Int. J. Human-Computer Studies (2000) 000–000

A perception/action substrate for cognitive modeling in HCI

ROBERT ST. AMANT and MARK O. RIEDL
Department of Computer Science, North Carolina State University,
EGRC-CSC Box 7534, Raleigh, NC 27695-7534,
email: stamant@csc.ncsu.edu, moriedl@eos.ncsu.edu

(Received 00 Jan 0000, and in final form 00 Jan 0000)

This article describes a general-purpose programmable substrate designed to allow cognitive modeling systems to interact with off-the-shelf interactive applications. The substrate, called VisMap, improves on conventional approaches, in which a cognitive model interacts with a hand-constructed abstraction, an artificial simulation, or an interface tailored specifically to a modeling system. VisMap can be used to construct static scenarios for input to a cognitive model, without requiring its internal modification; alternatively, the system can be integrated with a cognitive model to support direct control of an application.

1. Introduction

Research in human-computer interaction (HCI) has traditionally held close, beneficial ties to cognitive modeling research. Cognitive modeling research drove some of the earliest rigorous examinations of user behavior, providing insight and guidance in areas such as the effects of practice on performance, rational decision-making, and expert problem-solving in the user interface [Card *et al.*, 1983]. Conversely, HCI research has given cognitive modelers challenging problems in realistic domains and environments, in which solutions have useful theoretical and practical implications [Alterman *et al.*, 1998; Gray *et al.*, 1993]. In some research efforts, the distinction between HCI research and cognitive modeling research has blurred almost to the point of disappearing [Barnard and May, 1999; Kieras and Meyer, 1997; Kitajima and Polson, 1995; Kitajima and Polson, 1997].

The mutual relationship between these two fields has arisen for more than historical reasons. As a surrogate for the real world, the user interface provides a simplified, more tractable environment in which interesting problems can still be posed and solved. In fact, the properties of a tractable environment for a cognitive model correspond strikingly with the properties of graphical user interfaces. Cognitive modeling research has been strongly influenced by Newell and Simon's view of human problem solving as search through a problem space [Newell and Simon, 1972]. Problem spaces traditionally abstract away continuous, non-deterministic, 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 [St. Amant, 1999].

In view of such correspondences, we might expect cognitive models to routinely interact with the user interfaces of off-the-shelf applications. Perhaps surprisingly, this is not yet the case. For some cognitive modeling systems, visual input is generated via the look-up of properties in a static, hand-constructed interface specification, e.g., through a file interface [Kitajima and Polson, 1997]. Other models interact with dynamic simulations of interfaces, constructed to mimic the behavior of a real interface, but tailored to the input and output requirements of the model [Kieras, 1999]. Still other models interact directly with user interfaces, but under significant restrictions, which commonly involve extending a user interface management system to generate feature-based or object-based representations that mirror the interactive display and are appropriate as input and output for a cognitive model [Anderson and Lebiere, 1998; Ritter *et al.*, 2000].

A different approach, with comparable goals, is to arrange for a cognitive model to interact directly with applications through their user interfaces. The main drawback is that the model must now address a new set of complexities dealing with visual processing, object manipulation, action planning, and so forth, at a greater level of detail than required by the approaches above. The potential advantages of extending a cognitive model in this way, however, are compelling:

- *Ecological validity.* By their nature, interface simulations and specifications are abstractions; they do away with unimportant details of a real interface. Real user interfaces exhibit variation in timing, variation in the predictability and reliability of actions, and the occurrence of events uninitiated by the user, among other behavioral properties, which may or may not be relevant to performance on a given task. Neglecting these subtleties, however, can bring the validity of empirical cognitive modeling results into question. We can forestall potential objections along these lines by reducing the distance between cognitive models and real environments. If researchers can deal with a real user interface instead of a simulation or specification, they can dispense with an additional layer of indirection in the evaluation of the performance of a cognitive model.
- *Real-world problem relevance.* Cognitive modeling goals usually include a concern for solving problems of general interest, rather than focusing solely on problems for model calibration or benchmarking. Real user interfaces easily provide real-world problems; additional design and maintenance effort is required to replicate such problems in a simulation or specification.
- *External standards for comparison.* Progress in cognitive modeling often arises from comparisons of different models on the same problem. In some cases (e.g., the performance of EPIC and ACT-R/PM on Nilsen's menu selection task [Hornof and Kieras, 1999; Byrne *et al.*, 1999]) it is straightforward to determine whether two instances of a problem are directly comparable. As researchers take on more complex problems in richer environments, however, this determination can be more difficult, especially if the tested interfaces differ significantly (say, an active simulation in one case and a symbolic specification in another.) This difficulty can be isolated if environmental information and interactions stem from a single independent source, a real user interface.
- *Development effort.* In some cognitive modeling efforts, considerable work is devoted to developing realistic user interface scenarios, either as static representations or device simulations; ironically, much of this effort reproduces functionality that already exists in a form appropriate for human users, but is inaccessible (i.e., programmatically) to a cognitive model. Peter Polson has observed that automatic processing to generate realistic interface scenarios for model input and validation would be a significant step forward in cognitive modeling research [Polson, personal communication].

We have developed a practical approach to visual processing and manual interaction for cognitive models in HCI, based on a novel type of interface agent that we call an interface softbot [Zettlemoyer and St. Amant, 1999; Zettlemoyer *et al.*, 1999]. An interface softbot, unlike the current generation of interface agents, controls an interactive system through the graphical user interface, as human users do, without relying on an application programming interface (API) or access to source code. To support this functionality in the Microsoft Windows environment we have developed a programmable substrate that we call VisMap, for "visual manipulation." In VisMap, a sensor module takes pixel-level input from the display, runs the data through image processing algorithms, and builds a structured representation of visible interface objects. An effector module generates mouse and keyboard gestures to manipulate these objects. The sensor and effector act as the eyes and hands of an artificial user, to be controlled externally by a computational cognitive model.

VisMap is not a cognitive model in itself. Rather, it is a suite of functions that allow an existing cognitive model, without extensive internal modification, to interact directly with off-the-shelf applications. The system is designed to be broadly compatible with common assumptions made in cognitive modeling, but in cases where practical functionality conflicts with cognitive plausibility, the practical direction is usually chosen. We believe that the implications of this system for cognitive modeling research are significant. VisMap should allow modeling researchers to more easily develop realistic input scenarios and evaluate the ecological validity of models with respect to real-world applications. We hope that in general VisMap will support more researchers in treating cognitive modeling as an exploratory research tool, expanding the current boundaries of experimental practice.

The body of this article fills out this brief description. Section 2 discusses several cognitive modeling systems that interact in different ways with a user interface environment. Each model incorporates perception and motor components that interact with a modified user interface or a programmatic intermediary. This indirect interaction facilitates development but also includes potential pitfalls for researchers. Section 3 describes VisMap's direct interaction with the user interface, concentrating on the tractability of the interface in comparison with the physical, dynamic, three-dimensional world. Interactive applications, by design, are well-suited to the properties of computational vision and action. Section 4 discusses the design of VisMap's visual and motor components. These components are significantly limited with regard to cognitive plausibility; nevertheless their design is broadly consistent with current approaches to modeling perception and action in HCI. Section 5 gives two examples of VisMap in practice. In the first, the system is used to generate a symbolic specification that describes a sequence of actions in an application. The result is appropriate for static input to a cognitive modeling system. In the second example, VisMap carries out a simple sequence of actions, directed by ACT-R/PM in a loosely coupled integration. In both examples the emphasis is not on the correctness of the model (or of VisMap itself) with regard to cognitive processing, but rather on the novel interaction capabilities that the integration enables. Section 6 closes with a brief discussion of the implementation status of the system, related modeling tools under development, and future work.

2. Indirect interaction with the interface

Our research has been influenced by work in visual and to some extent motor processing for cognitive modeling in HCI. This work spans a number of distinct research areas: comprehensive, general purpose cognitive models (such as Soar [Newell, 1990], ACT-R [Anderson and Lebiere, 1998], and others), tools to support task analysis, and programmable user models and their application to user interface design and evaluation [Runciman and Hammond, 1986; Young *et al.*, 1989].

Most modern cognitive modeling systems for HCI incorporate some form of visual and motor processing, tailored to interaction with the user interface. In this section we review a representative sample of these systems, limiting our consideration to those that simulate behavior, rather than model it in abstract terms (cf. interactive cognitive subsystems [Barnard and May, 1999].) For each of the models, we describe its general aims and scope, the level of detail at which the environment and interaction with the environment are represented, and the general engineering approach to making environmental information available to the model. All of these models have been under active development for several years, and many have passed through different version and even name changes; our description reflects their current status to the best of our knowledge.

LICAI, a linked model of comprehension-based action planning and instruction taking [Kitajima and Polson, 1995; Kitajima and Polson, 1997], simulates the execution of exploratory tasks in applications with graphical user interfaces. LICAI has the stated goal of modeling behavior in environments that dynamically change in response to user actions [Kitajima and Polson, 1995]. Nevertheless, LICAI adopts a static specification approach to interaction with the environment. The environment is described by a set of logical predicates that specify the identity, properties, and display status of display objects. For example, the predicates that represent an icon in the menu bar titled “Graph”, used to activate a pull down menu, are as follows:

```
(is-on-screen $object-123)
(is-a-kind-of $object-123 display-object)
(is-a-kind-of $object-123 graph-menu-item)
(is-pointed-at $object-123)
(not (is-highlighted $object-123))
(not (is-grabbed $object-123)).
```

The first three predicates describe the object itself. The latter three predicates describe the display status of the object in an interactive context, for a situation in which the model has moved the mouse pointer over the menu header but has not selected it. Other visual properties, such as location, shape, size, and color, may also be represented, as well as relationships between objects, such as part-whole relationships.

Actions in the model are an elaboration of modern STRIPS-style operators, which involve an action name, a precondition, and an effect [Fikes and Nilsson, 1971; Penberthy and Weld, 1992]. Preconditions and effects are conjunctions of predicates that represent, respectively, constraints on the execution of an action and the state of the environment after the action has been carried out. The representation also includes information about a display object and the intended function of the physical action; thus a physical action such as *Mouse-move-cursor* can be specialized, in a sense, to the action *Point-at Graph in menu-bar*. Other examples of basic physical actions include the following:

Move-mouse-cursor: Move the mouse cursor to a specific location.

Single-click, Double-click: Single or double click the mouse button.

Press-and-hold-mouse-button-down, Release-mouse-button: A more primitive decomposition of mouse button actions.

Type: Enter a sequence of characters via the keyboard.

The static specification of environmental information integrates easily into the action representation, but at some development cost. Generating a sufficiently detailed specification involves a good deal of time and effort. Further, experimental scenarios must be planned with great care: a specification must provide the appropriate environmental responses to actions selected by the model, proceeding in lock step through the sequence. While LICAI developers have produced impressive results, specification by hand imposes a significant barrier on development.

GLEAN (GOMS Language Evaluation and Analysis) is a tool for GOMS simulation [Kieras, 1999]. Models are built in GOMSL, or GOMS Language, and represent the knowledge users apply in carrying out tasks on devices. GLEAN acts as an interpreter for GOMSL by processing and executing these models in a simulated interactive environment. Although aimed at task analysis, GLEAN nevertheless represents much of the state of the art in the interaction between cognitive models and their environments.

GLEAN performs visual processing of its environment, but not down to the level of pattern recognition. Instead, visual inputs are represented as symbolic objects associated with lists of attribute-value pairs, which may include location, size, color, and other properties. A red "Start" button in a user interface, for example, is represented as follows:

```
Visual object:  Start-button
    Type is Button
    Label is Start
    Color is Red
```

If such an object is visible on the screen and in focus in the GLEAN simulation's working memory, then the properties of that object become available to GOMS methods. An attentional mechanism brings objects into focus one at a time, where focusing can be as simple as searching through visual working memory until an object with appropriate features is found. GLEAN provides both manual and visual actions, as follows:

Keystroke: This manual operation generates a typed character from the keyboard.
Type-in: This manual operation enters a sequence of keyboard characters.
Hold-down, Release: These manual operations generate lower-level keyboard events.
Click, Double-click: These manual operations generate mouse button events.
Point-to: This manual operation moves the mouse cursor to an object.
Home-to: This manual operation moves the hand to its position over the keyboard.
Look-for-object-with-property-and-store. . . : This is a visual operation that scans the display for an object with a specific property value and associates it with a named tag.
Wait-for-object-with-property-and-store. . . : This visual operator behaves comparably.

In contrast to LICAI, in which detailed specifications of an interface are built by hand, GLEAN takes input from visual inputs generated by device simulations. A device simulation provides all the information required by a GOMS method in the course of executing a GOMSL program. In the current version of GLEAN, device simulations are coded in C++ and compiled together with the system. While this allows great flexibility, it can also be time consuming, and so GLEAN provides an alternative by which models can be executed without device simulations. Instead, auxiliary information can be associated with the model itself, an approach that suffices for some types of static environmental information that need not change in response to user actions.

EPIC (Executive-Process/Interactive Control) is a prominent example of a detailed cognitive model of low-level interaction with a computer interface [Kieras and Meyer, 1997]. EPIC supports the construction of models that allow accurate and detailed evaluation of human perceptual/motor performance. EPIC does not process at the pattern recognition level, but instead deals with symbolic objects. Physical visual objects represent location, size, color, and other properties. These physical objects are made available to the cognitive simulation as psychological visual objects, which include such properties as position on the retina.

The EPIC cognitive architecture is considerably more sophisticated than that of GLEAN, but shares some of its properties at the level of environment interaction. EPIC's manual and visual operators, for example, are very detailed: instead of simple movement, EPIC allows styles of movement, which include punch, point, pose, peck, ply, and patter; visual operations include move, fixate, preposition, and prepare. Both systems nevertheless use the same general mechanisms for interacting with their environments. Our discussion of GLEAN's properties in this regard thus applies equally to EPIC.

The ACT-R visual interface[1] uses a more direct approach to environmental interaction [Anderson and Lebiere, 1998]. ACT-R is a theory of the nature of human knowledge, implemented in a production system. The visual interface integrates different theories of attention into a framework that encompasses both cognitive and perceptual processes. Research with the visual interface, both theoretical and empirical, has emphasized the processing of alphanumeric characters. A character object in the environment is represented as an object with specific properties, as follows:

```
(Letter-E
    isa Abstract-Letter
    value ‘‘E’’
    line-pos ...)
```

In the default optimizing mode of the visual interface, processing is object-based, under the assumption of perfect visual recognition ability. Cognitive operators have direct access to objects such as the one above, and characters and strings are automatically processed with no chance for error. In the non-optimizing mode, however, processing is feature-based, rather than object-based. Characters are represented in an LED form that contains sixteen line segments, which constitute distinct visual features that must be processed for recognition. Once recognized, objects—their identity as well as their associated features—are available as chunks in the higher-level cognitive process.

The implementation of the visual interface for ACT-R/PM is based on the interface construction facility built into Macintosh Common Lisp. The modeler constructs an interface in the MCL interface toolkit, and then translation routines automatically build iconic representations for interface objects such as characters, strings, and other types of object. Operations that access and manipulate these representations include the following:

Press-key: This manual operation types a keyboard key.

Move-mouse, Click-mouse: These manual operations move the mouse cursor and generate button events.

Find-location: This visual operation finds screen locations based on a specification that may be a screen position, a description of a display object, or a list of the features of an object.

Move-attention: This visual operation shifts attention to different locations on the screen.

Ritter et al.'s Sim-eyes and Sim-hands [Ritter *et al.*, 2000] take the conceptual approach of ACT-R/PM a step farther, by embedding perception and action in the user interface. The Sim-eye and Sim-hand systems are based on a powerful, general simulation architecture for environment interaction. Instead of environment specifications or device simulations that run independently of a user interface environment, Ritter et al. advocate a scheme in which a user interface management system is extended to incorporate a simulated sensor/effector compo-

[1] By “visual interface” we mean to encompass both ACT-R Visual and its successor ACT-R/PM.

nent. This approach provides a number of development benefits: it can lead to more accurate models that are less costly to develop; it reduces inadvertent model-task dependencies; it supports experimentation across models and across tasks at a low development effort. The Cognitive Model Interface Management System has been used to simulate perception and motor activity for the Garnet user interface management system [Myers *et al.*, 1990] and ACT-R, and the Tcl/Tk interface to Soar 7, demonstrating the generality of the work.

We describe all of these approaches and their relatives as being *indirect*; the information available to a cognitive model does not arise directly from the visual display. Instead, some input specifications are constructed by hand; others are constructed automatically, but based on data structures internal to the use interface, rather than on the visual representations that result from these structures. In other words, no direct computational process exists between the information visible in the environment and the input actually processed by the model. When evaluating cognitive models that take input indirectly, rather than from a real interface, researchers must keep in mind that the relevance of experimental results depends on the accuracy of the simulation or specification. Anderson and Lebiere describe this issue in terms of "stresses" on cognitive modeling research [Anderson and Lebiere, 1998], which can arise in a number of ways, including the following:

Missing object and object feature information. A cognitive process might plausibly include a sequence that involves moving to an OK button to confirm an action; an interface specification or simulation would include the necessary descriptive information for the action. In a real environment, however, one often finds multiple occurrences of the same object, distinguishable by location and the surrounding visual context. An OK button might belong to a dialog box in the currently active application, or to an application in the background. Abstracting away such distractors simplifies the task for a cognitive model.

Missing classes of event and object information. In some cases, a simulated interface or specification may dispense with entire classes of information. Timing is a common example. Cognitive models sometimes assume that the sequential relationships between events in the environment are relevant, but not their duration. This approach can run into problems in a real user interface—how long should a model wait until the environment responds to an action, or returns to quiescence? Clicking on a menu header and waiting for the menu to appear may take less than a second, for example, but waiting for an application to start may take much longer; sometimes undetected events can cause the interface never to respond at all. (Dix gives a general, theoretical discussion and describes some of the practical implications of this and related interface modeling issues [Dix, 1993].)

Inconsistent object information. By simulating or reproducing an environment, even at a high degree of detail, rather than interacting directly with it, artifacts can be introduced into the object recognition process. In the ACT-R visual interface, for example, characters are represented in an LED representation, which cannot capture curved segments or some types of diagonal segments. Letters such as "A", "B", and "C", must be represented by horizontal and vertical segments only. This is a minor example in comparison with the potential differences between static object specifications, or device simulations, and actual interface objects.

The significance of these examples is not that cognitive models should be extended to handle all classes of information, at all levels of detail. On the contrary, abstractions are necessary, to limit development effort and to allow for the valid comparison of disparate models. Rather, the point is that if one has design control over both a cognitive model and the environment in which it is evaluated, questions of ecological validity can escape one's attention; building a specification of an interface, or an interface designed for a specific cognitive modeling experiment, one is parsimonious with one's effort, implementing only those factors judged relevant to the problem at hand. Working with real interfaces, as discussed in the following section, forces explicit consideration of how to handle the subtleties of interaction with a sometimes complex environment. Addressing these issues directly has the additional benefit of improving opportunities for exploration, possibly allowing novel, unexpected relationships to be identified.

3. Direct interaction with the interface

VisMap stands in contrast to the systems described in Section 2 by interacting directly with the user interface. VisMap is designed to be situated between a cognitive model and the user interface. From the point of view of the user interface of an application (or interactive utility, or the operating system), VisMap is a set of functions for passive observation and active control. From the perspective of a cognitive model, VisMap is essentially invisible: it maps the primitive perception and motor operations of the model, such as *find-object-with-features*, *move-mouse-to-location*, *press-key*, and so forth, to these observation and control functions.

Being to some extent independent of both the user interface and a cognitive model, VisMap must depend for its generality on common properties of cognitive models and some kinds of task analysis models. For VisMap's design we make some assumptions about a hypothetical generic cognitive modeling simulation, assumptions that reflect most of the relevant environment interaction properties of the models discussed above. Identifying these assumptions also helps to separate the functionality expected of VisMap from that of the cognitive model that controls it.

The generic model represents attentional processing, but does not extend to pattern recognition. It relies on an iconic representation of objects in the environment, in which graphical icons may be accessed by identity or at the object feature level. Features include type, location, size, and related geometrical properties. Modeling emphasis, however, is on character and text processing; text icons may be accessed by direct matching. The generic model supports only a few types of motor actions: pressing keys on the keyboard, pressing the mouse button, and moving the mouse pointer. These last actions can be tied to the results of visual processing, so that objects can be manipulated. Finally, perception and action occur in a serial process. This generic model does not incorporate features of more sophisticated, existing cognitive models, in particular multiple modalities, parallel processing, and detailed visual and motor decompositions. Nevertheless, significant functionality is present, enough to motivate one of VisMap's central design goals: to make relevant interface information and appropriate interface operators available to this generic model.

Even for such a restricted model, it might seem that we have set ourselves overly ambitious goals for perception and action in VisMap. We know of no artificial vision systems that can reliably produce arbitrary iconic representations from the physical, three-dimensional world. Robots, even in domains with simple motor requirements, can have trouble dealing with the dynamics and instability of real environments. Fortunately, we can rely on significant constraints on the complexity of the user interface, imposed by explicit standards and common design conventions.

Although the user interface is designed to match human perceptual abilities, it is a vastly simpler environment than the physical world: it is discrete, rectilinear, and two-dimensional. Unlike the real world, almost all objects in the interface are clearly delineated from their background. Buttons and other controls are rectangular, with lined borders and shadows. Text is drawn in contrasting colors. Three-dimensional interfaces are still a rarity; in a conventional interface, obscured information can simply be brought to the foreground for inspection, with little search or inference required.

Further, the interface is highly structured. For example, related items are grouped and aligned vertically or horizontally, as we see in palettes, toolbars, and fill-in forms. Window borders partition information. Icons and text strings are often drawn from a small, fixed set, for immediate visual recognition. By design, interfaces support the efficient recovery of information through their visual structure [Woods, 1991]. The interface is also heavily annotated. Simple inspection is often enough to establish the correct interpretation of objects in the environment. Buttons and menu items are labeled with their functions. Visual highlighting indicates that a type-in field has the current focus, or that a specific button is the default action. The active window has a border with a distinctive color. When implicit structure is insufficient for correct interpretation of information, explicit cues are supplied.

Comparable constraints apply to action in the user interface. User interfaces are designed to simplify and facilitate the successful execution of actions. Design guidelines include the following [Apple Computer, 1992; Gentner and Nielsen, 1996; St. Amant, 1999; Shneiderman, 1998; Woods and Roth, 1988]:

- *User control:* The interface should not initiate actions, but rather respond like a tool.
- *Consistency:* A user action under well-defined conditions should always produce the same effect.
- *Perceived stability:* The interface should remain stable (e.g., in visual layout) as time passes.
- *Continuous representation:* Objects and actions of interest should remain continuously visible.

These guidelines promote the development of interfaces in which simple motor actions are generally reliable, without the need for complex dependencies between actions for handling failures and unexpected contingencies.

All of these factors contribute to the feasibility of effective computational vision and action in the interface.

4. VisMap

As an integrated system, VisMap is strongly influenced by techniques for image processing [Gonzales and Woods, 1992] and artificial intelligence planning [Hendler *et al.*, 1990]. Figure 1 shows the basic architecture. The bulk of the processing in VisMap is devoted to visual processing and motor behavior, as discussed below.

4.1. VISUAL PERCEPTION IN VISMAP

The goal of the visual component of VisMap is as follows. Input to the system is the raw contents of the display, at a pixel level; output is a description of the high-level user interface components. Processing must be application- and domain-independent, and must follow the general functional outline of biological visual processing. Our work draws on the

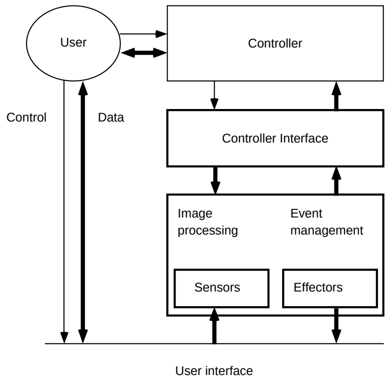

FIGURE 1. Generic ibot architecture

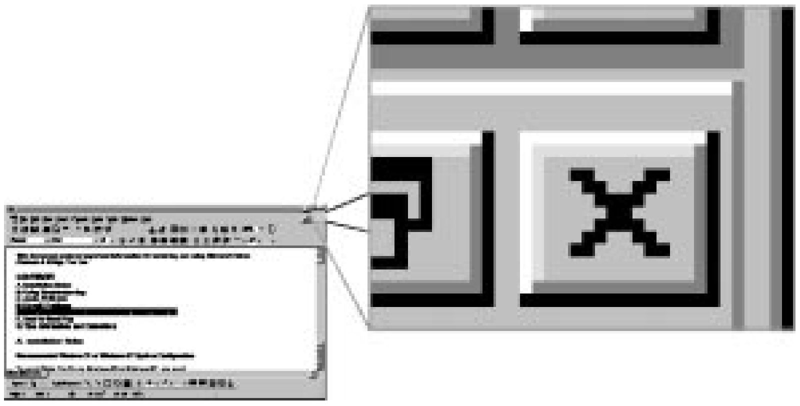

FIGURE 2. Source data for vision system

computational account of high-level vision proposed by Ullman [Ullman, 1996], and an implementation of many of these ideas by Chapman [Chapman, 1991]. As will be clear from the discussion below, our work also adopts two conventional assumptions in vision modeling research [Marr, 1982]. First, despite the considerable differences between biological systems and computer systems at a hardware level, we can nevertheless develop computational processes that are faithful to biological processes at a functional or task level. This assumption is also common in most computational cognitive modeling. Second, vision is not a single, indivisible process but rather a set of stages in which information is incrementally processed to produce increasingly refined representations of the environment.

4.1.1. *Low-level processing*

In biological systems, low-level vision is responsible for analysis of color, depth, motion, shape, and texture of objects. In VisMap, the goal for low-level processing is transform the raw data, a two-dimensional array of colored pixels, as shown in Figure 2, into a set of distinct regions.

Operationally this is a straightforward image segmentation problem [Gonzales and Woods, 1992]. The process begins with a screen capture, which records the color values of each pixel

```
GrowGroupFrom (location)
   Set groupColor = color of pixel at location
   Add point at location to ExpansionList
   while (length of ExpansionList > 0)
      ExpandAbout (location)

ExpandAbout (location)
   Increment pixelCount
   For each neighboringPixel around location
      If (color of neighboringPixel = GroupColor)
         Call ComputeType (NeighboringPixel)
         if (NeighboringPixel isa BorderType)
            Add NeighboringPixel to BorderList
         Call UpdateBoundingBox()
         Add NeighboringPixel to ExpansionList
```

FIGURE 3. Region segmentation algorithm

on the screen. Region segmentation is the result of the pixel grouping algorithm shown in pseudo-code in Figure 3. The segmentation algorithm allows pixels to be activated in any order, growing from a single starting point, and pixel values can 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. In the resulting representation, if two pixels are adjacent (with eight-neighbor connectivity) and have the same color, then they are in the same region, and region membership is associative. The result is internally homogeneous, often rectilinear regions of pixels.

A special case of motion analysis is also handled at this level: changes to the environment, such as the appearance and disappearance of windows, the updating of text, and so forth. This is not the more complex task of object tracking (the movement, appearance, and disappearance of objects), which overlaps to some extent with the object recognition process and is partially handled during a later process. Motion analysis here covers only the basic recognition of change. The implementation works through a bitwise comparison, assisted by standard graphics hardware. On the triggering of such a screen change event, the segmentation algorithm breaks the affected area into appropriate regions.

Low-level processing in VisMap differs in many important ways from more cognitively plausible accounts of vision. For example, some processes, such as edge detection, are trivial, here because there are few meaningful gradients of color or intensity in the user interface, and regions and boundaries tend to have distinct and consistent coloring. Many of the results that might be expected from a complete vision system, such as interpretation of differences in texture, are missing. The entire process is serial, rather than parallel. Three-dimensionality, including stereo processing, and continuous object movement are neglected almost entirely. The two processes nevertheless share some important abstract properties. First, results at the low-level stage are comparable, though considerably simplified in the case of the computational system. Second, both processes are bottom-up, relying on the image data rather than any knowledge and goals derived from specific tasks. Third, low-level processing in VisMap implicitly reflects a view of attention as selective processing, in particular early selection [Allport, 1989]. The segmentation algorithm can be "focused" on a specific starting point and a given region, limiting further processing at a higher level. Alternatively, a late selection form

```
Width:
    region.lowerRightX() - region.upperLeftX() + 1

Height:
    region.lowerRightY() - region.upperLeftY() + 1

Area:
    MaxPossibleNumPixels = Width() * Height()
    area = ActualNumPixels() / MaxPossibleNumPixels

ContainedIn:
    inner.upperLeftX() ≥ outer.upperLeftX() and
    inner.upperLeftY() ≥ outer.upperLeftY() and
    inner.lowerRightX() ≤ outer.lowerRightX() and
    inner.lowerRightY() ≤ outer.lowerRightY()
```

FIGURE 4. Examples of intermediate features

of attention can be managed, as part of high-level processing. In general, however, modeling the details of attention is not a part of VisMap; this is considered the responsibility of an external cognitive model.

4.1.2. *High-level processing*

In biological systems, high-level vision applies the results of low-level vision to accomplish tasks such as object recognition and classification, visually guided manipulation, navigation through an environment, and other activities. Our concern will be with object recognition of icons that commonly appear in the user interface. Object recognition in VisMap occurs at an intermediate and a high-level stage of visual processing. The main distinction between the two stages is the reliance at the high level on knowledge about objects, rather than general physical knowledge such as spatial relationships used at the intermediate level [Ullman, 1996].

At an intermediate level, visual features and relationships are derived from the constructs produced during the low-level stage. Intermediate-level functions and relationships execute in bottom-up fashion, relying solely on the information acquired from the low-level pixel grouping process. To streamline the complexities of high-level visual processing, segmented regions are bounded rectangularly. Since most graphical interface components are rectilinear, this approximation is acceptable, though a more accurate method could be implemented if needed. Representative examples are given in Figure 4. In these examples, spatial properties refer to the bounding box of a region.

Some number of auxiliary functions are defined at this level that have no real cognitive counterparts; they are aimed at practicality rather than cognitive plausibility. An example is *ActualNumPixels* in Figure 4. As VisMap matures, dependence on these will be reduced.

At the high-level processing stage, objects are recognized, drawing on information produced during the intermediate and low-level stages. Rules are defined to represent structural relationships between regions; each rule corresponds to a different type of interface object. Each rule can be considered a specialized implementation of a template for an object. For an object on the screen to be recognized, there must exist a template which describes how specific components must be present and in the correct relative positions.

```
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 5. A description rule for drop boxes

For example, the template rule in Figure 5 finds drop boxes. Other templates identify right angles, rectangles, circles, check marks, up, down, left, and right triangles; these contribute toward composite templates that identify buttons, windows, check boxes, radio button, list boxes, vertical and horizontal scroll bars, the entire alphabet for a single typeface, plus more specialized objects. As the sample template rule shows, high-level templates make calls to pre-defined intermediate-level region operations. Successive applications of such sets of templates detect and identify the visible controls in the interface. The existing set of templates in the system was developed opportunistically to provide the functionality needed to identify all of the most common interface controls.

The result of the high-level processing stage is a set of meaningful interface objects that become available to a cognitive model layered on top of the substrate. Non-meaningful shapes (as defined implicitly by our heuristic rules) are deemed part of the irrelevant background. Note that among these non-meaningful shapes may be partially occluded windows, widgets, and text; in general, if an object is not entirely visible, it may not be recognized. In the real, three-dimensional world, this limitation would be catastrophic. In the user interface, however, where for the most part only fully-visible objects in the foreground are of interest, the limitation does not prevent effective action.

As with the low-level stage, the intermediate and high-level processes reflect some important properties of biological vision, but also differ significantly from more cognitively plausible descriptions. The differences are most pronounced at the intermediate stage, where there is very little flexibility in the computational process. For example, Hildreth and Ullman, in describing the properties of intermediate-level vision, give the example that it is not feasible to have an “inside/outside” detector at every possible location to test whether a point falls within a contour [Hildreth and Ullman, 1989]; however, this is essentially the purpose feature computations in VisMap serve. At the high-level stage, processing is limited to object recognition, and it is not at all clear how the template-based approach would generalize to other tasks such as manipulation and navigation, or even recognition in three dimensions.

4.1.3. *Cognitive modeling interface*

The operations supported by the system, at the programmatic interface to a cognitive model controller, are as shown below.

Get-widgets: This operation returns a list of all widgets visible on the screen. An optional argument can specify the type of widgets to be returned. All the familiar user interface controls in the Windows user interface can be returned: buttons, scroll bars (including the scroll box, scroll arrows, and background regions), list boxes, menu items, check boxes, radio buttons, and application windows. Depending on the value of a parameter setting, container relationships between widgets are computed.

Find-widget: Given a test of the properties of a widget, this operation returns a specification of the matching objects it finds on the screen. Currently these properties are limited to its type and spatial properties (location, width, height). If no such widget is found, the operation returns an empty list.

Get-pixel-groups, Find-pixel-group: These operations are the equivalent of their widget counterparts. As input they take a rectangular boundary, and return objects representing the pixel groups within that boundary. Given the boundaries of a widget, then, these operations return the pixel groups that constitute that widget. These operations can test for spatial properties as well as for the color of a pixel group.

Get-strings, Find-string: These operations are the text-based equivalent of *get-widgets* and *find-widget*. The first operation returns a list of objects representing all the strings visible on the screen that are separated either vertically or by horizontal white space. The second operation returns objects representing the text and spatial characteristics of a specified string, if such is visible on the screen.

Get-letters: This operation is a more primitive version of get-strings. It returns all the individual letters recognized on the screen, with their location and extent.

Get-screen-objects: This operation combines get-widgets and either get-strings or get-letters, depending on a parameter setting. Depending on the value of a parameter setting, container relationships between objects (widgets and strings) are computed.

Set-bounds: This operation constrains the operations that follow to occur only within the specified rectangular bounds. With this operation a cognitive model can focus processing on a region such as a single window, for example, ignoring objects outside its boundaries.

Wait-for-object: This operation returns when a new object of a specific type has become visible.

On the face of it, this set of operations has an obvious shortcoming: it places too few constraints on the development of a cognitive model. An improved set of operations would include underlying representations of retinal zones (a fovea, parafovea, and so forth, as in EPIC and Sim-eyes), more constraints on tests of visual object features (e.g., modeling the limitations of feature integration, as in ACT-R), and in general a better account of attention, all of which would help prevent the development of too-powerful cognitive models. These operations are thus currently best treated as primitives with which one can construct different plausible controller interfaces.

A rudimentary visual model that addresses these concerns is under development. Our steps in this direction are tentative, however. One of the goals of VisMap is to provide a general-purpose tool for a variety of cognitive models. We face a practical tradeoff between power and generality: a more powerful (i.e., detailed) visual representation may reduce the range of models to which VisMap is applicable, due to conflicts with the existing assumptions of these models; a weaker representation will increase this range, but at the cost of greater modeling effort. That said, the current operations provide some degree of modeling support, in the form of timing metrics.

A duration computation is associated with each operation. For searches at the pixel group level, duration is based on the area to be processed. For finding strings and widgets, duration is based on the number of objects within the region under consideration. This information is also made available to an external cognitive model to improve its internal timing estimates; in other words, VisMap provides information about the visual environment to the model in exchange for more accurate timing assessments for its processing. Because of the concessions

VisMap must make to functionality, its performance in real time often cannot match its duration computations. As VisMap matures in efficiency and representational detail, however, we expect that the default duration estimates (and their actual performance) will more closely approach those of cognitively plausible models.

VisMap incorporates some 30 feature computation functions and 80 interpretation rules of the types given in Figures 4 and 5. Practical limitations exist on many operations, however, in addition to the issues raised above. Visual changes to the mouse cursor, for example, are not detected. Sometimes objects are aligned in such a way that their identically colored boundaries overlap, causing recognition rules that depend on perimeter patterns to fail for those objects. The rules for character recognition are also imperfect. Graphical patterns occasionally cause spurious letters to be returned. Kerning can cause some uppercase letters to touch their neighbors, which cannot be handled by the recognition process. Thus extraneous, missing, and even incorrect letters can result. To partially remedy (but not completely solve) this problem, cost-based string matching is performed for finding specific strings, using the Levenshtein distance metric [Sankoff and Kruskal, 1983]. A longer term solution will involve devising a more robust recognition process, potentially following the lead of Anderson and Lebiere toward general pattern recognition [Anderson and Lebiere, 1998]. Finally, VisMap's recognition of only a single typeface limits its flexibility in detecting differences between selected and unselected windows and some button icons. VisMap is a work in progress; we expect that these limitations can be addressed in the short term.

4.2. MOTOR BEHAVIOR IN VISMAP

Motor behavior in VisMap is an extreme simplification of actual performance, rudimentary but adequate for competent action in the user interface. The motor module concentrates on generating actions that have appropriate durations, but neglects properties of the actions below this level of abstraction. Thus, for example, the duration of a mouse movement follows Fitts' law [Fitts, 1954], but the trajectory of the movement from a starting location to an ending location has no lateral deviation, and its velocity is constant over the course of the movement, without the characteristic initial acceleration and final closed-loop targeting phase [Meyer *et al.*, 1988].

Given relevant information about objects in the interface, the motor module can select icons, click buttons, pull menus down, turn on radio buttons, and carry out other standard, familiar operations. Operationally, these actions are implemented as specialized functions for manipulating the event queue at the operating system level. Its event insertions are indistinguishable from user-generated events. All action in the user interface proceeds from sequences of the primitive events given below. As with visual processing operations, useful higher-level abstractions, such as *click-button*, have been defined. These are handled in a straightforward way as sequences of the more primitive events.

Key-down, Key-up: These operations press and release keys on the keyboard. Their duration is a parameterized constant, set by default at 200ms.

Mouse-up, Mouse-down: These operations press and release the left mouse button. Their duration is a parameterized constant, set by default at 100ms.

Mouse-click, Mouse-double-click: These operations are composed of Mouse-up and Mouse-down operations. Their duration simply sums the durations of their component actions.

Move-to-point: This operation moves the mouse pointer to the location supplied. The duration T of the operation is governed by Fitts' law in the following version [MacKenzie, 1995]: $T = a + b \log_2 (A/W + 1)$. The values of the constants a and b can be specified per task, but are set by default to $a = 230$ and $b = 166$ for pointing with the mouse [MacKenzie, 1995]. The value for the distance A is measured between the current mouse pointer location and the location given; the target width W is set to a nominal constant value.

Move-to-object: This operation retrieves the location property of an object and sets the appropriate parameters for execution of the Move-to-point operation.

Drag-to-point: This operation is comparable to Move-to-point, except that the parameters a and b are set to different values: $a = 135$ and $b = 249$ [MacKenzie, 1995].

The default duration computation for each of these operations is as described. These default computations can be overridden by an external cognitive model interfacing with the motor module. As with our discussion of vision processing, VisMap makes no commitment to specific action decompositions, concentrating instead on low-level functionality, which is straightforward. The motor modeling limitations of the system should be clear. Actions are represented at a much coarser level of detail than in systems such as EPIC, Soar, and ACT-R/PM. Nevertheless their more sophisticated decompositions of movement actions can be built on top of the functions described above, with duration functions modified appropriately.

5. VisMap in practice

This section illustrates the use of VisMap in two different settings. The first example shows how VisMap can be used to produce a sequence of static scenario descriptions, for later input to a cognitive modeling system. In the second example, VisMap is coupled with ACT-R/PM to carry out a simple visual task.

5.1. GENERATING SCENARIOS WITH VISMAP

Some cognitive models expect a static specification of the state of the user interface as input; VisMap can be used to generate such specifications. We demonstrate the generation procedure with a simple visual task. A document called `numbers.text` contains a sequence of words naming the numbers from "one" to "ten". The task is to launch a word processing application, open the document and select the word "five". Figure 6 shows the state of the interface near completion of the task.

This task required only a simple, programmatic controller, designed without concern for cognitive plausibility. Its role is to generate the environmental information on which a hypothetical cognitive model can base its decisions. To do this, it relies on both VisMap primitive operations and knowledge about the structure of the visual display and interaction in the Microsoft Windows user interface.

At each step, the controller generates a description of the current state of the interface, as represented by the objects returned by VisMap. The printed representation of these objects can easily be modified to match the syntactic input requirements of a cognitive model, but in this example is not intended to match any specific model. The analysis below describes each of the controller's steps at two levels of detail: the VisMap operations that execute perception and motor actions in the interface, and the more abstract tasks that these operations constitute. Tasks are annotated with a discussion of the interaction issues VisMap faces, its techniques for addressing potential problems, and some of its specific limitations.

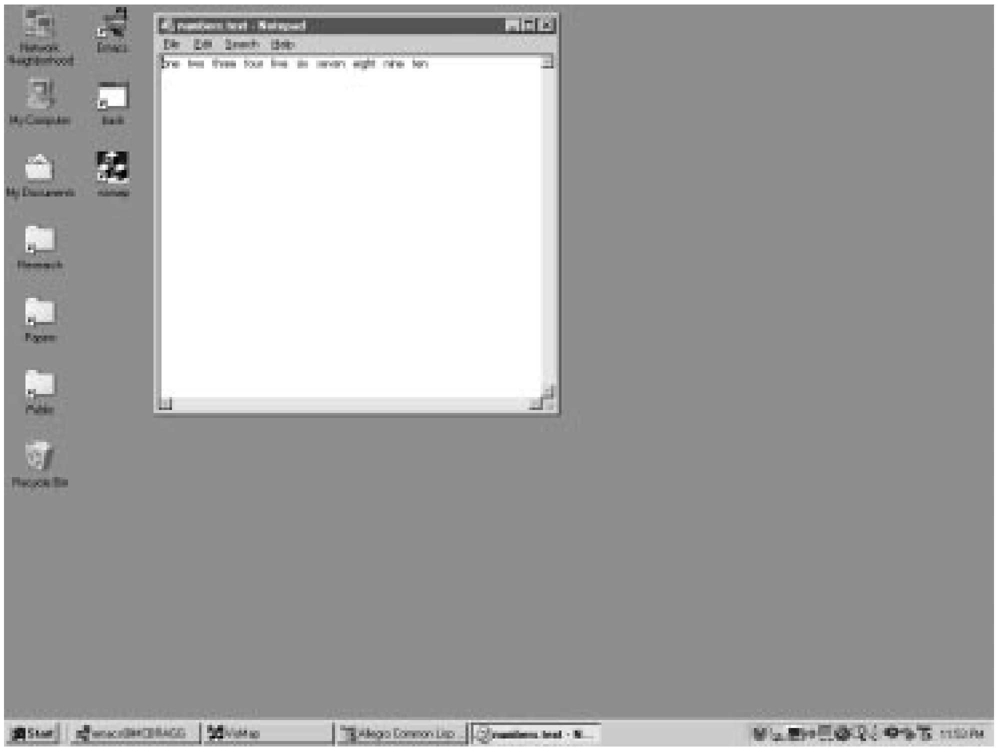

FIGURE 6. The Notepad example environment

```
(MOUSE-POSITION 443 264)
(BUTTON :LEFT 3 :TOP 745 :RIGHT 53 :BOTTOM 763)
(BUTTON :LEFT 554 :TOP 16 :RIGHT 566 :BOTTOM 26)
(BUTTON :LEFT 536 :TOP 16 :RIGHT 548 :BOTTOM 26) ...
(VSCROLL :LEFT 558 :TOP 55 :RIGHT 566 :BOTTOM 404)
(HSCROLL :LEFT 163 :TOP 409 :RIGHT 553 :BOTTOM 553)
(WINDOW :LEFT 157 :TOP 11 :RIGHT 572 :BOTTOM 423) ...
(TEXT :LEFT 370 :TOP 754 :RIGHT 402 :BOTTOM 754 :STRING "AIIEGRO")
(TEXT :LEFT 405 :TOP 754 :RIGHT 446 :BOTTOM 754 :STRING "COMMON")
(TEXT :LEFT 449 :TOP 754 :RIGHT 468 :BOTTOM 754 :STRING "LISP")
(TEXT :LEFT 12 :TOP 495 :RIGHT 51 :BOTTOM 495 :STRING "RECYCIE")
(TEXT :LEFT 54 :TOP 495 :RIGHT 69 :BOTTOM 495 :STRING "BIN") ...
(TEXT :LEFT 166 :TOP 60 :RIGHT 184 :BOTTOM 60 :STRING "ONE")
(TEXT :LEFT 191 :TOP 60 :RIGHT 210 :BOTTOM 60 :STRING "TWO")
(TEXT :LEFT 217 :TOP 60 :RIGHT 243 :BOTTOM 60 :STRING "THREE")
(TEXT :LEFT 250 :TOP 60 :RIGHT 271 :BOTTOM 60 :STRING "FOUR")
(TEXT :LEFT 277 :TOP 60 :RIGHT 296 :BOTTOM 60 :STRING "FIVE") ...
```

FIGURE 7. Partial description generated from the starting state

(1) *Start Notepad.* The controller can follow different courses of action to start an application in Windows. A labeled icon for the application may be present on the desktop. Many applications can be started by selecting them from the Start menu, which is usually activated by pressing a labeled button in the lower left corner of the desktop. Each

running application is also associated with a labeled button in a task bar at the bottom of the screen. Raising an already running application involves the following sequence:

(a) *Search for the string "Notepad" in the task bar.* (Operations: set-bounds; find-string.) The boundaries of the task bar are hard-coded in the controller. These are used to limit visual processing to the appropriate region. A text object is returned if the string is found in this region.

(b) *Activate the application.* (Operations: set-bounds; get-widgets; mouse-click; wait-for-object.) Processing bounds are reset to the entire display, and its state is recorded. This is necessary for the following step, in which the application button is clicked and the controller waits for the appearance of the Notepad window. VisMap detects changes by polling the visual display for differences from an earlier recorded state. When the Notepad window appears, it will be detected and returned. In the controller, these individual steps are performed as part of a set procedure, simulating an event-driven process, as might be appropriate for a specific cognitive model.

The other procedures for starting an application are comparable. Some of the assumptions and practical limitations of VisMap can be seen even in this short sequence. External knowledge guides the controller's interpretation of the function of interface icons; for example, it can safely treat strings and buttons as being equivalent in the task bar, an equivalence that does not hold elsewhere in the display. One of the alternative methods of starting the application, selecting from the Start menu, relies on comparable knowledge about Windows conventions. Further, the controller must hard-code the location of the Start button, because VisMap cannot recognize its boldface label.

A partial description of an intermediate state of the interface, a representation of the entire display, is shown in Figure 7. Although the recognition rules produce only an imperfect rendering of the displayed text, VisMap can find each of the words reliably. Also notice that none of the graphical icons on the left of Figure 6 are returned. The low-level visual processing stage generates the appropriate internal pixel group regions, with their associated features; these can be retrieved by a call to get-pixel-groups. However, because there exist no rules to recognize the combinations of features, no objects are created to represent the icons.

(2) *Open the document* `numbers.text`. In most applications, opening a document can be managed in different ways: a keystroke accelerator may open a file selection dialog box, or a recently used file may be present in a pulldown menu. The controller follows the most basic approach:

(a) *Find the File menu at the top of the Notepad window.* (Operations: set-bounds, find-string.) Identifying a menu with a given name involves more than simply finding the appropriate string. In Windows, if more than one application is open, more than one instance of the string "File" may be visible. The string "File" may even be present in the text of a document. The controller in this example takes the top left corner of the Notepad window to be the origin of a coordinate system and searches through the text objects inside the boundaries of the window for occurrences of the string "File". It returns the nearest instance, measured by vertical distance, which limits it to the menu bar. An alternative strategy might incorporate more information about the structure of application windows in the user interface.

(b) *Pull down the menu.* (Operations: move-to-point; get-widgets; mouse-click; wait-for-object.) Like the appearance of an application window, a menu selection operation requires that the controller wait until the menu actually appears. As before, this occurs by via polling process.

   (c) *Find the "Open..." option on the menu.* (Operations: set-bounds; find-string; move-to-point; mouse-click; wait-for-object.) A comparable sequence here waits for the appearance of the dialog box.
   (d) *Find and select the document* `numbers.text`. (Operations: set-bounds; find-string; move-to-point; mouse-click; wait-for-object; type-string; press-key.) In this operation the controller assumes that the document is in the default directory, so that it need not browse through the file system hierarchy. The name of the document may be visible in a list box in the Open file dialog. The controller follows an alternative procedure by simply typing in the string "numbers.text" into the active text box, and pressing the return key.

(3) *Select the word "five" in the document.* The only point of note here is that words and letters, like other visible objects, have spatial extent.
   (a) *Find the word "five".* (Operations: set-bounds, find-string.) Notepad, by default, displays new documents in a system font, readable by VisMap. In an extended version of this example, not described here, the controller changes the Notepad font to MS Sans Serif, enabling it to process formerly unreadable text. Here the text is already in the appropriate visual form. VisMap has one additional requirement: the words are separated by three spaces, rather than a single space; otherwise word boundaries are not detected reliably.
   (b) *Select the word.* (Operations: move-to-point, mouse-down, move-to-point, mouse-up.) A common short cut involves a double-click of the mouse inside the boundaries of the word.

With an appropriate controller, VisMap can carry out more extensive procedures than the scenario given, but they all break down the end to similar operations performed on similar interface objects. All along the way, the substrate generates and stores descriptions of the state of the interface. This process can also be carried out by hand, using human guidance rather than an automated controller for transitions between interface states. This scenario demonstrates one of the main roles of VisMap, as an automated or semi-automated generator of realistic interface scenarios, for model input and validation.

5.2. EXECUTING MODELING ACTIONS WITH VISMAP

In this second example we use a portion of a task devised by Byrne to illustrate the operation of ACT-R/PM [Byrne, 1997]. Byrne notes that the design is *not* an attempt at a veridical cognitive model, but is instead an example of a simple, visual task. A sequence of words is presented on the screen. The task is to find the nouns among these words, and for each one found, to click on it to select it, and then to delete it. Our discussion here will be brief due to the similarities to the previous example. The primitive VisMap operations that accomplish this task are identical. Their combination and dynamic application, however, are now governed by a running cognitive model.

Productions for the task include *attend-word*, *move-to-noun*, *click-noun*, *delete-noun*, and *skip-word*. Nouns and other words are also added to memory, including "dog," "girl," "the," "hit," and so forth, along with each word's grammatical category. In the execution of the unmodified ACT-R/PM model, the environment is a dialog window designed to display some number of regions labeled with words. Attention is directed by the vision module to each word in turn. For nouns, the *move-to-noun* production activates the motor module to move the mouse pointer to the appropriate location. Other words are skipped. Once the movement has completed, the mouse is clicked to select the noun, and the delete key is pressed to delete it. The process repeats with a new display of words.

In our restricted version of this task, with the integration of ACT-R/PM and VisMap, only a single iteration is performed. Rather than a dialog designed specifically for the task, the integrated system relies on a word processor, here again Notepad, open to a document containing appropriate text. The *click-noun* production is modified to perform a double-click selection action, but otherwise the ACT-R/PM model remains the same.

ACT-R/PM ordinarily carries out actions in the user interface by calling functions that retrieve appropriate information and effect changes through a Lisp-based user interface management system. In the original task, ACT-R/PM works through an experiment module that consolidates perceptual and motor actions. In the integrated system, these experiment functions are redirected to call VisMap operations. A *process-screen* function generates a hierarchical description of the features of the words visible; output functions for pressing keys, moving the mouse pointer, and pressing the mouse button directly call their VisMap equivalents. An abbreviated trace of the task execution is shown in Figure 8.

This scenario demonstrates the second main role of VisMap, as an online, integrated vision and action substrate for a working cognitive model.

6. Discussion

This section discusses the limitations of VisMap as a development tool for cognitive modeling, and plans for future improvement of the system. For the current implementation, the feature computations and rules were all constructed by hand in a programming language development environment. This can be laborious, especially given the constraints imposted by the simple control structure for rule selection. We are currently working on three tools to address development issues, for feature and rule definition, scenario extraction, and regression testing. We intend for the tools to form the core of a toolkit for exploratory research and practical development in the area of computational models of visual cognition.

Feature and rule definition: We are developing a simple graphical interface, comparable to pixel-editing facilities in a drawing package, to allow users to define features by selecting them and drawing them, rather than describing them programmatically. We are also developing a rule definition and editing tool that allows users to configure features spatially and combine them in text form, to produce high-level rules.

Scenario extraction: VisMap produces display descriptions in a few limited forms. As we saw in Section2, however, cognitive models describe environments in slightly differing formats. Currently, specifying a translation can be handled programmatically, but a more structured, language-based solution would be more general and flexible. Such a translation language is currently under development.

Regression testing: Regression testing allows developers to establish a baseline of performance for a system, and then, with each change in the system, to ensure through repeated testing that its behavior does not significantly degrade. Some elements of regression testing are already supported by the system. The motor module works as a sensor as well as an effector: it can monitor the behavior of users interacting with the user interface. Watching the event queue, VisMap can selectively log all events of interest and insert these into an

```
; Loading ACTR-DEMO:actr-demo;example1.lisp
; (/Research/systems/vismap/domains/actr-demo/example1.lisp)
; Loading ACTR-DEMO:actr-demo;example1.actr
; (/Research/systems/vismap/domains/actr-demo/example1.actr)

Running ACT-R at time 0.000.  Cycle 0 Time 0.000:  Attend-Word
Running ACT-R at time 0.185.  Cycle 1 Time 0.185:  Skip-Word
PM: Skipping word:  "the"
Running ACT-R at time 1.235.  Cycle 2 Time 1.235:  Attend-Word
Running ACT-R at time 1.420.  Cycle 3 Time 1.420:  Move-To-Noun
PM: Going to delete word:  "boy"
VisMap:  Move cursor to (126 104).
Running ACT-R at time 5.106.  Cycle 4 Time 5.106:  Click-Noun
VisMap:  Key press:  Mouse-Double-Click.
Running ACT-R at time 5.806.  Cycle 5 Time 5.806:  Delete-Noun
Running ACT-R at time 5.856.  Cycle 6 Time 5.856:  Attend-Word
VisMap:  Key press:  Delete.
Running ACT-R at time 6.256.  Cycle 7 Time 6.256:  Skip-Word
PM: Skipping word:  "the"
Running ACT-R at time 7.306.  Cycle 8 Time 7.306:  Attend-Word
Running ACT-R at time 7.491.  Cycle 9 Time 7.491:  Skip-Word
PM: Skipping word:  "hit"
Running ACT-R at time 8.541.  Cycle 10 Time 8.541:  Attend-Word
Running ACT-R at time 8.726.  Cycle 11 Time 8.726:  Skip-Word
PM: Skipping word:  "hit"
Running ACT-R at time 9.776.  Cycle 12 Time 9.776:  Attend-Word
Running ACT-R at time 9.961.  Cycle 13 Time 9.961:  Skip-Word
PM: Skipping word:  "the"
Running ACT-R at time 11.011.  Cycle 14 Time 11.011:  Attend-Word
Running ACT-R at time 11.196.  Cycle 15 Time 11.196:  Move-To-Noun
PM: Going to delete word:  "girl"
VisMap:  Move cursor to (166 104).
Running ACT-R at time 14.882.  Cycle 16 Time 14.882:  Click-Noun
VisMap:  Key press:  Mouse-Double-Click.
Running ACT-R at time 15.582.  Cycle 17 Time 15.582:  Delete-Noun
Running ACT-R at time 15.632.  Cycle 18 Time 15.632:  Attend-Word
VisMap:  Key press:  Delete.
...
VisMap:  Key press:  Return.
...
```

FIGURE 8. A trace of ACT-R/PM and VisMap on the noun selection task

internally maintained queue for further processing. This data can be fed back into the user interface to act as a simple script of previously recorded session.

Much of the effort in building these tools is in making the functionality accessible to modelers not necessarily familiar with the underlying detailed design of the system. This entails a significant amount of new research. To interpret the graphical definition of new features correctly, for example, the system will need to perform some amount of visual pattern generalization. We expect to handle these issues as they arise, to produce practical development tools within a short time.

The remaining issues for future work involve integration with more sophisticated cognitive models that operate at a higher level of abstraction, and improvement of the capabilities of visual and motor processing. We believe that VisMap can be integrated with modern cognitive models with minimal programming effort. Our efforts up to the present have concentrated on functionality issues rather than integration issues. Nevertheless, as discussed in Section 5.2, we have loosely coupled VisMap with ACT-R/PM, and we have also used VisMap with controllers based on AI planning systems and arbitrary code in Lisp, C++, and Java. We expect that integration with other specialized modeling languages will pose no difficulty. The main integration question that remains open is the information and processing requirements that VisMap imposes on existing models. The simple existence of an accessible interface to a given environment is no guarantee that a cognitive model will be able to interact with it: representing and processing the relevant knowledge remains to be done. Integration with a variety of different models will provide some measure of validation for VisMap beyond the preliminary arguments we have advanced here.

Acknowledgements

The authors would like to thank L. Zettlemoyer, E. Rasmussen, and D. Foley for their technical contributions to this project.

Appendix VisMap software

The examples discussed in Section 5 were developed using Franz Allegro Common Lisp 5.0 for Windows and GNU Emacs version 19.34.6. The runtime environment is the standard user interface to Microsoft Windows 98, with a screen display size of 1024 by 768.

VisMap comprises some 3,000 lines of C++ code in its sensor/effector modules, plus about 4,000 lines of Common Lisp code for the programmatic interface to external controllers. Although the system gathers its information through WIN32 system calls, it makes very little use of the operating system other than to gather low level window events and screen buffer information. Only about 3% of the system is specific to the Windows operating system. Another 40% is specific to Windows interface conventions for visual display: the appearance of buttons, list boxes, and other controls. One of the strengths of the software is that it separates operating system issues from user interface issues. The system is thus largely platform-independent and operating system-independent (e.g., the system should easily port to a Windows emulator running on a Macintosh or a Unix machine.)

The VisMap system, including the examples discussed in this article, can be obtained via anonymous ftp from `ftp://simon.csc.ncsu.edu/Public/`.

References

Allport, Alan 1989. Visual attention. In Posner, Michael, editor 1989, *Foundations of Cognitive Science*. MIT Press. 631–682.

Alterman, R.; Zito-Wolf, R.; and Carpenter, T. 1998. Pragmatic action. *Cognitive Science* 22(1):55–105.

Anderson, John and Lebiere, Christian 1998. *The Atomic Components of Thought*. Lawrence Erlbaum.

Apple Computer, 1992. *Macintosh Human Interface Guidelines*. Apple Computer, Inc.

Barnard, Philip J. and May, Jon 1999. Representing cognitive activity in complex tasks. *Human-Computer Interaction* 14:93–158.

Byrne, Michael D.; Anderson, John R.; Douglass, Scott; and Matessa, Michael 1999. Eye tracking the visual search of click-down menus. In *CHI '99 (ACM Conference on Human Factors in Computing)*. 402–409.

Byrne, Michael D. 1997. *ACT-R Perceptual-Motor*. [WWW document]. URL: http://www.ruf.rice.edu/˜byrne/RPM/.

Card, Stuart K.; Moran, Thomas P.; and Newell, Allen 1983. *The psychology of human-computer interaction*. Lawrence Erlbaum Associates, Hillsdale, NJ.

Chapman, David 1991. *Vision, instruction, and action*. MIT Press, Cambridge, MA.

Dix, Alan John 1993. *Formal Methods for Interactive Systems*. Academic Press, San Diego.

Fikes, R. E. and Nilsson, N. J. 1971. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3/4):189–208.

Fitts, P. M. 1954. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental Psychology* 47:381–391.

Gentner, Don and Nielsen, Jakob 1996. The anti-mac interface. *Communications of the ACM* 39(8):70–82.

Gonzales, R. C. and Woods, R. W. 1992. *Digital Image Processing*. Addison-Wesley, Reading, MA.

Gray, W. D.; John, B. E.; and Atwood, M. E. 1993. Project ernestine: Validating a GOMS analysis for predicting and explaining real-world task performance. *Human-Computer Interaction* 8(3):237–309.

Hendler, James; Tate, Austin; and Drummond, Mark 1990. AI planning: Systems and techniques. *AI Magazine* 61–77.

Hildreth, Ellen and Ullman, Shimon 1989. The computational study of vision. In Posner, Michael, editor 1989, *Foundations of Cognitive Science*. MIT Press. 581–630.

Hornof, Anthony J. and Kieras, David E. 1999. Cognitive modeling demonstrates how people use anticipated location knowledge of menu items. In *CHI '99 (ACM Conference on Human Factors in Computing)*. 410–417.

Kieras, David and Meyer, David E. 1997. An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Human-Computer Interaction*.

Kieras, David 1999. *A Guide to GOMS Model Usability Evaluation using GOMSL and GLEAN3*. University of Michigan.

Kitajima, Muneo and Polson, Peter G. 1995. A comprehension-based model of correct performance and errors in skilled, display-based, human-computer interaction. *International*

Journal of Human-Computer Studies 43(1):65–99.

Kitajima, Muneo and Polson, Peter G. 1997. A comprehension-based model of exploration. *Human-Computer Interaction* 12(4):345–389.

MacKenzie, I. Scott 1995. Movement time prediction in human-computer interfaces. In Baecker, Ronald M.; Grudin, Jonathan; Buxton, William A. S.; and Greenberg, Saul, editors 1995, *Readings in Human-Computer Interaction: Toward the Year 2000*. Morgan Kaufmann. 483–493.

Marr, David 1982. *Vision : a computational investigation into the human representation and processing of visual information*. W. H. Freeman, San Francisco, CA.

Meyer, David E.; Abrams, Richard A.; Kornblum, Sylvan; Wright, Charles E.; and Smith, J. E. Keith 1988. Optimality in human motor performance: Ideal control of rapid aimed movements. *Psychological Review* 95(3):340–370.

Myers, Brad A.; Giuse, Dario; Dannenberg, Roger B.; Zanden, Brad Vander; Kosbie, David; Pervin, Ed; Mickish, Andrew; ; and Marchal, Philippe 1990. Garnet: Comprehensive support for graphical, highly-interactive user interfaces. *IEEE Computer* 23(11).

Newell, Allen and Simon, Herbert 1972. *Human Problem Solving*. Prentice Hall.

Newell, Allen 1990. *Unified Theories of Cognition*. Harvard University Press, Cambridge, MA.

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

Polson, Peter G. ation.

Ritter, Frank E.; Baxter, Gordon D.; Jones, Gary; and Young, Richard M. 2000. Supporting cognitive models as users. Under review.

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

Sankoff, David and Kruskal, Joseph B., editors 1983. *Time Warps, String Edits, and Macro-Molecules: The Theory and Practice of Sequence Comparison*. Addison Wesley.

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

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

Ullman, Shimon 1996. *High-level vision*. MIT Press, Cambridge, MA.

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

Woods, David D. 1991. The cognitive engineering of problem representations. In Weir, George R. S. and Alty, James L., editors 1991, *Human Computer Interaction and Complex Systems*. Academic Press. 169–188.

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

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

Zettlemoyer, Luke; St. Amant, Robert; and Dulberg, Martin S. 1999. Ibots: Agent control through the user interface. In *Proceedings of the Fifth International Conference on Intelligent User Interfaces*. 31–37.