IBOTS: Agent Control Through the User Interface

Luke S. Zettlemoyer, Robert St. Amant, Martin S. Dulberg
Department of Computer Science
North Carolina State University
EGRC-CSC Box 7534
Raleigh, NC 27695-7534
{lszettle | stamant | msdulber}@eos.ncsu.edu

ABSTRACT
This paper describes an *ibot*, a specialized software agent that exists in the environment of the user interface. Such an agent interacts with applications through the same medium as a human user. Its sensors process screen contents and mouse/keyboard events to monitor the user's actions and the responses of the environment, while its effectors can generate such events for its own contributions to the interaction. We describe the architecture of our agent and its algorithms for image processing, event management, and state representation. We illustrate the use of the agent with a small feasibility study in the area of software logging; results are promising for future progress.

KEYWORDS
Agents, intelligent assistants, user interface

INTRODUCTION
Agents that collaborate with users in an interactive software environment are becoming increasingly common. An agent can act as a personal assistant to the user, helping to solve difficult or unfamiliar problems. The agent's contributions take the form of making suggestions, exploring and demonstrating alternatives, critiquing user decisions, and so forth--in the ideal case, all the activities we might associate with an intelligent human assistant.

Commercial interactive software applications have begun to provide sophisticated application programmer interfaces (APIs) for agents in the user interface. While this is a step in the right direction, conventional techniques have subtle shortcomings. Introductions to programming user interface agents usually open with some variation on this: "In order to implement an agent for your application..." This simple beginning contains several unstated assumptions.

- *You are the developer of the software for which you would like to build an agent.* Without being able to modify and recompile the application, your ability to add functionality is somewhat limited.
- *Your agent's actions will be limited to a single application.* Interoperability between applications, however, is one of the most highly-touted advantages of modern interactive software; an agent might reasonably be expected to help users cope with the potential complexities.
- *You have foreseen the uses for which your agent will be needed.* That is, your agent's design and functionality must be compatible with the existing application; if the necessary hooks are missing, the application or API may need modification.

As others have recognized, not all of these assumptions will hold in practice. Lieberman, for example, stresses the need to attach agents to conventional applications while minimizing the advanced planning required of the program's developer [3]. Our goals are largely inspired by his work in integrating user interface agents into conventional applications. Our approach demonstrates significant progress toward this goal, in the form of a prototype agent that can access off-the-shelf applications through the user interface, with no reliance on the foresight of application developers.

In the long term, we hope to provide users with agents that work across applications, that have access to all the functionality the users themselves can reach, and that can work with off-the-shelf applications. Our ideal agent, in principle, will act just as a human assistant might. Figuratively speaking, it will pull the keyboard and mouse over in front of itself, type or mouse a sequence of commands to show the user a possible solution or to carry out some task on its own, and then return control.

To do this an agent needs access to what the user sees and what the user is able to do. Our approach involves the notion of "external" user interface agents, agents that operate outside and independently of any given application, effectively as if we were to replace the human user with a software assistant in the user interface. By analogy to Etzioni and Weld's softbots, we call this type of agent an

IUI 99 Redondo Beach CA USA

ibot. In our implementation, an ibot's sensors are image processing algorithms that run over the frame buffer contents--processing exactly what a human user sees--and routines for reading the system event queue. Its effectors are routines for inserting events in the system event queue so that its intentions can be conveyed to the interface, in the same way the commands of a human user would be. The ibot agent maintains a simple internal model of this environment that it can reason and plan about. The result is an agent that interacts with an application, in both input and output, through the same medium as the user.

An eventual goal for our ibot research is an agent that automatically builds a representation of the visible user interface (learning if necessary), observes the user's actions and modifications to the information visible in the interface, and provides the kind of intelligent assistance described above. We have made significant early progress toward this goal, in that the pieces (sensors, effectors, internal model-building routines) are in place. To simplify our proof of concept, we have arranged for the interaction of our prototype with applications to occur off-line, without the user in the loop. We expect that in the near future we will consider issues involving more robust real-time agent interaction.

We demonstrate the effectiveness of the ibot approach with a proof-of-concept application in the area of software logging. An ibot agent records the actions of the user in solving a specific problem using an off-the-shelf application. It plays back the trace offline, analyzing patterns from a usability perspective. The goal of our analysis is to identify inefficiencies, to suggest improvements to the interface or to sequences of interactions.

In the remainder of this paper we describe the architecture of the current prototype, the domain-independent image processing techniques that act as the system's sensors, the mouse and keyboard actions that act as the system's effectors, and the internal model the agent maintains of its environment. We show how these components mesh in the context of the logging application and discuss its potential strengths and limitations as a practical tool.

RELATED WORK

The ibot effort draws on three areas of research: user interface agents, programming by demonstration (PBD), and programmable user models (PUMs).

Lieberman outlines a number of areas relevant to our approach. His discussion emphasizes the importance of granularity of event protocols, styles of interaction with the user, and parallelism considerations. Event granularity determines the level of abstraction at which an agent interacts with an interface. For example, should mouse movements be included in the information exchanged? If not all mouse movements (possibly a very large number, depending on the sampling rate), then which ones are important? An interaction style describes the way in which an agent interacts with the user. That is, it may not always be sufficient for an agent to execute commands in an interface; it may be necessary to communicate directly with the user. This can force a different interaction style, for example, on an agent designed mainly for direct manipulation interactions. Issues of parallelism can enter the picture when the agent and the user both try to manipulate the same interface object. System performance can also be affected by the activities of an agent. Our current system is too immature to have considered these issues in detail. It works at a system event granularity, though it can use inference to reach a higher level of abstraction in some cases. As yet it has no mechanisms for communicating directly with the user or managing parallel activities.

Potter's TRIGGERS system [4] is an early example of an approach similar to ours. TRIGGERS performs pattern matching on pixels on the computer screen in order to infer information that is otherwise unavailable to an external agent. A "trigger" is a condition/action pair. Triggers are defined for such tasks as surrounding a text field with a rounded rectangle in a drawing program, shortening lines so that they intersect an arbitrary shape, and converting text to a bold typeface. The user defines a trigger by stepping through a sequence of actions in an application, adding annotations for the TRIGGERS system when appropriate. Once a set of triggers have been defined, the user can activate them, iteratively and exhaustively, to carry out their actions. From TRIGGERS we adopt the notion that the screen itself is a powerful source of information for an agent, if it can be properly interpreted.

A PUM is an engineering tool for interface designers [5,2]. It simulates a user, to some appropriate degree of accuracy, in order to provide a designer with feedback before more extensive (and expensive) usability testing. Our agent can be seen as a very limited form of PUM. Its sensors and effectors interact with an interface in the same way as a human user, although the models behind the interactions are not realistic models of human motor control, perception, or cognition. These, however, are natural extensions of the approach, and are under current consideration.

IBOT ARCHITECTURE

Our agent architecture is designed to mimic the interaction of a human user with one or more software applications. Leaving aside visual interpretation, possibilities for interaction are very limited: the user can make a few stylized gestures with the mouse and press various keys on the keyboard, watching the screen to see the effects of these actions. The architecture provides sensors and effectors at the same point of interaction as the human user.

Figure 1 shows the architecture and how it is layered on top of the operating system. Hooks are placed into the operating system that retrieve the information needed to interact with any executing application. The agent's low-

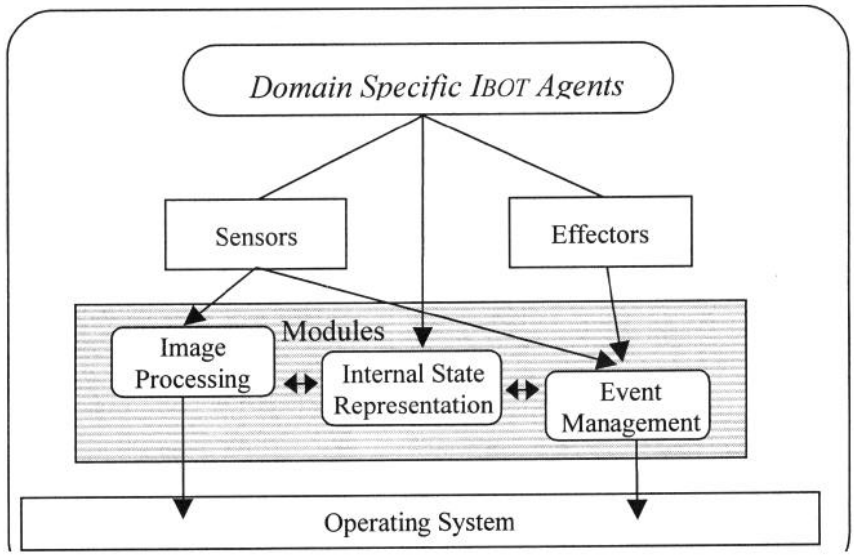

Figure 1 System Architecture and IBOT agents

level interactions with the operating system gives the appearance of direct interaction with application interfaces.

The architecture is divided into three separate modules: the image processing module (IPM), the event management module (EMM) and the internal state representation module (ISRM). Each gathers information from the other modules or the operating system to complete its specified tasks.

The three-module design very roughly models the basic human facilities that are brought into play during interaction with a user interface. The image processing module corresponds to the eyes of a user, with the ability to observe (though not necessarily interpret) visual cues on the screen. The event management module manipulates the system event queue to simulate the range of interactions a user can perform with a keyboard or mouse. Finally, the internal state representation is the system's simple equivalent of a mental model of objects in the interface.

Image Processing Module

Through a set of domain-independent image processing algorithms, the image processing module generates information about the contents of the screen, including visual identification of user interface widgets. Inference for this identification follows a conventional three-stage process, which starts by examining the contents of the screen buffer at a low-level pixel representation and finishes with a description of the high level user interface components.

The process begins with a set of pixel colors and coordinates. Pixels are grouped according to screen location and color, forming pixel groups. These pixel groups provide information through operators, which are used to describe the visual properties of the screen buffer. Finally we apply a set of heuristics that use the pixel group operators to determine exactly where the user interface components reside on the screen.

The initial screen capture is managed via operating system calls. The resulting array of pixel color values is then fed through the IPM, which carries out a process of segmentation, representation, and description. These three stages are an adaptation of a general three-step image processing process [1].

Segmentation

Segmentation applies a pixel grouping algorithm to pixels to distinguish groups by their differing colors, as shown in Figure 2. The purpose of this algorithm is to group together all pixels that are adjacent to each other and have the same color.

```
While Receiving Pixel Colors
If Pixel is already grouped and
        group color = pixel color
    return /* no change needed */
If Pixel is already grouped
      and group color != pixel color
        remove pixel from previous group
If Pixel is ungrouped then
        Create new pixelGroup containing pixel
For each 8-neighbor of pixel
  If neighbor is in a pixelGroup and neighbors
      pixelGroup color = the new pixelGroup color
   Merge pixelGroup with Neighbor pixelGroup
  EndFor
EndWhile
```

Figure 2 Segmentation algorithm

The result is that for any two pixels p and q, p and q belong to the same pixel group if and only if p and q are adjacent and share the same color. This provides a method to group

all of the raw color information into building blocks that are used later in the image processing.

The segmentation algorithm allows the pixels to be activated in any order and allows pixel values to be overwritten as parts of the screen buffer change. Partial or total analysis of the screen can be performed depending on how many pixel values are added through the algorithm. Once a complete block of pixel information has been obtained it can be passed to the Representation stage.

Representation

Representation is implemented as a set of operations that are called to gather higher level information about the pixel groups which have been formed. These operations can be divided into two categories, inter-group operations and intra-group operations.

Intra-group operations gather information about the relationships between pixels within a particular pixel group. This includes operations such as computing the bounding box of a pixel group, determining the eight neighbor connectivity of individual pixels, and computing estimations of the surface area and perimeter of pixel groups.

Inter-group operations identify relationships that exist between two or more pixel groups. Inter-group operations include determining the topological orientation of one pixel group in relation to another (e.g., left of, above, contained within) and operations that group pixel groups based on common characteristics derived from applying any of the pixel group operations (e.g., list all pixel groups contained in a particular pixel group).

An example of an inter-group operation currently called during image processing is GetArea(). GetArea() makes calls to other group operations and then returns an estimate of the total area occupied by the pixel group. In pseudocode it can be represented as in Figure 3.

```
Operation GetArea()
    MaxPossibleNumPixels = GetWidth() * GetHeight()
    Area = ActualNumPixels() /MaxPossibleNumPixels
Return Area
```

Figure 3 An inter-group representation algorithm

The current set of operations provides basic functionality that can be extended to describe any functional or geometrical relationship between pixels and their corresponding pixel groups.

Description

Description is the third and final step. It makes use of all of the information gathered to impose meaning on the visual cues. Our method of description consists of the iterative application of a set of heuristic rules that categorize and identify user interface components.

To identify a widget, the IPM applies rules that specify its low-level geometrical components and any necessary relationships that exist between the widget and other pixel groups on the screen. For example, the rule in Figure 4 finds list boxes. All of the function calls correspond to the pixel group operations described above.

```
If there exists a downArrow()
    That is containedIn() a raisedButton()
    That is toTheRightOf() a rectangularTextArea()
    Which is recessed() and has a width()
            greater than its height()
Then we have found a list box
```

Figure 4 Rule to find a list box

The system processes all visible widgets with successive applications of its rules. This iterative application of operations and heuristics has been surprisingly successful. It is important to note that we have not focused on the sophistication of our image processing routines; they are quite simple. Our singular advantage in this area is the discrete, regular, and highly repetitive nature of patterns in modern GUIs. This allows us to apply relatively standard image processing techniques to great effect. The IPM has been able to identify every type of interface component for which we have taken the time to develop the heuristics. Figure 5 shows a partial list of components the system currently identifies and the number of pixel group operations directly called by the heuristic. We are developing geometrical analysis operations and character recognition algorithms within this image processing framework that we expect will capture almost every relevant visual aspect of the display.

Component	*Number of pixel Group operations*
Buttons	6
Scroll Bars	8
Check Boxes	4
Radio Buttons	5
List Boxes	8
Windows	7

Figure 5 Partial list of interface components.

Figure 6 shows the type of information that is gathered through processing in the IPM. This shows how the system as a whole is able to find all of user interface widgets visible on the screen. The IPM was run on the dialog box on the left of the figure, creating an internal representation of all of the widgets. This internal representation was then used to draw the right side of the figure. All of the visible UI components for this particular dialog have been identified. The visual feedback shown in Figure 6 would normally not accompany a session with the system.

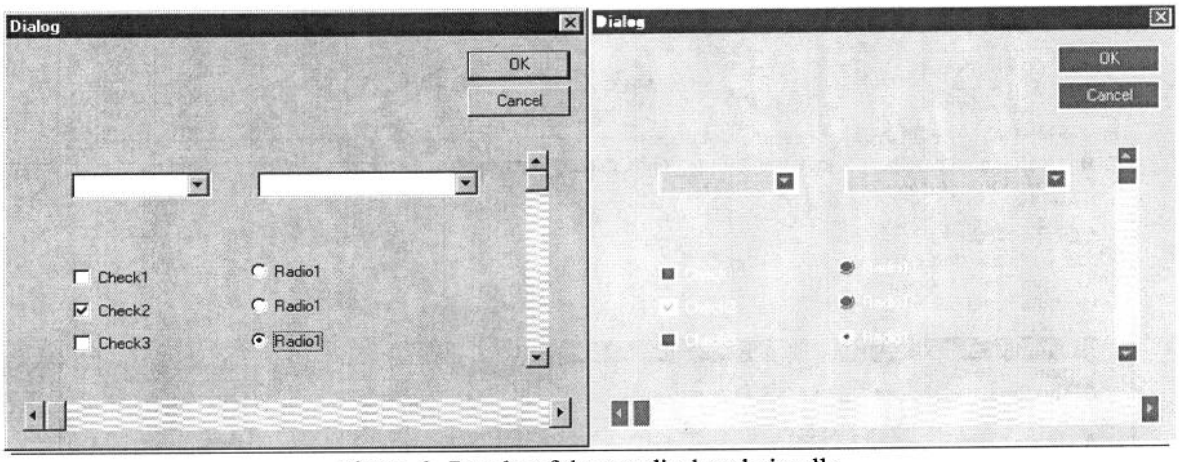

Figure 6. Results of the IPM displayed visually

Event Management Module

As both a sensor and an effector, the EMM provides the ability to interact with interfaces that the IPM has observed. This is achieved by directly accessing the operating systems event queue. The event handling module has the ability to both observe the events as they pass through the queue and to load the queue with its own events. We have restricted the EMM to consider only keyboard and mouse events because they are sufficient to provide for full interaction with a conventional interface.

The EMM can observe events in two modes. Currently it watches the event queue and selectively logs all events of interest—that is, those types of events that we have specified in advance. This allows the system to later return to this information to infer what has actually happened with the interface. However, the EMM also has the ability to process event information at runtime. This can be done in real-time with or without delay.

The EMM plays back events through its ability to insert items into the event queue in real-time. This is performed with short delays between events to allow a user to watch a particular action while it is being performed. While the EMM is performing an action, all keyboard and mouse input by the user is disabled, to prevent any confusion due to the temporary shift of control away from the user. When the EMM has finished, control returns to the user. Our work toward mixed-initiative control in this situation has only begun, but because we have only considered tasks that do not directly involve the user up to this point, we have encountered no problems. Nevertheless, the existing version of the EMM demonstrates that the effectors necessary for a future real-time advisory system are already in place.

In our current testing, the EMM operates in an off-line mode that allows it to recreate the actions of the user at its own pace. This allows other modules as much time as needed to infer meaning from the interactions that have occurred.

Internal State Representation Module

By gathering information from the EMM and the IPM, the Interface State Representation Module can combine the representation of the current appearance of the interface with the appropriate keyboard or mouse input in order to determine what is happening to a user interface at any particular time. This state information is recorded and made available for use by other modules in the system.

The system maintains a limited internal representation of the visual appearance of the interface in the form of a list of interface widget types and locations, coupled with lists of recent keyboard and mouse events. This information can be queried at any time to gather state information to support agent-initiated actions and observations.

Through access to the ISRM, an ibot agent gains access to basic system information, including

- *Common user interface knowledge represented in the ISRM.* The ISRM encapsulates the knowledge to identify basic user interface widgets across applications. This knowledge becomes accessible to an agent as lists of visible user interface widgets, which can be monitored or manipulated through the EMM.
- *State information about how the user is interacting with the application.* By watching the state of the mouse, keyboard, and application interface as represented in the ISRM an agent can gain valuable information about a user's actions. The agent can then formulate appropriate responses based on the user's past actions or inferred plans.

This information would be difficult for an agent to acquire through an API (assuming that one existed), while our system, through the ISRM, provides this basic information for all visible applications.

The system is not restricted to acting in any one domain and hence has no global knowledge representation. Instead, we foresee that developers will generate domain-specific representation systems on top of the information provided

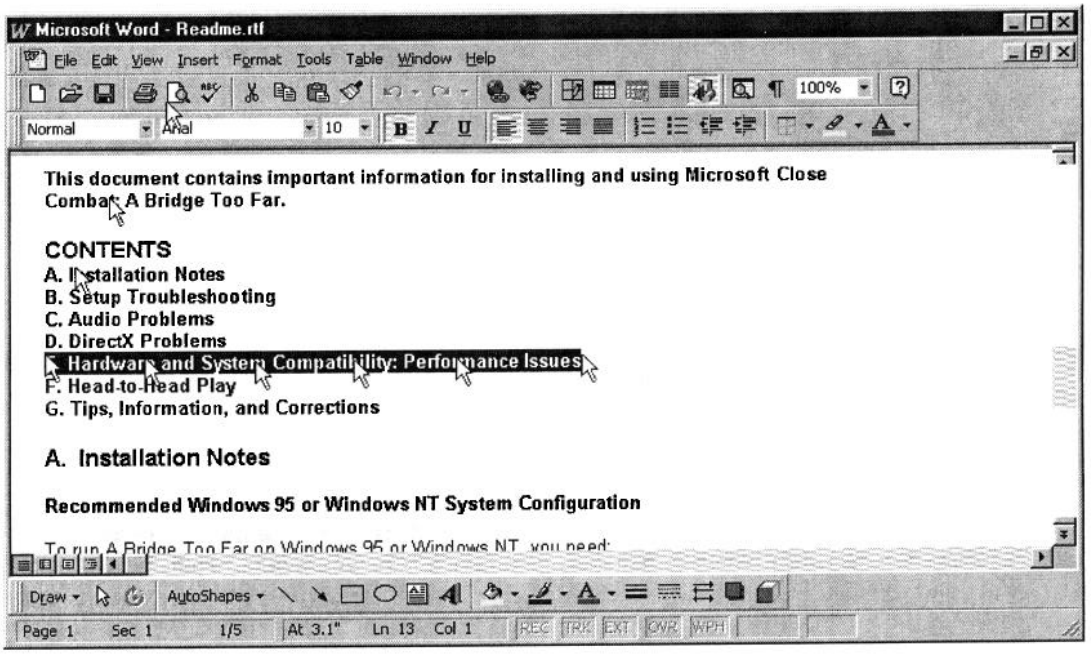

Figure 7 The system replaying users interactions in Microsoft Word

by the base system, tailored to the desired functionality of the agent.

SOFTWARE LOGGING APPLICATION

We chose software logging as a test area within the larger context of usability analysis for several reasons. First, some activities within usability analysis fit the profile of tasks appropriate for expert systems: codified domain knowledge, limited dependence on context, input data accessible by an automated system. Second, we are aware of no existing intelligent tools aimed specifically at usability analysis (though many more general-purpose intelligent tools, such as layout assistants, can be used to improve usability.) Finally, an automated system improves on the performance of a human observer in some areas simply due to increased consistency. For example, an automated system can monitor the placement and timing of low-level mouse and keyboard activity much more easily than a human observer.

Our current work is incomplete in several ways. First, the system is not yet interactive, so all of our analysis and most processing must be after the fact. Second, part of our effort is to incorporate enough knowledge about user interactions into the system to allow the system to compare expert behavior on a task with novice behavior, in order to identify areas for design attention. Only low-level representation issues have been considered so far, and no comparison is yet possible. The current status of the prototype is sufficient nevertheless to illustrate some of the functionality of the system and to show how a domain-specific extension of the basic architecture will work.

We began with an informal evaluation, which proceeded as follows. The experimenter launched Microsoft Word, maximized the application to the full screen, and loaded a text file. The EMM component was activated to monitor the experiment. Each experiment participant was then given two editing tasks, described in Figure 8.

Offline, after the interactive portion of the experiment had concluded, these traces were played back with the logging agent activated as seen in Figure 7. On the occurrence of specific patterns of mouse events (e.g., before and after a mouse down or mouse up event) the IPM activated its routines to record and process the frame buffer. A stream of screen specifications was the result, a structured text description of the kind of information shown graphically in Figure 6. Each token in the stream consists of a mouse/keyboard event or an element of a screen description, which is a list of a recognized widget type and its boundaries.

This task involves editing a file supplied by the experimenters, Readme.rtf, *in Microsoft Word.*

Task 1: Your task is to eliminate section E, "Hardware and System Compatibility. . ." from the document. Start by deleting the Section E header from the table of contents at the top of the document. Change the lettering of section headers F and G to E and F, respectively, to reflect the deletion. Now perform the corresponding changes in the body of the document. Save the revised file as `Readme1.rtf`.

Task 2: At the top of section A add the following table:

Processor	P100	P133	P166+
Capability	No	Maybe	Yes

Add page numbers to the document so that they appear on all pages in the bottom left hand corner. Save the revised file as `Readme2.rtf`.

Figure 8 Usability analysis tasks

These streams of tokens are combined into higher-level abstractions by a log analysis module. We begin with a geometrical description of the significant regions of the window and types of behavior we expect to see within these regions. For example, a sequence consisting of a mouse-down event in the menubar region of a window followed by vertical mouse-movement events in the menu that appears constitutes a menu-select operation. For our purposes, we are not so much concerned with identifying the sequences that correspond to common operations, but rather with sequences that show inefficient or even incorrect behaviors. We have implemented two examples:

- *Menubar search:* The user moves the mouse horizontally within the menubar region of a window. The more common case is preceded by a *mouse-down* event, in which the user is searching through the menus for a specific item, but some users move the mouse over the menubar headers without looking at the menus as they try to recall the placement of an item.
- *Menu search:* The user mouses down in the menubar region, which causes the appearance of a menu. This may be followed by a pause (during which the user is presumed to be reading the menu items) or by vertical mouse movements down the menu.

These sequences can be thought of as pattern recognition rules: they identify specific types of behavior of interest to the system. Currently the system does nothing with this information except to record it, so that we might later bring it to the attention of an interface designer. While this is sufficient from a usability perspective, the eventual plan is for the system to be able to interpret these sequences on the fly, to give the user assistance in unfamiliar situations.

In AI terms, we are effectively building plan critics, where the plan is the sequence of events the user steps through. A critic observes the construction of a plan for specific undesired patterns, and corrects them. In the current implementation only the observations are the responsibility of the system, their correction the responsibility of the human designer; future versions will integrate both areas of critic functionality into the system.

As a standalone system, the logging agent is far from complete. Only a few critics have been implemented, and its functionality is extremely limited. Nevertheless it shows something of the promise of the external agent approach to such problems.

CONCLUSION

The current prototype is implemented in approximately 1700 lines of C++ code. It runs in the Windows operating systems and gathers its information through WIN32 system calls. IBOT makes very little use of the OS other than to gather low level windows events and screen buffer information.

The architecture provides a framework that could be ported to any operating system that supports access to the mouse and keyboard event messages as well as access to the video buffer at a pixel level. This architecture is a framework that provides a tool to access information about the current state of the I/O devices that a computer provides for interaction, including the mouse, keyboard, and display as well as all of the potentially useful observable information that they provide.

The functionality of the system we have described overlaps that of conventional systems for logging user interaction, even macro recorders. These can easily collect the same kind of information, but analysis is a different matter. Without a detailed specification of the interface, it can be extremely difficult to interpret mouse movements and gestures in the appropriate context. The dynamically changing specification of an interface, as generated by our ibot agent, provides some of this necessary context.

One might see our work as a rather roundabout way to reach the goal of agents integrated into conventional interfaces. Unfortunately, the benefits of agent-based user interfaces are often not apparent to mainstream software developers. System software will not support such agents until they demonstrate their advantages, but in the meantime it is very difficult to experiment with them without such support. We are limited to home-grown systems where we have the control we need. Our approach solves several problems: it can be used with existing software, requires no knowledge of the internals of the software, runs across multiple applications, and is based mainly on platform-independent algorithms. In contrast to application-internal agents, our ibot has the disadvantage of being forced to reason almost exclusively based on visual cues and user actions. Our preliminary work, however, suggests that this approach can be adequate for some interesting classes of tasks.

REFERENCES

1. Gonzales, R.C. and Woods, R.W. *Digital Image Processing.* Addison-Wesley Publishing Company, Reading, MA. 1992.
2. Kieras, D. and Meyer, D. E. An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Human-Computer Interaction.*
3. Lieberman, H. Integrating User Interface Agents with Conventional Applications. *Proceedings of IUI'98.* (San Francisco, CA, January, 1998.) ACM Press, 39-46.
4. Potter, R. TRIGGERS: Guiding Automation with Pixels to Achieve Data Access. In *Watch What I Do: Programming by Demonstration.* MIT Press, Cambridge, MA. 1993.
5. Young, R.M., Green, T.R.G. and Simon, T. Programmable User Models for Predictive Evaluation of Interface Designs. In *Proceedings of CHI'89.* ACM Press, 15-19.