Toward Automated Exploration of Interactive Systems

Mark O. Riedl
Department of Computer Science
North Carolina State University
EGRC-CSC Box 7534
Raleigh, NC 27695-7534
moriedl@unity.ncsu.edu

Robert St. Amant
Department of Computer Science
North Carolina State University
EGRC-CSC Box 7534
Raleigh, NC 27695-7534
stamant@cs.ncsu.edu

ABSTRACT
The ease with which a user interface can be navigated strongly contributes to its usability. In this paper we describe preliminary results of a project aimed at making the evaluation of user interfaces from this perspective more routine. We have designed a system to carry out an autonomous, exploratory navigation through the graphical user interface of interactive, off-the-shelf software applications. The system is not a robust tool, but rather a proof of concept that can exhibit interesting behaviors. The traversal process generates a representation of the connectivity of the user interface, as well as navigational paths to specific commands. The reasoning component of the system is based on the ACT-R architecture, while the perceptual and motor components of the system are built on top of the SegMan perception/action substrate. We present the design of the system and its use in exploring a simple user interface.

Keywords
Cognitive models, interface agents, interface evaluation.

INTRODUCTION
User interfaces are generally designed to be learnable [5], in the sense that a curious user can learn at least some of a system's functionality through experimentation, by trying out its operations to see the results. This process teaches the user about the capabilities of the system and the way it decomposes the domain into separate areas of functionality. Ideally, all of a system's functionality should be accessible in this fashion (though in practice, especially in complex problem domains, this goal may be unattainable.) In operational terms, learnability is closely tied to the navigational properties of an interface.

Exploration is demanded by the design of menu-based systems, in which operations are organized hierarchically and information is made available incrementally, through the selection of these operations. As a very simple example, to open a document in Microsoft Notepad a user must enter the name of the document into the Open dialog. A user experienced with Notepad will know that the File menu contains a menu option called "Open" and that selecting this option will activate the Open dialog. A less knowledgeable user may still be able to infer this information from knowledge about other applications, and even a complete novice may learn it by trial and error. It is a reasonable assumption that all users must resort to exploratory navigation at some point in their use of an unfamiliar application. The prevalence of this activity makes it worthwhile to understand the navigational properties of an interface in human information processing terms.

A good deal of research has been devoted to analyzing the navigation properties of interfaces, especially for hypertext systems, from the perspectives of design and evaluation (e.g., browsing strategies [4, 6]). Our particular interest is in the design of automated systems to assist in evaluation. While the navigational process as outlined above can easily be grasped by even novice users, it poses a more difficult and interesting problem for an automated system; questions arise in several areas.

- *System issues*, e.g., can the low-level actions necessary for interface navigation be carried out by an automated process?
- *Task analysis issues*, e.g., which navigation paths are most important? Are all navigation paths relevant?
- *Cognitive issues*, e.g., what knowledge (procedural and declarative) is necessary for navigation to take place? How does navigation augment or modify existing knowledge?

We have developed a system, based on the ACT-R cognitive architecture, that autonomously carries out a limited form of exploratory navigation, in an unmodified interactive system. Production systems such as those based on ACT-R [1] and Soar [9] are well-adapted to the task of exploration. A production system as implemented in an ACT-R model encodes facts the agent knows about the world, or declarative knowledge, and production rules that transform declarative knowledge into behavior. The agent's behavior alters the state of the world, new declarative knowledge is generated, and new productions are used to generate behavior. Cognitive models like

IUI'02, January 13-16, 2002, San Francisco, California, USA.

ACT-R also have the ability to learn. The assumptions that a human agent might make when exploring a novel application interface can be encoded into declarative knowledge in a model. The actions that a human agent might make when manipulating the interface are encoded into production rules. A well-designed cognitive model should be able to start with a minimal amount of declarative knowledge about the user interface and a simple set of productions and, through exploring an application's interface, develop a declarative representation of the interface and how it operates.

The work described in this paper is part of a more comprehensive project in automated interface evaluation. Cognitive modeling architectures (e.g., ACT-R, Soar, and EPIC [8]) are now mature enough to produce reliable predictions about user performance, prior to evaluation with real users. The application of these models to specific problems, however, requires significant expertise in modeling, task analysis, and interface design. Our long-term goal is to build what we call Cognitive Model Interface Evaluation (CMIE) tools, systems that support the display of the user interface, experimental control over the cognitive model and its simulation runs, feedback on model execution, model execution diagnostics, and simple display facilities for model traces [11]. The system and the embedded cognitive model that we describe below constitute a step in the direction of a practical CMIE tool.

THE COGNITIVE MODEL

The ACT-R architecture is well-suited to the exploration of a user interface, in part because it explicitly divides knowledge into declarative and procedural elements. The declarative elements are chunks of factual knowledge stored in working memory. The procedural elements are production rules that operate on declarative chunks to generate behavior and to generate new chunks of knowledge. ACT-R models have been applied to many cognitive problems, such as counting and arithmetic, in which productions generate new declarations until a chunk that stores the answer to the problem is generated.

ACT-R also contains perceptual and motor modules that allow models to handle tasks of searching and attending to a computer display, under some implementation restrictions. The motor module simulates hand and finger movements for manipulation of a keyboard and a mouse, while the perceptual module searches and attends to features on the display. Production rules describe how the perceptual module should focus attention on perceptual elements in the computer display, and how the motor module should execute appropriate responses to the display. ACT-R models have been applied to simple problems, such as scanning menus for target elements, as well as to more complex problems, such as simulations of driving behavior.

The cognitive model we have designed explores a graphical user interface in real-time. It builds directly on well-understood models of low-level cognitive tasks. The model scans the display for meaningful visual features, the way a model might scan a menu list of words for a target word. Simple cognitive transformations are made to interpret how particular visual features might be used; widgets are identified and catalogued. The model chooses a widget and moves the mouse to click on that visual feature. We flesh out this brief description in the sections below.

A metaphor for exploring the user interface

An application's interface in the Windows graphical display commonly consists of a main window and some number of pull-down menus. Pull-down menus themselves look like windows, small bounded regions rendered to appear raised above the application's display. Menus can be thought of as separate from the main window, revealing functionality that was previously hidden. In our cognitive model, the graphical user interface is treated as a set of distinct screens that are causally connected but graphically separate. In this way the graphical user interface for a specific application becomes like a set of rooms, a common user interface metaphor. Each room has individual characteristics, such as the items it contains, that distinguish it from other rooms. Doorways provide a mechanism for transition between rooms. In the graphical user interface each window, dialog, or menu constitutes a screen that corresponds to a room. The features unique to each screen are familiar widgets: strings of text, buttons, etc. Some widgets are special in that their use causes a transition to a new screen. This is true of pull-down menus, for example; clicking on the string of a pull-down menu causes the appearance of a new screen containing a list of menu options. Some menu options cause transitions to new screens by causing dialog windows to appear.

Though the rooms metaphor is useful, it is not exact down to the lowest level of detail. In particular, transitions are not necessarily reflexive. For example, to transition to the Open dialog screen, we must transition to the file-menu screen by clicking on the "File" string and then transition to the Open dialog screen by clicking the "Open" string. Clicking on the "Open" string causes the menu screen to disappear and the Open dialog to appear. Closing the Open dialog does not cause a transition back to the file menu screen, but back to the main application screen, from which we started.

Building the cognitive model around the screens-as-rooms metaphor provides a useful representation of the application's interface when exploration is terminated. The representation of the application's interface in the agent's memory is of screens and the widgets that cause transition. Figure 1 shows a partial dump of the ACT-R

```
Main-Screen
  isa SCREEN
  prev nil
  type Persistent
  current t
Newscreen782
  isa SCREEN
  trigger Newmenu660
  prev Main-Screen
  type Transient
  current nil
Newscreen1586
  isa SCREEN
  trigger Newstring842
  prev Newscreen782
  type Persistent
  current nil
Newcloser908
  isa WIDGET
  name “cancel”
  screen Newscreen1586
  obj Text896
  goto Main-Screen
  kind Closer
Newmenu660
  isa WIDGET
  name “file”
  screen Main-Screen
  obj Text629
  goto Newscreen782
  kind Menu
Newstring842
  isa WIDGET
  name “open”
  screen Newscreen782
  obj Text806
  goto Newscreen1586
  kind String
```

Figure 1. Declarative representation of an application interface after exploration.

memory after exploration. Main-screen is the main application window, Newscreen782 is the file menu, and Newscreen1586 is the open file dialog. The widgets that enable transition between the screens are Newmenu660, the "File" string that causes the file menu to pull down, Newstring842, the "Open" string in the file menu that causes the appearance of the Open dialog, and Newcloser907 is the "Cancel" button in the Open dialog. The full representation would include a catalog of all transitional widgets as well as non-transitional widgets found on each screen. Figure 2 is a graphical representation of the declarative knowledge presented in Figure 1. The dialog on the top left is an application we will use for examples throughout this paper. The hierarchical order of the screens is explicitly captured, as are the transitional mechanisms. The pathways through the application's interface can easily be reconstructed from the model's internal representation.

Widget identification

The screens-as-rooms metaphor abstracts away some of the complexity of building a model that can identify and focus attention on windows and menus. However, the task of identifying widgets, especially those that will cause transition from one screen to the next, remains. We assume that the screen is segmented into visual features

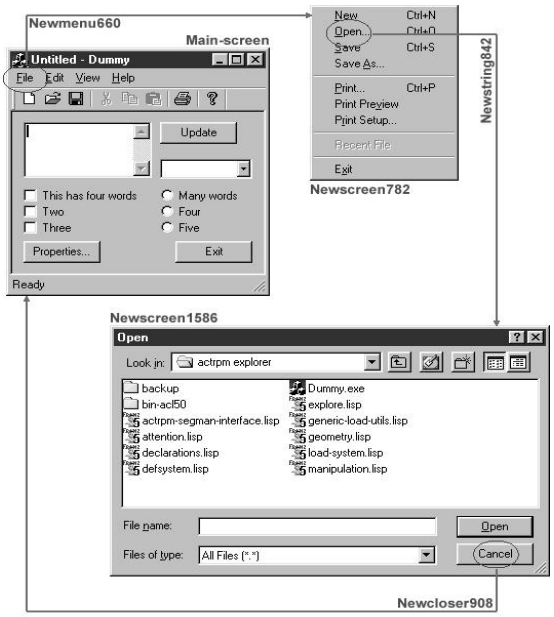

Figure 2. Graphical representation of declarative knowledge after exploration.

pre-attentively before the cognitive model ever considers the meaning of these visual features. Thus the cognitive model is presented with visual objects that consist of entities such as words, rectangles, icons, and so forth. The process of segmenting the graphical user interface is handled by an independent system (SegMan, as described below) tied into the ACT-R perceptual module. The ACT-R model needs only to direct attention to these visual objects and apply productions to the visual objects in order to catalog them. For simplicity's sake, the model categorizes all visual objects as widget or non-widget. Visual objects categorized as widgets are further categorized as transitional or non-transitional.

The identification of transitional widgets is essential to task of exploration. This identification is simplified by conventions of the Windows interface. Widgets that cause the appearance of dialog windows are often labeled with strings ending with ellipses, indicating that unseen functionality is to be revealed when the widget is activated. Widgets so labeled are automatically cataloged as transitional widgets. This convention is not by itself sufficient enough for the model to identify all transitional widgets. Pull-down menus are transitional widgets but do not use ellipses. Instead, the transitional nature of menus is implied by the location of strings of text at the top of the window, in the visually distinct area of the menu bar. Special production rules can identify menus as transitional widgets by applying a heuristic that tests whether strings are within the bounds of the menu bar, an approximation of the heuristic a human user might use to identify a string of text as a menu. Finally, there are widgets that do not follow the ellipses convention and are not menus. Further heuristic productions are based on a simple set of

```
(p attend-to-menu
   =goal>
          isa find-object
          kind menu
   =loc>
          isa visual-location
          time now
          attended nil
          screen-y (less-than 35)
   ==>
   !send-command! :VISION
      move-attention :location =loc
   =goal>
          lastloc =loc)
```

Figure 3. Production to find a menu string.

vocabulary terms that tend to identify transitions. Terms like "Properties," "Options," and "Setup" usually cause the appearance of dialog windows even if they do not contain ellipses. Other terms like "Cancel" and "OK" usually cause screen transitions by closing the current window or dialog. This vocabulary set is a plausible replacement for the contextual understanding that even a novice human user can rely upon.

Model validity

While the techniques and productions used in the exploratory ACT-R model are based on simpler, validated models, there is no guarantee that our model is also a valid representation of a human user performing the same exploring task. Our representational choices are consistent with higher-level representations of exploratory navigation (e.g. Spence's model [15]), but the model itself has not been validated with respect to human performance. This is not unusual in cognitive modeling research, in cases where a model is used as a proof of concept or to elucidate a plausible reasoning process, though it does reflect the maturity of the work. In our case, validation and performance tuning are open questions for further research.

MODEL PROCESSING

ACT-R provides perceptual and motor modules for interaction with a computer display. ACT-R models do not interact directly with the computer display, however, but instead rely on modifications to the user interface management systems (UIMS) for input and output. One of our goals is to build a system that can explore a real application running on the Windows graphical desktop, in a cognitively plausible manner. To do this, the model needs to segment the actual running graphical display in real-time and convert the display into visual objects that the ACT-R model could understand. SegMan, a successor to the VisMap system [18, 17], is designed to segment the graphical user interface using simple pattern-matching templates and procedural relationships in order to segment the pixels of the display into meaningful groups that represent letters, words, buttons, windows, and icons. The implementation details of the SegMan system have been presented elsewhere; suffice it to say that it does a

```
(p found-widget-menu
   =goal>
          isa find-object
          kind menu
          lastloc =loc
   =loc>
          isa visual-location
          attended t
   =obj>
          isa visual-object
          screen-pos =loc
          value =value
   =vocab>
          isa vocabulary
          name =value
          kind menu
   =screen>
          isa screen
          current t
          type persistent
   =state>
          isa module-state
          module :VISION
          modality free
          last-command move-attention
   ==>
   =newmenu>
          isa widget
          name =value
          screen =screen
          obj =obj
          kind menu
   !pop!)
```

Figure 4. Production to incorporate a valid menu feature into internal representation

reasonable job of identifying the features of the graphical user interface that are relevant to its usage. Using the SegMan system enables us to connect the ACT-R model directly to the state of the Windows graphical user interface from the same perspective as a human user.

Screen segmentation

To interface the SegMan system with the ACT-R perceptual model, we first constrain SegMan to segment only within the bounds of the current "screen" that the cognitive model is considering, eliminating all distracting information from consideration. This is done to make the implementation of the cognitive model easier, so that it does not have to determine the bounds of application window itself. Windowing applications present themselves in a way that naturally constrains the user's attention by presenting its windows and sub-windows in a stack. Even if the sub-windows are not occluding previous windows, the top-most window still forces the user's attention because all windows lower in the stack are inert. We assume that any agent that uses the graphical user interface will be capable of constraining its focus of attention to within the bounds of the field of view associated with the top-most window in the application's stack.

The SegMan system uses segmentation to determine which window or menu is top-most. Within the real estate of the top-most screen, the SegMan system further segments the display to identify all the words and their positions relative to the screen bounds. These strings are passed on to the

ACT-R perceptual module as text features. At this time all other features except text are disregarded. This is done because widgets in the graphical user interface tend to be labeled textually. Thus, searching for the string, "OK", can identify the "OK" button. Not all features have text labels. Icon buttons in the toolbar is a prime example of this exception. It should be noted, however, that the functionality provided by buttons on the toolbar is almost always available in the pull-down menus. We are able to utilize a significant portion of any standard Windows application when only considering textual information.

Production rules

The declarative portion of the ACT-R model's memory is broken into *chunks* where a chunk can be any piece of factual information and all the parameters that describe it or a goal that needs to be accomplished and all the parameters that explicitly describe the goal. The production rules are memorized procedures about how to process goals. A production rule fires when the goals and the declarative facts in declarative memory match a specific activation pattern. For example, the find-something-to-explore production fires if, and only if, there is an unfinished goal to explore the interface and when there is a transitional widget on the current screen that has not been explored yet. The find-something-to-explore production fires, resulting in a new declarative goal being generated: to click on the unexplored transitional widget and record the changes that place on the display. This new goal and any new chunks created in memory will cause other productions to fire, such as the use-widget production. Through the selection of production rules and their manipulations of declarative memory, the following pattern of behavior emerges:

1. Serially attend to every text feature on the current screen and create new declarative chunks for transitional widgets.
2. Choose a transitional widget that has not been explored.
3. Move mouse to the chosen widget.
4. Click on the widget and create a new screen chunk.
5. Repeat.

This pattern causes the agent to explore the application interface in a depth-first-like fashion. Fortunately, most applications do not provide many levels of sub-screens and have a low branching factor. Furthermore, almost all sub-screens provide some widget that closes the screen, providing the agent with the ability to backtrack. The exception to this is menus. Menu screens (Menu screens are referred to as *transient* screens because they disappear when an action is taken, as opposed to windows and dialogs which are referred to as *persistent* screens which do not disappear until commanded) disappear once used, so backtracking from a screen spawned by a menu screen necessarily requires the agent to backtrack two levels instead of the customary one level. Because of this, exploration of the application interface cannot be strictly depth-first. The agent must, after backtracking two levels past a menu screen, retrace its steps back to the menu screen in order to fully explore all additional transitions from the menu screen. We illustrated an example of this in the scenario where the agent navigates to the Open dialog screen. Upon closing the Open dialog screen, the agent is returned to the main screen. If the agent had not yet explored the ``Print setup'' option on the file menu screen, it must return to the file menu screen by retracing its steps. Fortunately, the agent can rely on its internal model of the interface.

Serially attending to widgets on the current screen is very similar to finding a visual target in a field of visual distractors. The production to shift attention to a menu means that the model must pick out a string of text from the sensory buffer (implemented in the low-level ACT-R perceptual module) that meets certain qualifications. That qualification is a visual feature that is text and is less than 35 pixels from the top of the screen. Figure 3 shows the production to shift attention to a menu string. Once the perceptual attention has been drawn to a candidate, the model can confirm that the attended visual feature is in fact a valid menu by comparing it to the vocabulary that has stored in declarative memory. If the attended visual feature is determined to be a menu, then a new declarative chunk is created to represent that visual feature. Figure 4 shows the production to incorporate a valid menu into the agent's internal declarative representation of the interface.

Choosing an unexplored widget to explore is trivial in ACT-R. On the left-hand side of the production we specify the criteria for a widget that has not been explored yet. ACT-R will automatically bind a declarative chunk that satisfies the established criteria. A widget that has not been explored has a null goto field, indicating that we do not know what screen we will transition to. With the unexplored widget chunk bound to a variable, we can push a new goal to click on that widget onto the stack. The production to the left in binds an unexplored widget and creates the sub-goal to click on that widget. It is important to indicate that the unexplored widget that we bind is not a widget that will close the screen because we do not want to backtrack to a previous screen before we have exhausted the unexplored widgets on the current screen. Another important aspect of the explore-screen production is that it also pushes the goal to register the next screen onto the ACT-R goal stack. This will ensure that when the widget is clicked the next goal to be considered will be to serially scan the screen for widgets. This sets up the child screen so that it can be explored and, since the explore goal is not removed from the goal stack, the explore-screen production will again become valid for the child screen,

```
(p explore-screen
   =goal>
          isa explore
   =screen>
          isa screen
          current t
          registered t
   =object>
          isa widget
          screen =screen
          goto nil
        - kind closer
   ==>
   =newgoal-use-widget>
          isa use-widget
          target =object
   =newgoal-register>
          isa register-widgets
   !push! =newgoal-register
   !push! =newgoal-use-widget)
```

```
(p finished-exploring-in-persistent
   =goal>
          isa explore
   =screen>
          isa screen
          current t
          type persistent
   =object>
          isa widget
          screen =screen
        - goto nil
   =closer>
          isa widget
          screen =screen
          kind closer
          goto =
   ==>
   =newgoal>
          isa use-widget
          target =closer
   !push! =newgoal)
```

Figure 5. Productions for exploring unexplored widgets and for closing the screen.

causing further depth-first browsing. The production will continue to fire repeatedly as long as there are unexplored widgets on the current screen, after which point, a new productive will become valid. The production to the right in Figure 5 chooses a widget that will close the current window once all other widgets on the current window have been explored.

The productions to move and click the mouse on transitional widgets are straightforward. Each widget chunk in declarative memory has a corresponding location in the visual array. Moving the cursor requires the agent to recognize that the mouse cursor's location in the visual array is not the same as the widget's location in the visual array. A command is sent to the motor module to move the hand – which is holding the mouse – so that the mouse cursor's location is that of the widget's position. Clicking on the widget, likewise, requires a production to send a command to the motor module to click the mouse button. Once the widget has been clicked upon, the same production that initiated the click command must create a new chunk to represent the new screen that appears. Depending on what kind of widget has been clicked on, the new screen could be a window or dialog (a persistent screen), or a menu (a transient screen). Different productions are required for each possibility. Aside from the type of screen chunk that is added to declarative memory, there is no other difference between the production that clicks on a menu widget and the production that clicks on a regular string widget.

Altogether, the model contains 42 productions of the types discussed above. Working memory is initially loaded with 22 chunks representing widget and screen types, vocabulary, and goals. At the end of the exploration of the application shown in Figure 2, over 700 items have been processed by working memory.

Limitations

The current exploration system suffers from a number of limitations of varying severity. The first limitation is that its cognitive model's productions were designed with the ``typical'' Windows application in mind. The typical Windows application uses a standard set of widgets and obeys certain conventions. Applications such as Adobe Photoshop could not be explored with the current system because of its rampant use of non-standard widgets, non-standard toolbars, and the use of multiple open documents. Even running in a typical application, the exploration model is limited by its reliance on conventions and on its built-in vocabulary. If the conventions, such as the use of ellipses, are violated, the model will fail to recognize transitional widgets. If the model's vocabulary is used in unanticipated ways, the model will see transitional widgets where there are none.

The exploration model relies heavily on its understanding of conventions and vocabulary, at least partly because the current implementation is unable to distinguish an entirely new screen from an existing screen that has just changed. Often clicking on widgets in the screen will cause the screen features to change. New widgets could appear or text within the screen could change radically enough that the model might not be able to find a sufficient number of similarities. The flip side of this limitation is that if a child window is spawned, it might be too similar to the previous screen to be regarded as a new screen. In order to simplify the domain, the model tries to identify those widgets that will cause transitions to other screens. We believe that making that determination is easier than determining which screen we are in at any given time, given that a screen can be highly dynamic. To determine which widgets are transitional widgets requires the system to possess more background knowledge however, and the sufficiency of the background knowledge is limited by the designer's ability to predict new situations.

For similar reasons, our approach also cannot handle tabbed dialogs and other interaction scenarios in which clicking a widget causes a change in the state of a window without generating an entirely new window.

A more general limitation arises from the *tabula rasa* flavor of this approach. With no information about the specific domain of an application, the system cannot take actions that lead to states in which different sets of operations are appropriate. A simple example of this limitation can be seen if we consider Cut or Copy operations: these are only active if some object is selected in the application. Cut and Copy are immediate rather than transitional operations, but they have analogs, for example, in operations that allow the modification of object properties. In the tasks to which we have put the exploration system, it has generally worked in an open application with no external document or other information loaded. To be able to handle operations such as Cut and Copy, the model would require an understanding what it means to perform Cut and Copy operations and when they are applicable. In the future, the agent will allow such information to be provided during initialization. In fact, the declarative background knowledge should be customizable to any level of expertise.

A different but equally general limitation lies in the stability of the graphical user interface as a domain in which autonomous agents, whether based on cognitive models or not, can operate. If the domain is highly unstable, then the agent is likely to mistake changes in the Windows desktop that are unrelated to the operation of the target application as being significant. Furthermore, if unrelated elements in the Windows desktop occlude part of the target application's interface, the agent can become lost or confused. When the agent becomes confused or lost, the ACT-R production system on which it is built will halt because the cognitive model will have entered a state in which no productions are valid.

Finally there are a few shortcomings in the specific implementation we have built. Our system is still relatively fragile, a proof of concept rather than a working tool. Because of vocabulary and image processing limitations, the system currently performs a single-level traversal of arbitrary applications (i.e., examining the contents of all top-level menus) completely, but with only selective exploration of dialogs arising from some menu items. Thus the only application that the system has explored in detail is the example application shown in Figure 2. We will soon have performed partial exploration of Notepad, Internet Explorer, PowerPoint, and XEmacs. The implementation problems are not conceptual, but more a matter of bookkeeping, and we expect to be able to address them in the short term. As another part of our continuing development we are porting the implementation to the most recent version of ACT-R, 5.0, from its current use of ACT-R 4.0 and the perceptual-motor extension, ACT-R/PM 1.0b5.

DISCUSSION

Given a system such the one we have described, we face two questions: what kind of results can be generated, and how should they be used?

The exploration process can provide summaries of the navigational structure of an interface. For example, for our sample application, if we treat the menu structure as a graph to be traversed, with a closed application as the starting node at depth 0, then we can produce summaries such as the following:

Maximum depth:	4
Mean depth:	2.45
Maximum transitions:	6
Mean transitions:	2.3

For example, the main screen of the example application has six transitions (menu headers) and leads through a menu selection to a Print Setup dialog to a Print Options dialog at the deepest point. From this kind of summary we might identify unusual outliers or unexpected averages when comparing screens within the application and between similar applications. How depth and number of transitions affects usability is highly dependent on the type of application and the degree of expertise of the user. However, if an application is known to have good usability, its metrics can be compared to the metrics of other similar applications as a means of determining how these metrics relate to the usability of a certain class of applications.

We can also generate a map of an application from which paths from the starting screen to specific target strings can be derived. This path information can be exported so that other agents can be made aware of the layout of the application. The user might ask of another computer agent where functionality associated with the specific string, “Play sounds,” can be found in the application. The traversal of the map produces a path through the Properties menu. Another possibility, not yet possible in our system, is suggested by research in automated evaluation of the visual layout of interfaces [14, 16]. In this case, layout evaluation would not be based on absolute metrics, but rather on relative comparisons for consistency.

Our current work on the system falls into two areas. First, we are improving the robustness and generality of the implementation; we believe that it will eventually be possible to apply the system to arbitrary applications to produce detailed analyses. Second, we are examining relationships to the literature on automated interface generation. Given the data produced by the navigation process, it should be possible to construct a structured representation of the temporal and spatial changes that can occur within an application in the graphical user interface, such as a grammar. We believe that such a grammar could motivate the construction of more robust production

rules that make fewer assumptions about the interface itself. Such grammar-based production rules might also allow us to reduce the number of limitations because we would be able to better predict how a widget will modify the screen, allowing us to end our reliance on the distinction between transitional widgets and non-transitional widgets.

The behaviors displayed by the current system, and those of related systems [17, 18], suggest a direction of growing interest to cognitive modeling researchers: the evaluation of off-the-shelf interactive applications by modeling techniques. A wide range of results have been produced by using cognitive models to evaluate different aspects of computer applications, on and off the desktop. Tasks have included menu selection [2], dialing cellular telephones while driving [12], and flying aircraft [13], among many others. In most cases, however, even for models that include perception and motor components [3, 7, 10], the models have access either to the internals of the application or to its interface. This raises the question, in some cases, whether plausible assumptions are made about the transfer of information between the model and the environment, rather than respecting known constraints on human visual or motor processing [1]. The system described in this paper is the first we know of that performs an automatic evaluation (under the limitations discussed in the previous section) of an independently developed system strictly from the user's perspective on the interface.

ACKNOWLEGEMENTS

This effort was supported by the National Science Foundation under award 0083281, by the Space and Naval Warfare Systems Center, San Diego, and by NFS Career Award 0092586. The information in this paper does not necessarily reflect the position or policies of the U.S. government, and no official endorsement should be inferred.

REFERENCES

1. Anderson, J. & Lebiere, C. *The Atomic Components of Thought*. Lawrence Erlbaum, Mahwah, NJ, 1998.
2. Byrne, M.D. ACT-R/PM and menu selection: applying a cognitive architecture to HCI. *International Journal of Human-Computer Studies*, 55 (2001), 41-84.
3. Byrne, M.D., Anderson, J.R., Douglass, S., & Matessa, M. Eye tracking the visual search of click-down menus. In *Proceedings of CHI'99* (Pittsburgh PA, May 1999), ACM Press, 402-409.
4. Catledge, L.D. & Pitkow, J.E. Characterizing browsing strategies in the World-Wide Web. *Computer Networks and ISDN Systems*, 27,6 (1995), 1065-1073.
5. Dix, A.J., Finlay, J.E., Abowd, G.D., Beale, R. *Human-Computer Interation, 2nd ed.* Prentice Hall, 1998.
6. Erran, C., Crawford, S., & Chen, H. Browsing in hypertext: a cognitive study. *IEEE Transations on Systems, Man, and Cybernetics*, 22, 5 (1992), 865-883.
7. Hornof, A.J., Kieras, D.E. Cognitive modeling demonstrates how people use anticipated location knowledge on menu items. In *Proceedings of CHI'99* (Pittsburgh PA, May 1999), ACM Press, 410-417.
8. Kieras, D. & Meyer, D.E. An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Human-Computer Interaction*, 12, 4 (1997), 391-438.
9. Newell, A. *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA, 1990.
10. Ritter, F.E., Baxter, G.D., Jones, G., & Young R.M. Supporting cognitive models as users. *ACM Transactions on Computer-Human Interaction*, 7, 2 (2000), 141-173.
11. Ritter, F.E., Van Rooy, D., St. Amant, R. A user modeling design tool for comparing interfaces. Under review.
12. Salvucci D.D. & Macuga, K.L. Predicting the effects of cell-phone dialing on driver performance. In *Proceedings of the 4th International Conference on Cognitive Modeling* (Fairfax VA, July 2001), 25-30.
13. Shoppek, W., Holt, R.W., Diez, M.S., & Boehm-Davis, D.A. Modeling behavior in complex and dymanic situations – the examples of flying an automated aircraft. In Proceedings of the 4th International Conference on Cognitive Modeling (Fairfax VA, July 2001), 265-266.
14. Sears, A. Layout appropriateness: a metric for evaluation user interface widget layout. *IEEE Transactions on Software Engineering*, 19, 7 (1993), 707-719.
15. Spence, R. A framework for navigation. *International Journal of Human-Computer Studies*, 51 (1999), 919-945.
16. St. Amant, R. Navigation and planning in a mixed-initiative user interface. In *Proceedings of the 14th National Conference on Artificial Intelligence* (Providence RI, July 1997), AAAI Press, 64-69.
17. St. Amant, R. & Riedl, M.O. A perception/action substrate for cognitive modeling in HCI. *International Journal of Human-Computer Studies*, 55, 1 (2001), 15-39.
18. St. Amant, R. & Zettlemoyer, L.S. The user interface as an agent environment. In Proceedings of the 4th International Conference on Autonomous Agents (Barcelona Spain, June 2000), 483-49.