# The user interface as an agent environment

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

Luke S. Zettlemoyer
Department of Computer Science
North Carolina State University
EGRC-CSC Box 7534
Raleigh, NC 27695-7534
lszettle@eos.ncsu.edu

## ABSTRACT

Theoretically motivated planning systems often make assumptions about their environments, in areas such as the predictability of action effects, static behavior of the environment, and access to state information. We find a striking correspondence between these assumptions and the properties of graphical user interfaces. We have developed a novel type of interface agent, called an ibot, to exploit these correspondences. Ibots can interact with off-the-shelf applications through the user interface rather than programmatically, gaining access to functionality not readily available to artificial agents by other means. In this paper we describe the relationship between these agents and the theoretical and heuristic properties of user interfaces. We demonstrate the feasibility of our approach to interface agents with an implemented prototype that interacts with an unmodified application for graphical illustration.

## 1. INTRODUCTION

As the sophistication of interactive applications has grown, so has the complexity of problems we attempt to solve with them. Enter agents. *Interface agents* are designed to help with problems that are too large, complex, or mundane for users to solve alone [22, 21]. This paper focuses on a class of interface agent we call interface softbots, or *ibots,* [39, 40]. Unlike the current generation of interface agents, an ibot controls an interactive system through its graphical user interface, as human users do, without relying on an application programming interface (API) or access to source code. We have developed a programmable substrate that provides sensors, effectors, and skeleton controllers for this interaction. Sensor modules take pixel-level input from the display, run the data through image processing algorithms, and build a structured representation of visible interface objects. Effector modules generate mouse and keyboard gestures to manipulate these objects. These sensors and effectors act as the eyes and hands of an artificial user, controlled by a planning component. Together the sensor, effector and controller components provide a general-purpose means of managing interactive applications, through the same medium as a real user.

We have two practical motivations for building ibots. First, in modern graphical user interfaces we often encounter what we might call "visually incompetent" interface agents: one agent blinks and fidgets in the corner of the screen, drawing attention away from the user's work; another raises dialog boxes that obscure important information on the screen; another offers suggestions for straightforward actions that it is unable to carry out itself. It is clear that such agents have little conception of the complex visual environment they share with users, and their contribution to the interaction suffers for it. By working directly in this environment, an ibot has no choice but to leverage its capabilities and respect its limitations. The other point is a development issue for agents that assist in the use of problem-solving tools. Commercial applications are tailored to interaction with human users, and often do not accommodate the activities of an agent-based assistant through an application programming interface. To provide a foundation for the contribution of the assistant, developers commonly build research systems that replicate much of the basic problem-solving functionality of off-the-shelf interactive software, simply because it is otherwise inaccessible through programmatic means. Ibots can solve this problem by integrating their behavior into existing, familiar environments.

We also have strong research motivations for exploring the possibilities of ibots. Cognitive modeling researchers have long treated the user interface as a useful surrogate for the real world. Their work has been hampered, however, by the operational distance between cognitive simulations and real user interfaces [5, 4]. We have recently developed a low-level model based on ibot technology that bridges this gap. The system contains a coarse computational model of vision, from low-level region segmentation to high-level object recognition, and a rudimentary model of motor behavior. In our current work the system is helping us to extend existing cognitive models to different interface domains and to test their ecological validity on real-world applications.

The second research motivation provides the focus for this

paper. We have observed a striking similarity between the assumptions planning systems commonly make about their environments, on the one hand, and the properties imposed on an environment by interface design guidelines, on the other [32]. Control is one example. If the environment can change only through an agent's actions, and remains static otherwise, problem solving is facilitated significantly. This desirable environmental property is reflected in the classical planning assumption of the absence of exogenous events, those that can occur outside the agent's volition. It is similarly reflected in the HCI guideline that the conventional interface must not initiate actions but rather respond to the user's commands as a tool [6, 37]. This ideal of user/planner control is so well-entrenched that in both HCI and planning the terms "action" and "event" are used interchangeably [15].

This is just one of many such correspondences. These relationships suggest that the user interface could play the role of a general-purpose testbed for planning agents: planners often abstract away the continuous, uncertain, dynamic, and unobservable properties of an environment, such that it becomes discrete, deterministic, static, and accessible—properties associated with broad classes of modern graphical user interfaces. Given the sensors and effectors provided by ibot technology, and an appropriate set of domain-specific operators, a planning agent might gain access to a great deal of existing functionality now available to users but generally not considered accessible to agents.

The goal of this paper is to make a detailed argument for the feasibility and appropriateness of the user interface as an agent environment. (For conciseness, throughout this paper we refer to the interactive software environment of an interface agent—operating system interface, applications, utilities—generically as the user interface.) In past work we have informally discussed the construction and application of ibots; here we put a more detailed foundation under this work. In the next section we describe the user interface as an agent environment, giving an overview of formal and heuristic models of user interfaces and their consistency with environmental properties that simplify agent behavior. The section that follows describes a simple hierarchical planning architecture for ibot agents that relies on these observations, and an implemented prototype that uses an unmodified application, Adobe Illustrator, to create and manipulate formatted text and simple graphics. We end with a brief discussion of connections between this paper and work in interaction and agency.

# 2. THE USER INTERFACE AS AN AGENT ENVIRONMENT

Human-computer interaction research and artificial intelligence research have close historical ties, going back at least as far as Newell and Simon's *Human Problem Solving* [25]. We see this relationship, for example, in HCI problem space representations based on goals and actions [17], and the notion of the user as a rational problem-solving agent [9]. Here we lay out some of the core properties of modern user interfaces and show how they correspond to familiar constraints on the search space of actions and plans for an agent in its environment. Our discussion draws from a broad literature on modeling user interaction, in particular taxonomies of design guidelines [6, 16, 30] and formal mathematical models of user interaction [11, 12].

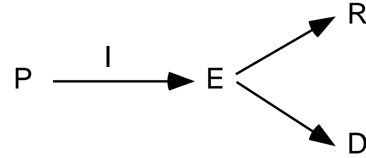


**Figure 1: The red-PIE model**

## 2.1 Formal models of the interface

One of the most extensively analyzed formal models of user interfaces is Dix's PIE model [11, 12]. A user interface is modeled as a triple $(P, I, E)$, where $P$ is a set of sequences (programs) whose elements are taken from some finite set of commands $C$; $E$ is the effects space, the external results of executing programs; $I : P \rightarrow E$ is the interpretation function, which represents system computation. In other words, the user's commands are viewed as a stream of discrete inputs to the model. Each input causes some internal state change, or interpretation of the command, and the result appears as the effect of this interpretation. The red-PIE extends the basic PIE model by separating the result $R$, which may involve only internal state changes, from the external display $D$, as shown in Figure 1.

In each of the paragraphs below we describe a specific property or refinement of the model and explain how it is reflected in real user interfaces and the design guidelines intended to produce them. We then discuss the implications for planning agents and their interaction with the interface.

It should be clear that the properties we describe do not apply to all interactive systems. They are intended to facilitate human problem solving for specific types of tasks. Games are a good example of applications that do not fall into the same class of systems. Game design often intentionally imposes barriers to easy use: some information may be hidden from the user; scenes may change too quickly for human reactions; even the goals of playing the game may not be clear. These applications are beyond the scope of our work. Nevertheless our observations will apply to a large class of common productivity applications, and provide a good starting point for agent behavior.

**Discreteness.** The PIE model represents input to an interface as a sequence of discrete values; in search terms we can similarly represent interaction with an interface as the traversal of a discrete state space.

This property is easy to see in real interfaces: keyboard and mouse gestures are discrete, and mouse movements are usually to specific objects or locations. There are exceptions,

such as free-form drawing in graphics applications, but the discreteness property applies to almost all common user actions in large classes of interactive applications.

The same observation extends to the visual properties of the interface. Visual design guidelines promote a highly structured arrangement of discrete objects [23]. Window borders partition information. Interactive objects almost always have distinct borders, setting them apart from the background and each other. By design, interfaces support the efficient recovery of information through their visual structure [36]. Discrete, easily recognized objects are a large part of this efficiency.

**Accessibility.** In an accessible system, state information is directly available to the user. In the PIE formalism, this is managed by defining a function *transparent* between the display $D$ and the result $R$, such that in a fully transparent system, a user can obtain all information about the the effect of an action by applying the transparency function (through some hypothetical mental activity) to the display. Full transparency is feasible, however, only for extremely simple applications, such as a desk calculator without memory. Even in the simplest word processor, for example, not all the text in a document of moderate size can be viewed at once, nor can it be inferred. This problem is addressed by observation strategies. User interfaces are not fully transparent; instead they provide mechanisms for making information observable or inferrable. Scrolling makes offscreen text visible; a paste action (followed by an undo) shows the contents of the cut/paste buffer; preference settings become visible with a menu selection.

One of the basic design guidelines for direct manipulation systems is to make relevant objects and actions continuously visible [30]. In other words, a good interface makes available all information relevant to the decision-making process.

The implication for agent behavior is that information relevant to a given decision will almost always be accessible in the current state in a well-designed user interface. If it is not immediately acessible it will be available through the application of one of a small number of observation strategies. This limits the amount of inference required of an agent to build an internal model of its environment.

**Stasis.** A static interactive system does not change simply with the passage of time, but instead only when the user takes a non-null action in it [29]. The PIE model is defined such that the only changes to the interface are responses to input from $P$, and that input consists only of actions initiated by the user. After taking an action, the user waits until the system reaches quiescence before taking another. This leads to Dix's description of modern user interfaces as "event in, status out" [11, p. 242].

Two design guidelines promote the steady state interface: perceived stability and user control. The perceived stability guideline entails that the user should be able to depend on stability (e.g., visual layout) as time passes [6]. Further, many designers hold that users should exercise nearly complete control over the user interface; this is one of the defining characteristics of direct manipulation [30]. In practice, this degree of control and stability is not always present. Exogenous events may occur in some circumstances. For example, some interfaces will dynamically produce a notification on the arrival of mail or the change in the status of a network connection. (In applications for collaborative work, as with games, exogenous events are integral to their functionality.) Nevertheless in many applications event occurrences are of well-defined types and can be dealt with by simple, fixed strategies, as with accessing state information; often such events can be safely ignored.

In planning terms, this simply means that the agent controls state transitions. Exogenous events are rare and can be handled in a straightforward way.

**Determinism.** PIEs are explicitly deterministic; $I$ is defined to return a single value for each input from $P$. Dix identifies a number of scenarios for which a non-deterministic function $I_{ND}$ may be appropriate (e.g., to model unpredictable system timing in returning results), but yet again these can be abstracted away by simple strategies (e.g., waiting and polling the system until a response occurs to account for timing variation.) From a design perspective, this issue is reflected in a variety of guidelines, most clearly in the property of consistency, especially procedural consistency [6]. Consistency can be described in terms of matching users' expectations: the same action taken under a well-defined set of conditions should always produce the same effect.

The implication for agent behavior is that actions with deterministic effects can be effective in this environment. Tied to the accessibility and stasis properties of interfaces, an interface agent can expect to have full state information at the execution of an action, and that its effect will be predictable.

## 2.2 Heuristic design guidelines

In addition to mathematical models of the interface, a large number of less formal approaches attempt to capture properties of the interaction of users with the interface. HCI researchers, most notably those who build task analysis models, have worked with practitioners to produce a rich set of guidelines that impose structure on user interaction. These guidelines can also inform the design of an interface agent in significant ways.

**Hierarchical problem decomposition.** User interfaces rely heavily on a divide and conquer approach to problem solving. For example, in a graphical application of any complexity, the user will activate menu items to create information structures, manipulate their properties, and so forth. Some of these menu selections will not result in an immediate effect. Instead the user will be shown a dialog box with more information about the desired option. The user completes the task, relying on local information and actions, and then returns to the main interface. One guideline associated with this style of problem solving is that tasks should

be separated to allow their independent execution [16]. This facilitates incremental problem solving in which one task is brought to completion before the next is begun.

**Bottom-up problem solving.** Hix and Hartson, in their discussion of designing interactive systems that support hierarchical task execution, advocate that goals be satisfiable in bottom-up fashion, with intermediate solutions constructed and combined opportunistically at higher levels [16]. This can be seen in the prescription of non-modal behavior for user interfaces. For example, in a drawing application graphical objects can be selected in any order for a group operation, and in a word processor the typeface and size properties of text can be modified independently, in either order. In most applications that allow wide user latitute in selecting operations (wizards and some drawing applications being notable exceptions), the system does not enter into states that require a strict sequential ordering of solution components.

**Short solution length.** Hix and Hartson further emphasize the importance of giving users frequent closure on tasks [16]. This means that users should never encounter long sequences of actions toward a goal without some indications of progress along the way.

These design guidelines have useful and interrelated implications for agent interaction. Planning will be effective, especially hierarchical planning, if the plans of the agent can map to the behavior of the interface environment. Subgoal interactions will be minimal. The downward solution property will generally hold: solutions generated at an abstract level will decompose to solutions at the primitive level; goals in the interface will be independent, or at least block serializable [20]. All of these points, along with those in the previous section, suggest that the user interface is a tractable environment for a planning agent. Even given the strong restrictions on plan representation in planning research today [34, 35], and the recognized complexities of modern user interfaces, we might expect an agent to be able to solve problems in such an environment.

## 3. AN IBOT ARCHITECTURE

We have developed an interface agent architecture that exploits the properties of the user interface environment, in two parts. At the higher level we have a controller based on a hierarchical planning extension to UCPOP, a conventional partial-order planner [26].[1] The lower level comprises a set of image processing and event management modules that support direct interaction between plan operators and user interface widgets. We call this latter subsystem VisMap, for "visual manipulation." The architecture is shown in Figure 2.

[1]Using UCPOP was a matter of programming convenience; we needed a Lisp-based planner that could reliably handle universal quantification in preconditions and effects. Unfortunately, no Graphplan-based system [7], such as SGP [3], currently meets these requirements.

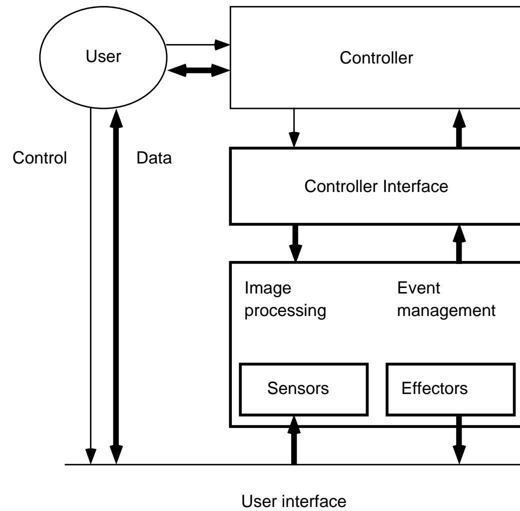


**Figure 2: An ibot architecture**

### 3.1 The controller level

UCPOP acts as the core planner in a very simple hierarchical plan generation/execution system; it is difficult to think of a simpler (or more limited) extension of partial order planning to hierarchical planning. The extended planner processes decomposition operators in addition to standard planning operators. A decomposition specifies an action to be decomposed, its parameters, a goal form, and a domain.

In plan generation, the core planner is invoked as usual to generate a "flat" plan to satisfy a top level goal. When a plan has been generated, its actions are executed in the interface, and its effects are used to update an internally maintained state representation that tracks the interface. The execution of actions in the interface is governed by procedural methods. For example, one common interface action is moving to a menu header icon, such as the Edit menu. The action `(move-to-action Edit-header)` is tied to a procedure that looks up the symbol Edit-header in a table to retrieve the string "Edit", then searches for that string in an internally maintained representation of text visible on the screen. Screen locations and other data are retrieved similarly.

Some actions are associated with decomposition operators. When such an action is encountered during plan execution, the core planner is reinvoked with the goal and domain specified by the decomposition. The system recursively generates and executes plans across abstraction levels (and domains) until the planning process is complete. Thus, for example, the selection of interface actions, such as the Edit menu icon or the Cut menu item, is distinct from the selection of objects, such as a box in a drawing package, each task associated with different properties. More than one object can be selected at one time, but only one action; action selection is not persistent. In the top level domain, only abstract object and action selection operators are represented. The execution of each such operator results in a recursive call to the

```
(define (domain UI-top-level-domain)
  (:action select-object
     :parameters (?object)
     :precondition (screen-object ?object)
     :effect
     (and (:forall (?other)
             (when (neq ?other ?object)
                (not (object-selected ?other))))
          (object-selected ?object)))
  (:decomposition select-object-decomposition
     :action select-object
     :parameters (?object)
     :goal (object-selected ?object)
     :domain UI-object-domain)
  (:action select-action
     :parameters (?action)
     :precondition (action ?action)
     :effect
     (:forall (?object)
         (when (object-selected ?object)
            (action-on-object ?action ?object))))
     . . .)


(define (problem create-header)
  (:domain UI-top-level-domain)
  (:goal (and (action-on-object Font... text)
             (action-on-object Size-24pt text)))
  ;; Symbol bindings within controller interface
  (:init ((application application)
          (object-selected NULL-OBJECT)
          (pointer-over START-POSITION)
          (text text)
          (position text-position)
          (visible Type)
           (control-object TEXT-MODE)
          (cascaded Type Size)
          (cascaded Size Size-24pt)
          (action Size-24pt)
          (cascaded Type Font)
          (cascaded Font Font-Braggadocio)
          (action Font-Braggadocio)
          . . .)))
```

**Figure 3: Interface domain and problem statement**

core planner with the appropriate domain. At each of these recursive calls, the system reprocesses the information on the screen. Because of the static and deterministic properties of the interface, this turns out to be the only time (aside from ubiquitous menu selection operations) that the system needs to resynchronize its internal representation with the interface.

A small number of critics locally refine the plan as it executes. In the current system, the only action possible to a critic is a mouse movement, to restore a state after a subgoal has been satisfied and the hierarchical process returns.

Let's consider an actual example, for concreteness. Part of our top-level domain for action in the user interface is shown in Figure 3. This domain is one of several; the operators in all domains include the following (decomposition operators marked with an asterisk): raise application, select object*, drag object, grow object*, add select object, select action*, enter text*, move to object, move object handle, select visible object, select control object, move to action, move to position, select visible action, cascade, select mode*, activate text entry, type string, deactivate text entry, set parameter, complete dialog.

```
> (hierarchical-plan :problem 'create-header)
.   .   .   enter-text
   move-to-position ((190 .  100))
   select-mode
      move-to-object (text-mode)
      select-control-object (text-mode)
   move-to-position ((190 .  100)) *
   activate-text-entry ((190 .  100))
   type-string ("The Ibot Report")
   deactivate-text-entry ("The Ibot Report")
select-action
   move-to-action ("Type")
   cascade ("Type" "Size")
   move-to-action ("Size")
   cascade ("Size" "24pt")
   move-to-action ("24pt")
   select-visible-action ("24pt")
select-action
   move-to-action ("Type")
   cascade ("Type" "Font")
   move-to-action ("Font")
   cascade ("Font" "Braggadocio")
   move-to-action ("Braggadocio")
   select-visible-action ("Braggadocio")
```

**Figure 4: Solution trace**

The sequence of events is as follows. Given the goal of generating a string of text, "The Ibot Report," in a specific location on a page within Adobe Illustrator, the system identifies the drawing region and the starting point for the text. It moves the mouse pointer to the text toolbar icon and selects it; it then moves to the starting point and enters the string. The goal also includes a specific typeface and and font size for the text. This is accomplished by appropriate selections from the "Type" menu. The solution generated by the planner is given in Figure 4.

At this point the system stops. The user examines the result, decides that the look is not exactly right, and changes both the typeface and the size of the text. The system is reactivated at this point with a new goal, that of putting an appropriately sized box to the left of the text. By analyzing the altered text, the system can make the desired change; the system is able to both control the application and observe its state to make incremental changes (e.g., growing objects or moving them around.) The final result is shown in Figure 5.

As can be seen from the solution trace, these are straightforward procedures, though not trivial. The planner can carry out a variety of larger tasks with comparable complexity in this interface. The approach we have outlined, however, would be extremely naive for problem solving in general, in the real world. In many ways the controller is similar to and perhaps even less sophisticated than microworld prob-

Figure 5: Illustrator results

lem solvers. As we have argued, however, there is strong evidence that current planning technology will be adequate for problem solving in the user interface. Deterministic operators are not a difficulty because of the predictability of the environment. Uncertainty about state information is not an issue because of the environment's accessibility properties. The constrained hierarchical structure of action in the interface means that the planner can solve problems without considering complex subgoal interactions. Finally, we are incorporating a modern planner into the system to replace the current core planner. Modern planners outpace UCPOP by one or two orders of magnitude [35], and thus we expect scalability not to be an issue.

## 3.2 The interface manipulation level

The VisMap substrate is responsible for handling mouse and keyboard manipulations as well as image processing of screen information, at the programmatic interface with primitive plan operators.

VisMap contains an event management module that acts as a simple motor system. It inserts events into the operating system's event queue, the result being indistinguishable from the actions of the user. These events include `move-mouse`, `mouse-down`, `mouse-up`, `key-down`, and `key-up`. By combining sequences of these events and focusing their locations to specific visual instances, the module realizes the primitive operators passed to it by the planner, for selecting icons, clicking buttons, pulling down menus, and so forth.

Complementing the effector module is an image processing module, which acts as a coarse, three-level model of vision. In the low-level stage, the module performs edge detection and a degenerate form of motion analysis to segment the screen into regions. In the intermediate stage, feature descriptions are attached to each region that describe its internal structure and relationships with other regions. In the high-level stage, a restricted form of object recognition is performed. This stage identifies sets of segmented regions based on their feature descriptions as high-level objects or specific visual patterns. These provide state information for the planner and necessary guidance for event management.

The image processing module contains 29 feature computation functions and 80 interpretation rules of the types shown

```
Operation GetArea()
   MaxPixels = GetWidth() * GetHeight()
   Area = ActualPixels()/MaxPixels
   Return Area


If object Obj is a downArrow()
   and Obj is containedIn() object RB
     such that RB is a raisedButton()
       and RB is toTheRightOf() object RTA
         such that RTA is a rectangularTextArea()
           and RTA is recessed()
               and has width() > height()
   Then Obj is a component of a dropBox
```

**Figure 6: A feature computation and an interpretation rule**

in Figure 6. The system recognizes all the common user interface controls in the Windows user interface: buttons, scroll bars (including the scroll box, scroll arrows, and background regions), list boxes, menu items, check boxes, radio buttons and application windows. The system also performs a simple pattern matching form of optical character recognition for the standard Windows typeface, which allows for text processing.

# 4. RELATED WORK

Our approach to agent/interface interaction can be somewhat unintuitive; a software engineer might ask, "Why do you bother with the visual interface—screen scraping—when you could go through an API?" Several answers are possible [33, 39], but the most important is that we are interested in understanding the complexities of the interaction of an agent with an environment, including motor, perceptual, and cognitive issues. Working at the user interface level instead of with APIs, we we can directly address interaction issues, we gain significant generality across applications, and many of our findings will have practical and theoretical implications for both interface agents and human users [32]. The user interface acts as a rich and yet manageable laboratory in which we can examine realistic problems identical to those faced by human users. This has made it an attractive domain for other researchers in intelligent user interfaces, agent-based systems, and cognitive modeling.

In programming by demonstration, or PBD, a system records actions performed by the user and infers a generalized program that can be later used in analogous situations [10]. TRIGGERS was an early PBD effort that explored the possibility of using the visible user interface for input [27]. The user defines condition/action pairs, called triggers, by stepping through a sequence of actions in an application, adding annotations when necessary. At each step TRIGGERS performs pattern matching on screen pixels to infer information that is otherwise unavailable to a non-embedded agent. By generalizing from visible changes to the state of the interface, the system creates programs that can perform such visually-oriented tasks as adding converting text to a bold typeface, surrounding a text field with a rounded rectangle in a drawing program, and comparable activities.

While TRIGGERS concentrated on taking input from the interface, other agents have tackled the complementary problem of controlling their external environment. Lieberman's agents, for example, control applications with operating system events that act as metaphorical marionette strings [21]. Lieberman's work has also identified some of the important design issues involved in this approach, which include the granularity of event protocols, appropriate styles of interaction with the user, and user/agent control considerations.

TRIGGERS and related systems embody the idea that the visible interface is a powerful source of information for an agent, if it can be properly interpreted [33]. Cognitive modeling researchers have adopted the same idea, but from a different perspective, in the form of programmable user models [28, 38]. PUMs are engineering models of human users; they can generate empirical data on how visual information is processed in simulated and real application interfaces. This concentration on the interaction of a PUM (a cognitively plausible agent) with the visual properties of a user interface has driven a number of recent advances in computational cognitive modeling [4, 19].

Perhaps the most interesting connection is to work on interaction and agency [1]. One of our central concerns is with what Agre calls the convergence method for agent design, a concentration on the mutual relationship between the agent and its environment rather than only on its internal processing [2]. Our exploitation of the properties of the interface is reminiscent of Kirsh's observations about how the efficient use of space can reduce search [18], and especially Hammond et al.'s techniques for environment stabilization [14]. Rather than relying on extensive memory and processing capabilities, an agent might actively modify (stabilize) the environment such that its properties facilitate problem solving. In the case of the user interface, some of the techniques discussed above could be considered forms of pre-stabilization. Of course, the key to stabilization is the agent's direct actions at runtime, rather than the environment designer's efforts; more of the flavor of stabilization is captured by the layer-by-layer artifact construction and customizable toolbars supported by some software. Ibots are not yet able to use interactive applications as cognitive tools in this way, as people do, but this is an important direction for our work.

## 5. CONCLUSION

To recap, we have described a type of agent that interacts directly with the user interface, and identified a number of ways that constraints on agent processing are accommodated by theoretical and practical properties of the interface. We have implemented a planner that exploits this relationship and demonstrated its feasibility with a small example in an unmodified commercial application. Other ibot domains include an agent that plays a competent game of Microsoft Solitaire, an application with non-standard icons, inaccessible source code, and no API [39]. We have also developed a simple tool for usability analysis [40], a research tool for novel interface gestures [13], and an assistant for graphical design.

Our planner and substrate are works in progress and still have significant limitations. The substrate can only process characters in a single typeface, for example, which limits its breadth. While it recognizes all standard interface widget types, actually using them is another issue. Those that require tight feedback, such as for scrolling through a document, are not yet handled well. The recognition and manipulation of application-specific objects (e.g., imported images in a drawing package) is limited, and the system has no ability to learn new patterns.

Two other important limitations, at the planner level, deal with user interaction and the development of domains to match complex applications. There is no shared control with the user, strictly speaking, in this framework; the planner is invoked with some goal, runs to completion, and can then be reinvoked. An ideal system would operate in a mixed-initiative fashion, with control shifting flexibly between the user and the planner [8, 24, 31]. Domain development is another important point: the planner is only as effective as its operators, and these are limited in the current system to symbol manipulations. For more sophisticated behavior, especially in drawing applications, some form of spatial reasoning will be necessary.

Addressing all of these concerns is part of our current work.

## 6. ACKNOWLEDGMENTS

This paper benefited greatly from the comments of two anonymous reviewers.

## 7. REFERENCES


[1] P. Agre and S. Rosenschein, editors. *Computational Theories of Interaction and Agency*. MIT Press, 1996.

[2] P. E. Agre. Computational research on interaction and agency. *Artificial Intelligence*, 73(1–52), 1995.

[3] C. Anderson, D. Weld, and D. Smith. Conditional effects in graphplan. In *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, 1998.

[4] J. Anderson and C. Lebiere. *The Atomic Components of Thought*. Lawrence Erlbaum, 1998.

[5] J. R. Anderson, M. Matessa, and S. Douglass. The ACT-R theory and visual attention. In *Proceedings of the Seventeenth Anual Conference of the Cognitive Science Society*, pages 61–65, Hillsdale, NJ, 1995. Lawrence Erlbaum Associates.

[6] Apple Computer. *Macintosh Human Interface Guidelines*. Apple Computer, Inc., 1992.

[7] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[8] M. H. Burstein and D. V. McDermott. Issues in the development of human-computer mixed initiative planning. In B. Gorayska and J. L. Mey, editors,

*Cognitive Technology: In Search of a Humane Interface*, pages 285–303. Elsevier Science, 1996.

[9] S. K. Card, T. P. Moran, and A. Newell. *The psychology of human-computer interaction*. Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.

[10] A. Cypher, editor. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.

[11] A. J. Dix. *Formal Methods for Interactive Systems*. Academic Press, San Diego, 1993.

[12] A. J. Dix, J. E. Finlay, G. D. Abowd, and R. Beale. *Human-Computer Interaction*. Prentice Hall, 2nd edition, 1998.

[13] M. S. Dulberg, R. St. Amant, and L. Zettlemoyer. An imprecise mouse gesture for the fast activation of controls. In *INTERACT '99*, pages 375–382, 1999.

[14] K. J. Hammond, T. M. Converse, and J. W. Grass. The stabilization of environments. *Artificial Intelligence*, 72(305–327), 1995.

[15] J. Hendler, A. Tate, and M. Drummond. Ai planning: Systems and techniques. *AI Magazine*, pages 61–77, Summer 1990.

[16] D. Hix and H. R. Hartson. *Developing User Interfaces*. John Wiley & Sons, New York, 1993.

[17] D. E. Kieras. Towards a practical GOMS model methodology for user interface design. In M. Helander, editor, *Handbook of Human-Computer Interaction*, pages 135–157. North-Holland, 1988.

[18] D. Kirsh. The intelligent use of space. *Artificial Intelligence*, 73(31–68), 1995.

[19] M. Kitajima and P. G. Polson. A comprehension-based model of exploration. *Human-Computer Interaction*, 12(4):345–389, 1997.

[20] R. E. Korf. Planning as search: A quantitative approach. *Artificial Intelligence*, 33:65–88, 1987.

[21] H. Lieberman. Integrating user interface agents with conventional applications. In *Proceedings of the Fourth International Conference on Intelligent User Interfaces*, 1998.

[22] P. Maes. Agents that reduce work and information overload. *Communications of the ACM*, 37(7):31–40, July 1994.

[23] K. Mullet and D. Sano. *Designing Visual Interfaces: Communication Oriented Techniques*. Sunsoft Press, 1995.

[24] K. L. Myers. Abductive completion of plan sketches. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 867–893. AAAI Press, 1997.

[25] A. Newell and H. Simon. *Human Problem Solving*. Prentice Hall, 1972.

[26] J. Penberthy and D. S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of the Third International Conference on Knowledge Representation and Reasoning*, pages 103–114. Morgan Kaufmann Publishers, Inc., 1992.

[27] R. Potter. Triggers: Guiding automation with pixels to achieve data access. In A. Cypher, editor, *Watch What I Do: Programming by Demonstration*, pages 361–382. MIT Press, 1993.

[28] C. Runciman and N. Hammond. User programs: A way to match computer systems and human cognition. In *Proceedings of the HCI'86 Conference on People and Computers II*, pages 464–481, 1986.

[29] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Inc., 1995.

[30] B. Shneiderman. *Designing the user interface: strategies for effective human-computer interaction*. Addison-Wesley Publishing Company, 1998.

[31] R. St. Amant. Navigation and planning in a mixed-initiative user interface. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence*, pages 64–69. AAAI Press, 1997.

[32] R. St. Amant. User interface affordances in a planning representation. *Human Computer Interaction*, 14(3):317–354, 1999.

[33] R. St. Amant, H. Lieberman, R. Potter, and L. S. Zettlemoyer. Visual generalization in programming by example. *Communications of the ACM*, March 2000. To appear.

[34] D. S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, Winter 1994.

[35] D. S. Weld. Recent advances in ai planning. *AI Magazine*, 1999. To appear.

[36] D. D. Woods. The cognitive engineering of problem representations. In G. R. S. Weir and J. L. Alty, editors, *Human Computer Interaction and Complex Systems*, pages 169–188. Academic Press, 1991.

[37] D. D. Woods and E. M. Roth. Cognitive systems engineering. In M. Helander, editor, *Handbook of Human-Computer Interaction*, pages 3–43. North-Holland, 1988.

[38] R. M. Young, T. R. G. Green, and T. Simon. Programmable user models for predictive evaluation of interface designs. In *Proceedings of the CHI'89 Conference*, pages 15–19, 1989.

[39] L. Zettlemoyer and R. St. Amant. A visual medium for programmatic control of interactive applications. In *CHI '99 (ACM Conference on Human Factors in Computing)*, pages 199–206, 1999.

[40] L. Zettlemoyer, R. St. Amant, and M. S. Dulberg. Ibots: Agent control through the user interface. In *Proceedings of the Fifth International Conference on Intelligent User Interfaces*, pages 31–37, 1999.