Attaching Interface Agent Software to Applications

Henry Lieberman

Media Laboratory
Massachusetts Institute of Technology
Cambridge, Mass. USA
lieber@media.mit.edu

Abstract

There has been much recent interest in the idea of "intelligent interface agent" software. An intelligent agent is software that can take independent actions in the interface on behalf of a user's goals, without explicit intervention by the user. Examples of such agents include Apple's hypothetical "knowledge navigator" [Apple 89], programming by demonstration systems [Cypher 93], mail and news filtering agents [Kozierok and Maes 93], and others.

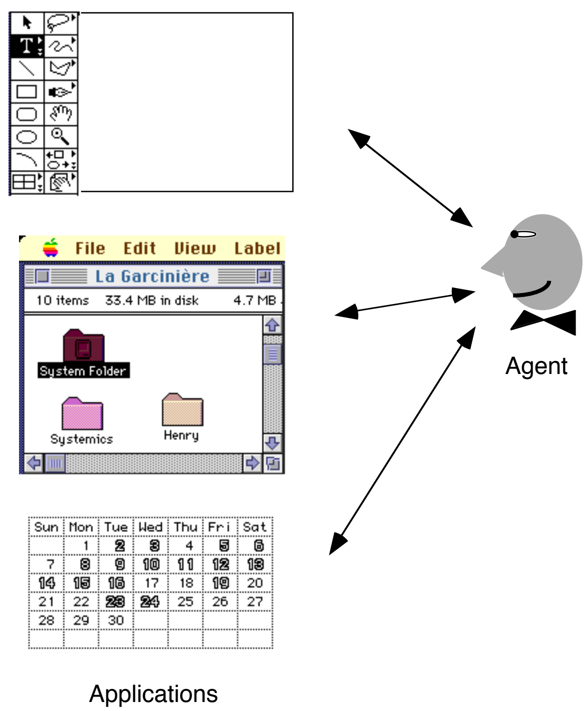
In most agent experiments to date, it has been necessary to implement the application in whose domain the agent operates from scratch, in order to enable the necessary application-agent communication. We would like to be able to "attach" an agent to a application that has been written more-or-less conventionally, or at least be able to tell an application developer what the agent may require. This paper explores the relation between agent software and more traditional applications whose interface is intended to be operated by a human user rather than a program.

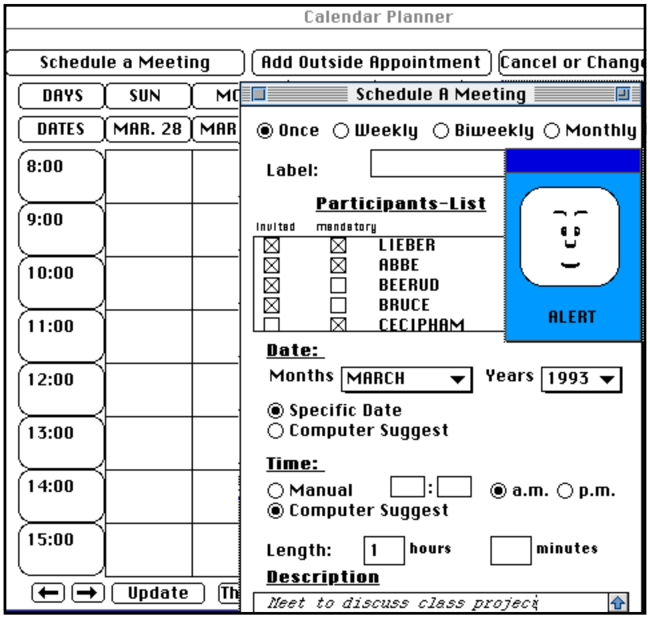
We've implemented some agents

In our group at the MIT Media Lab, we have implemented several projects which provide "intelligent agents" for the interfaces to common software packages. Mondrian [Lieberman 92] is an object-oriented graphical editor, similar to MacDraw, which has a learning agent. It records procedures that are visually demonstrated on concrete examples, and has some ability to generalize so that the recorded procedures can operate on examples other than those on which it was originally taught. Mondrian is an example of an *instructible* agent, one in which the user interacts with the agent in order to influence its future behavior. [Kozierok and Maes 93] implemented a meeting scheduling agent that detects patterns in users' responses to meeting requests in order to respond automatically or make suggestions in the future. [Sheth and Maes 93] has an agent that detects patterns of user interest in Usenet News messages.

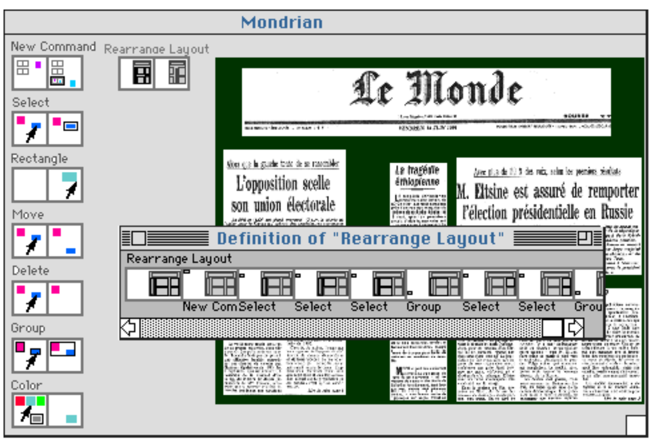
Our primary interest in these projects was experimenting with the learning algorithms and learning interfaces. However, we were forced in each case to implement our own versions of the application domain interface. Mondrian contains a home-brew graphical editor, and the calendar and mail reader programs were largely implemented from scratch. As a result, our versions of these programs have fewer features than commercial versions, and this limits the extent to which we could perform realistic user tests. We have talked to many other researchers performing similar work in other places and find most are also forced into the same choice.

We would like to be able to appropriate a commercial graphical editor like Canvas, a scheduling program like Meeting Maker, or a mail reader like Eudora, [or all of them at once!] and concentrate on developing our agent software. Yet there is no currently adequate way to for another program written on the same machine to harness the same power that these programs bring to their end users through their interface.





Calendar agent



Graphic editor agent

What's the problem?

There are two categories of problems that arise. The first is the fact that commercial applications are typically written so that they are not "callable" by another program, and their data structures are not accessible by another program. The only access to their functionality is through a user interface intended to be operated by a human in real time, not by an automated agent, which may be acting asynchronously. Finally, many kinds of interface extensions may require access to the source code, which is typically not accessible to either the end user, nor to an automated agent acting on his or her behalf.

Much of this is a historical consequence of the way that commercial computing environments have developed. Luckily, these problems are not conceptually serious, though they may cause significant practical difficulty. Most of them can be solved by the judicious application of object-oriented programming technology, though this has been slow to appear in commercial systems.

The second kind of problem has to do with how to structure the communication between application and agent. This is a more significant conceptual issue, involving a division of labor between the agent's tasks and the application's. It may involve decisions about how the agent stores knowledge about the application's functionality, negotiation between the agent and application, and parallelism issues.

It is important to separate these two sets of issues because they often get confused in discussions about the problem. In the long term, we can expect the next generation or two of operating systems to address the systems issues. In the short term, several proposed schemes for "inter-application communication" fall short of complete object integration, but may prove adequate for limited experiments. The more profound issues about division of labor and cooperative interfaces are research questions that remain.

What's an application?

The dominant paradigm in commercial software for personal computers has been that of an "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 slightly by allowing several applications to be "open" at once, short-circuiting the sometimes tedious leave-and-enter cycle.

The paradigm of thinking of software in terms of self-contained and isolated applications 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 are seen as a way of supplying software that acts as the representative of the user's goals in the complex environment. Agent software can provide glue between the applications, freeing the user from the complexity of dealing with the separate application environments.

Curiously, several research computing environments that appeared in the late 70's and early 80's were much better than current commercially popular systems from the integration standpoint. MIT and Xerox Lisp Machines, Smalltalk-80, and the Japanese ICOT Inference Machines each provided a single-language environment in which there was no real notion of an isolated "application". A graphics program that wanted text-manipulation facilities could simply call the text functions without "leaving the graphics application and starting up a text application", or communicating via some cumbersome event, network, or shared-file protocol. Those systems were ultimately not commercially successful because they were perceived to be inefficient, lacked attention to user interface issues, relied on specialized hardware, sold too expensively, or other reasons. But the kind of integration achieved by these systems needs to be looked at again as a goal for the next software generation.

Object-oriented programming issues

Many difficulties in integrating applications are basically object-oriented programming issues. That they arise at all is a consequence of the fact that most commercial operating systems and applications are not fully written in an object-oriented fashion. The term "object-oriented" is sometimes applied to applications because they are written in C++ or Objective-C, but these languages are, strictly speaking, object-like extensions to the procedural C language, and fall short of achieving the required integration. We list a few of the issues below.

- **Independence of representation from implementation.** Agent software should not have to depend upon details of memory storage formats for application data.

- **Attachment of procedures to objects.** Agents may want to take action when a given data element is accessed, changed or processed.

- **Inter-application object references.** Agents may want to hold data that references objects or components across applications. In the absence of an inter-application object system, the agent may have to maintain shadow objects that duplicate application objects and worry about how to maintain consistency.

- **Dynamic extensions to the object environment.** Many current environments do not have the ability to create or process new kinds of objects once the application is started. The interface itself may need to be dynamically extended, as we discuss below.

- **A persistent object store.** Agents may take action that persists across multiple user sessions.

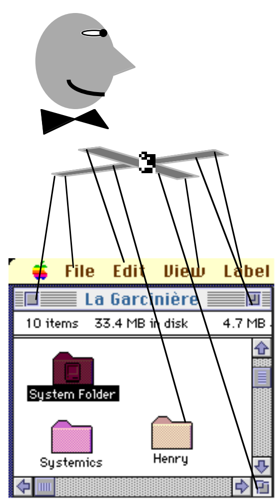
We won't discuss these issues further, as they are treated in the literature on object-oriented programming. We simply note that these are ultimately all requirements for writing application-independent agent software. We have confidence that future generations of operating system and application software will address these issues, though the transition period may be painful, and we will assume the existence of these facilities as a basis for long-term planning of agent programming environments.

Commercial efforts at application integration: "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, and simple macro-recording, but it is also envisioned that these might serve as support for future interface agent software as well. Examples of such efforts include AppleEvents, HP New Wave for Microsoft Windows, Microsoft OLE, and others.

These 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.



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. 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" 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.

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.

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.

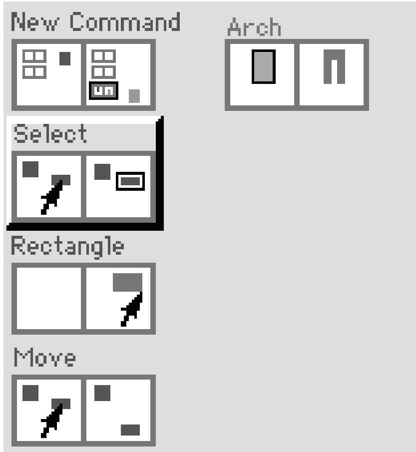
To take a concrete example, should we include mouse movements in the protocol that an agent should be able to receive? If we say yes, then we are obligating every application to report every time the mouse moves, which is potentially inefficient, and wasteful in the case the agent does not need this 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

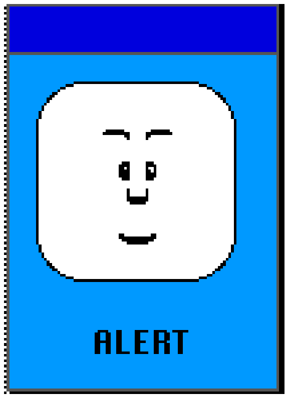
Another issue is that the agent itself may need to interface directly with the user, and most applications are not currently prepared to *share* their interface with another program whose interface they do not know in advance.

Mondrian, 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 right is an icon that operates the agent, the three icons below it are draw operations provided by the draw program, and the Arch icon was added by the agent to the interface as a result of user interaction.



Application and Agent-generated icons

Kozierok's Calendar Agent displays a cartoon face whose expression indicates the state of the agent and feedback about its predictions. Cypher's Eager [Cypher 91] colors menu operations for anticipatory feedback.



Anthropomorphized calendar agent

Applications should be able to accept as part of the protocol for communication with agents, 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. 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.

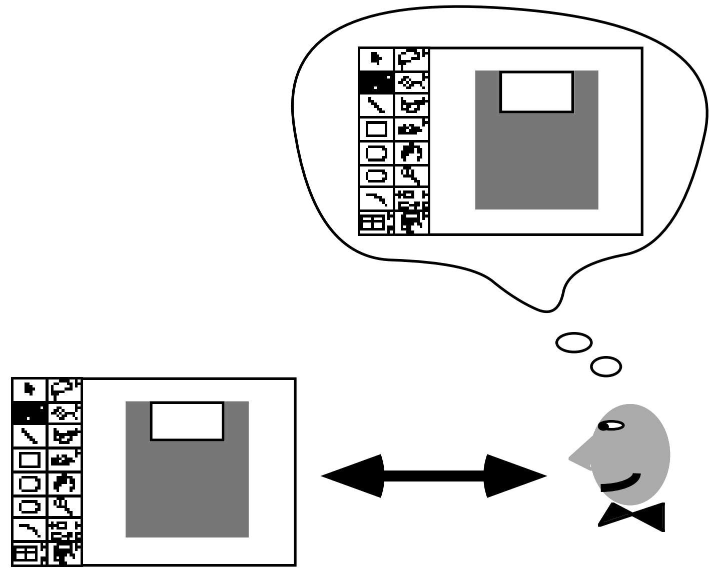
Agent meta-interpreters

One technique that we believe holds promise for achieving some independence between agent and application coding is the use of *meta-interpreters*. In traditional programming language theory, a meta-interpreter is an interpreter for a language written in that language itself, and which allows the target language to be extended by extensions to the interpreter. The best-known modern example is the CLOS *meta-object protocol* [Kiczales and Bobrow 92], which shows that judicious implementation allows considerable latitude in extending the target language without penalizing the implementation in terms of efficiency. Meta-interpreters have also been in long use in the logic programming community, and meta-rules have been used to introduce agent-like control into expert systems. They have the property that the meta-interpreter may make significant decisions about control issues without any awareness on the part of a program written in the target language.

We are not proposing that application code be run directly with a meta-interpreter. That would probably introduce too much overhead, and is impractical given current application implementation techniques. But many UIMS systems [Lieberman 85] provide an *interface interpreter*, which operate on menu and icon selections as functions, and direct-manipulation objects as arguments. We are exploring the idea of embedding agent functions in an *interface meta-interpreter*.

As we have noted, the event loop of a modern direct-manipulation interface effectively acts as an interpreter. An application that is "recordable" using an event protocol such as Apple Events, effectively writes a program written in the *event language*. An application that can be controlled by a *script* [e.g. Apple Script or TCL in Unix systems] is effectively acting like an interpreter for the event language. Such interpreters may be amenable to meta-interpreter techniques.

An agent's "mental model" of an application



Beyond these systems and interface issues are deeper issues about how agents represent what functionality is implemented by an application and act upon it for response to user requests, learning, detecting patterns in user behavior, and other agent functions. These questions fall into the domain of artificial intelligence rather than interface issues per se. The basic question is what is in the agent's "mental model" of the application?

In user interface research, a fundamental question is, "What is the user's mental model of an application?". Good interfaces are those that encourage the user to develop a mental model of the application that is consonant with their model of their working domain, and that accurately reflects the objects and operations that are implemented in the interface. The user can then translate their needs into the application's operations and interpret results [Norman's "gulf of execution" and "gulf of evaluation"].

An intelligent interface agent must also have a "mental model" of the application. For the agent to have utility to the user, the agent's mental model of the application should have some correspondence to that of the user. To help the user bridge Norman's gulfs, it should be able to relieve the user of the burden of translating goals into operations in the interface.

This does not necessarily mean that the agent need have an explicitly represented or one-to-one mapping between goals and interface operations. Agents can also adopt behavior-based, case-based, or neural net approaches where goals and methods are only implicitly represented.

In current work, we are interested in categorizing the requirements of various agent learning, generalization, and action strategies, so that we can determine how application situations need to be described in order for the agent to carry out its functions.

For example, Mondrian's generalization strategy is driven by an object-oriented protocol that takes into account the actions and objects being generalized, and the graphic context. Object inheritance is used to share knowledge about how to generalize related kinds of actions and objects. A case-based reasoning agent might have a protocol that involves situation descriptions, indexing according to a set of features, matching criteria to tell when two situations are considered similar, metrics that say how similar, and adaptation criteria for solutions.

The foreign object problem

Many programs must deal with *foreign objects* - objects that are not in a program's immediate address space and that the program therefore cannot handle directly. Foreign objects may be written in a language different from the language of the original program, they may be in another process, or they may be stored on another machine on a network.

The *foreign object interface* 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.

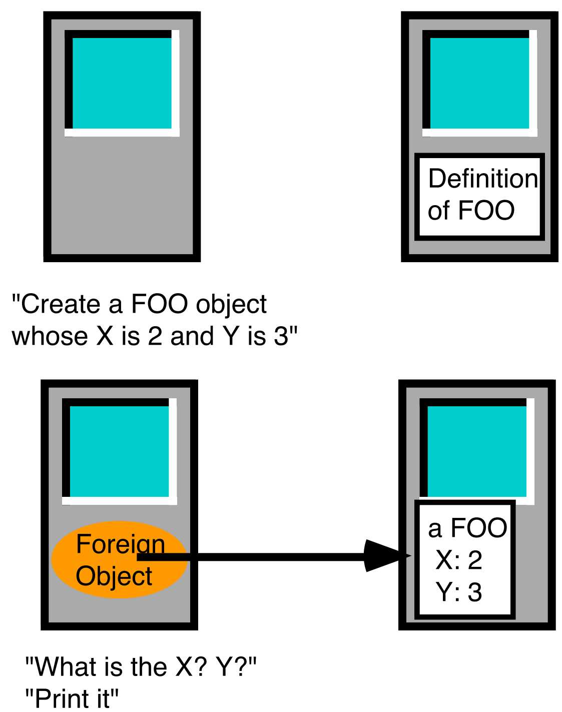
The foreign object interface is intended to be complementary to *foreign function interfaces*, which are typically found in implementations of Lisp and other languages. Remote procedure calls are another species of foreign function interface, one in which the foreign call involves network communication.

Such interfaces allow programs to call functions that are outside its own address space relatively transparently. But they provide no help for management of the objects that are passed as arguments to those functions. Some foreign function interfaces do provide translation of basic data types such as numbers and sstrings, but complex objects that may point to other objects are not typically provided for. Leaving object managment up to the applications programmer is typically too great a burden, aand effectively discourages routine use of multi-language or multi-process programs.

The prototype foreign object interface

We have implementd a prototype foreign object interface for Macintosh Common Lisp, which supports foreign objects using CLOS, the Common Lisp Object System. The foreign object interface permits a client program to create object instances that "live" on another machine [the server]. The definition of the object's class need only be loaded on the server machine.

The client program gets back a *foreign object,* which it can treat as a representative of the object living on the server machine. Functions applied to the foreign object, and requests to access the components [slots] of the foreign object result in forwarding the request to the server, which returns the answer. Thus, the foreign object can be treated transparently as if it were local to the client program.



It is important that foreign objects are created in a lazy or on-demand manner. The client progam 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.

Remote interfaces

To show one way in which the foreign object interface can be used, we show that the foreign object interface could be used to couple two user interfaces running on different machines.

How the foreign object interface works

Objects called CONNECTIONs represent remote hosts. There are CLIENT-CONNECTIONs and SERVER-CONNECTIONs, and these are further specialized by the communication mechanism, in our case APPLE-EVENT-CLIENT and -SERVER-CONNECTIONs. To initiate sharing, client and server connections sharing the same communications mechanism and the same identifying name are created.

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 appears in most Lisp systems for translating strings into references to objects [in Lisp's case, symbols]. 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. FOREIGN-INTERN translates a foreign object to an object on the server. FOREIGN-EXTERN performs the opposite transformation.

The metaobject protocol [Kiczales and Bobrow 92] can be used to achieve transparency between the use of foreign and local objects. MAKE-INSTANCE, SLOT-VALUE and FUNCALL are defined on foreign objects to perform their respective operations on the remote objects they represent.

Implementing the foreign object interface

The system exchanges objects with a Lisp running on another machine using the Apple Events [Apple 93] inter-application communication protocol. We used an Eval Server [a server task which evaluates textual expressions sent to it by a client] implemented in Apple Events. The Eval Server is much more powerful than is necessary to implement a foreign object interface. Some might even accuse us of "cheating" for using it, in view of the fact that such a thing would be hard to implement in C/C++. But in fact, the use is fairly restricted, limited to either calls to functions within the foreign object interface itself, functions that are implemented by the server program, and primitive data types such as strings and numbers. A C/C++ version of the foreign object interface could be written, and the server program could make its functions known to the client by a mechanism like the AETE resource in Apple Events. Thus, C++ and Lisp programs could co-exist on either side of the network barrier.

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.

Conclusion

We have surveyed a number of issues that arise in considering the problem of how to write interface agent software that can learn from user interface actions, take action independently in the user interface and interact with the user about how to perform these tasks. We have also considered what kind of support is needed from operating systems and applications to support agent software. The issues divide themselves into systems-level issues, including support for inter-application objects, and artificial intelligence issues, such as how the agent models the application's functionality and the division of labor between application and agent. We have based this analysis on experience with several agent-based applications, and are now working on generalizing our experience to produce more general software tools for application independent agents.

Acknowledgments

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

Support for this work comes from research grants from Apple Computer and the National Science Foundation. Additional support is provided by Alenia Corp., and ARPA/JNIDS.

Bibliography

[Apple 89]
Knowledge Navigator (Video)

[Apple 93]
Inside Macintosh: Inter-Application Communication, Apple Computer, Cupertino, CA, 1993.

[Cypher 91]
Allen Cypher, Eager: Programming Repetitive Tasks by Demonstration, in CHI '91.

[Cypher, ed. 93]
Watch What I Do: Programming by Demonstration, MIT Press, Cambridge, Mass. 1993.

[HP 91]
HP New Wave 4.0: Desktop Manager for Windows, Hewlett-Packard, Palo Alto, CA, 1991.
[Kiczales and Bobrow 92]
Gregor Kiczales and Daniel Bobrow, The Art of the Meta-Object Protocol, MIT Press, 1992.
[Kosbie and Myers 93]
David Kosbie and Brad Myers, A System-Wide Macro Facility based on Aggregate Events, in [Cypher, ed. 93].
[Lieberman 85]
Henry Lieberman, There's More to Menu Systems than Meets the Screen, SigGraph 85.
[Lieberman 92]
Henry Lieberman, Mondrian: A Teachable Graphical Editor, in [Cypher, ed. 93].
[Kozierok and Maes 93]
Robyn Kozierok and Pattie Maes, A Learning Interface Agent for Scheduling Meetings, Workshop on Intelligent User Interfaces, 1993.
[Maes and Kozierok 93]
Pattie Maes and Robyn Kozierok, Learning Interface Agents, AAAI Conference, 1993.
[Piernot and Yvon 92]
Philippe Piernot and Marc Yvon, The Aide Project: An Application-Independent Demonstrational Environment, in [Cypher, ed. 93].
[Potter 93]
Richard Potter, Triggers: Guiding Automation with Pixels to Achieve Data Access, in [Cypher, ed. 93].
[Sheth and Maes 93]
Beerud Sheth and Pattie Maes, Evolving Agents for Personalized Information Filtering, IEEE Conference on Artificial Intelligence for Applications, 1993.