inter-application communication protocol. We have not dealt with the problems of garbage collection across multiple address spaces; once an object is entered into the registry it remains forever. There are several proposals for algorithms for network garbage collection in the literature. Inter-language garbage collection, also desirable in the general case, is also not handled in our implementation.

ISSUES IN AGENT-APPLICATION INTEGRATION

Granularity of the event protocol

For the marionette string approach to work, the application writer must agree on a set of operations to "export" as events, which determines what is accessible to the agent, and the agent must agree to "import" the events. This is especially important in approaches like Kosbie's [Kosbie 93], where virtual events are created that do not correspond directly to explicitly visible user interface actions.

There is the problem of deciding at how fine a level to do this. In the limiting case, every time the program does *anything*, it must create an event corresponding to that action, and be prepared to execute that action in response to an event requesting it from the agent. In that case, the application effectively becomes a program written in the event language rather than the underlying programming language. The "application code" itself becomes nothing but an interpreter for the code written in the event language. At worst, the interpretation overhead would slow the application, and at best this would require two versions of everything, an interpreted and a compiled version.

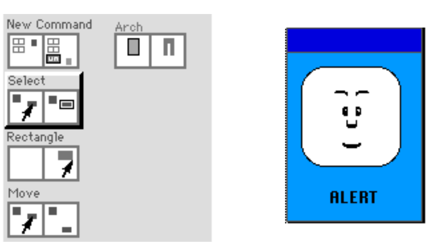
To take a concrete example, should we include mouse movements in the protocol that an agent should be able to receive? If we say yes, then we are obligating every application to report every time the mouse moves, which is potentially inefficient, and wasteful in the case the agent does not need this information. If we say no, we are preventing all agents from ever tracking the mouse, and it seems like at least some agent might in fact want this ability.

The moral is that there should be at least some provision for dynamic negotiation of a protocol between agent and application. It may not be possible to fix in advance an application protocol that will be satisfactory to every possible agent, nor an agent protocol that will be acceptable to every application.

Sharing an interface between agent and application

Another issue is that the agent itself may need to interface directly with the user, and most applications are not currently prepared to *share* their interface with another program whose interface they do not know in advance.

Mondrian [Lieberman 93], for example, adds icons to the draw program's palette, using the same "domino" icon style in which the draw program's operations are represented. In the illustration below, the New Command icon at the upper left is an icon that operates the agent, the three icons below it are draw operations provided by the draw program, and the Arch icon at upper right was added by the agent to the interface as a result of user interaction.



Left: Application icons [left column]
and Agent-generated Arch icon
Right: Anthropomorphized calendar agent

Kozierok's Calendar Agent [Maes and Kozierok 93] displays a cartoon face whose expression indicates the state of the agent and feedback about its predictions. Cypher's Eager [Cypher 93] has an anthropomorphic cat to represent the agent, and colors menu operations for anticipatory feedback.

Applications should be able to accept requests to extend their interface by adding or modifying user-accessible commands and objects, reserving parts of the screen or modifying the program's user interactions, as part of the protocol. Such requests are not typically a part of inter-application communication protocols.

Parallelism between the agent and application

Parallelism is also an issue. Any user interface program is in fact a parallel program, even if there is no parallelism going on in the functions implemented by the application. This is because there are always at least two processes that are running simultaneously: the computer and the user!

This is a well-known problem in user interface programming, but it becomes worse when the actions of an autonomous interface agent are added to the system. The application and agent program may be acting concurrently on the same objects, and synchronization must be provided for where necessary. Another merit of the marionette string approach is that it achieves cheap synchronization for programs that are completely "event driven": only one event may be processed at a time. For more complex kinds of programs, such as those which take background action while the user continues to use the interface, this kind of synchronization will be inadequate.

RELATED WORK

There are an increasing number of projects in agent software that are trying to integrate with more traditional applications, though there is still no definitive methodology for this. [Newell and Steier 93] is one of the few references that systematically discusses these issues.[Rich and Sidner 97] provide a good discussion of user/agent collaboration issues, including initiative, dialogue, and turn-taking. [Piernot and Yvon 93], [Kosbie and Myers 93] and [Frank 96] present proposals for application-independent agent kits, based on histories of user-interface events. Some interface agents, such as [Lansky et. al. 95] and [Etzioni 94] are relatively loosely coupled to applications with command-oriented or conversational interfaces, and don't require the tight agent-application coupling discussed here. All of these references depend upon some explicit cooperation from the target applications.

A related approach to Tatlin underlies Robo-Format [Ash and Schlimmer 95]. Robo-Format used examinability of Excel to automatically compile formatting templates for spreadsheet cells. Earlier, Myers and Werth's Tourmaline [Myers and Werth 93] used the scripting language WordBasic to generalize formatting templates for text.

ACKNOWLEDGMENTS

Thanks to Pattie Maes for her insights concerning intelligent agents. Max Metral has provided valuable insights into application/agent communication. Thanks to Jim Miller, Tom Bonura, Allen Cypher, Ike Nassi and Steve Strassman.

All names of commercial products mentioned herein are trademarked by their respective manufacturers.

Support for this work comes from research grants from Apple Computer, the National Science Foundation, British Telecom, Exol, the News in the Future Consortium. the Digital Life Consortium, and other sponsors of the MIT Media Laboratory.

REFERENCES

Ash, J., and J. Schlimmer, Robo-Format: A Sample Self-Customizing Application, Washington State University [ftp: //ftp.eecs.wsu.edu/ papers/ schlimmer/ash-robo.ps], 1995.

Caglayan, A., M. Snorrason, J. Jacoby, J. Mazzu and R. Jones, Lessons from Open Sesame! a User Interface Learning Agent, Conference on Practical Applications of Agents and Multi-Agent Systems [PAAM-96], London, 1996.

Cypher, A., Watch What I Do: Programming by Demonstration, MIT Press, Cambridge, Mass. 1993.

Etzioni, O., A Softbot-Based Interface to the Internet, Communications of the ACM, July 1994.

Frank, Martin, Standardizing the Representation of User Tasks, AAAI Spring Symposium on Acquisition, Learning and Demonstration: Automating Tasks for Users, Stanford, CA, March 1996.

Gaxiola, D., Tatlin: Integrating Commercial Applications into Programming by Demonstration, MIT BS Thesis, 1995.

Goodman, Danny, Danny Goodman's AppleScript Handbook, New York: Random House, 1994.

Kiczales, Gregor, and Daniel Bobrow, The Art of the Meta-Object Protocol, MIT Press, 1992.

Kosbie, D., and B. Myers, A System-Wide Macro Facility based on Aggregate Events, in [Cypher, ed. 93].

Lansky, A., M. Friedman, L. Getoor, S. Schmidler and N. Short Jr, The Collage/Khoros Link, Workshop on AI and the Environment, IJCAI-95, Montréal, Canada, August 1995.

Laubsch, J, Zebu: A Tool for Specifying Reversible LALR(1) Parsers, Hewlett-Packard Labs, 1992. ftp:// ftp.digitool.com/ pub/MCL2/contrib/

Lieberman, H., Mondrian: A Teachable Graphical Editor, in [Cypher, ed. 93].

Maes, P., and Robyn Kozierok, Learning Interface Agents, AAAI Conference, 1993.

Myers, B. and Werth, A. Tourmaline: Text Formatting by Demonstration, in [Cypher, ed. 93].

Newell, A., and David Steier, Intelligent Control of External Software Systems, AI in Engineering, Vol. 8, pp. 3-21, 1993.

Piernot, P., and M. Yvon, The Aide Project: An Application-Independent Demonstrational Environment, in [Cypher, ed. 93].

Potter, R., Triggers: Guiding Automation with Pixels to Achieve Data Access, in [Cypher, ed. 93].

Rich, C. and C. L. Sidner, COLLAGEN: When Agents Collaborate with People, Autonomous Agents Conference, Marina del Rey, California, February 1997.