International Journal of Artificial Intelligence in Education (1999), 10, 314-334

Embedded Training for Complex Information Systems

Brant A. Cheikes, Marty Geier, Rob Hyland, Frank Linton, Annette S. Riffe, Linda L. Rodi, Hans-Peter Schaefer, *The MITRE Corporation, Instructional Technology Center, 202 Burlington Road, Mail Stop K302, Bedford, Massachusetts 01730-1420 USA, e-mail: bcheikes@mitre.org*

Abstract. One approach to providing affordable operator training in the workplace is to augment applications with intelligent *embedded training systems* (ETS). Intelligent embedded training is highly interactive: trainees practice realistic problem-solving tasks on the prime application with guidance and feedback from the training system. This article makes three contributions to the theory and technology of ETS design. First, we describe a framework based on Norman's "stages of user activity" model for defining the instructional objectives of an ETS. Second, we demonstrate a non-invasive approach to instrumenting software applications, thereby enabling them to collaborate with an ETS. Third, we describe a method for interpreting observed user behavior during problem solving, and using that information to provide task-oriented hints on demand.

INTRODUCTION

Complex information systems play a critical and growing role in today's workplace environment. This is particularly true in government offices, where taxpayers expect their civil servants to "do more with less." To maintain or expand services in the face of declining budgets, government agencies are undertaking enterprise-wide modernization programs that propose to increase efficiency and reduce cost by making widespread use of information technology. As these modernization programs proceed, workers are required to adopt new tools and adapt to new ways of doing business. Due to the complexity of the tools, the quality and effectiveness of the associated training mechanisms are increasingly being recognized as key factors determining the overall success of modernization efforts.

Although training is understood to be critical, training organizations still face budgetary pressures and are expected to do more with less. Managers are demanding that employees be trained at or near the workplace, in order to eliminate the costs associated with sending workers to remote training locations. In response, training approaches too are shifting towards increased use of information technology, and distance learning infrastructures are proliferating.

In this article we focus on information technology intended to help people learn to operate complex information systems that have so-called WIMP (windows, icons, menus, and pointing device) graphical user interfaces (GUIs). Our approach involves augmenting information systems with *embedded training systems* (ETS), task-oriented training tools that operate in the same computing environment as the prime application (Figure 1). The goal of an ETS is to teach users how to carry out their job assignments using the prime application. Once an initial training period is concluded, ETSs are expected to remain available for skill-maintenance training on demand.

In practice, ETSs typically are implemented using conventional Computer-Based Training (CBT) or multimedia authoring tools. Unfortunately, while well-designed CBTs and multimedia presentations can effectively present concepts and illustrate procedures, they offer little or no opportunity for authentic practice. Yet practice is essential for acquiring and maintaining information technology skills.

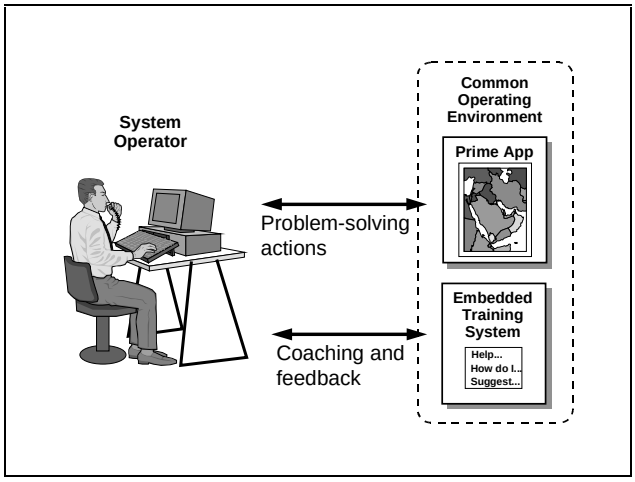

Figure 1. Embedded training system

This article explores embedded training as a domain for the application of *Intelligent Computer-Assisted Instruction* systems (cf. Regian, Seidel, Schuler & Radtke (1996) for a thorough discussion of ICAI technology and its effectiveness). Specifically, we describe ICAI methods of providing coaching during practical exercises on a prime application. The results reported here come from the development of a prototype embedded training capability for operators of a military intelligence application.

Concept of operations

In general, a constellation of knowledge and skills is required in order to use a complex information system effectively to carry out job tasks, *inter alia:*

- Understanding of the general principles of the prime application's computing environment;
- Understanding of the general principles of operating WIMP GUIs in the computing environment, and specific knowledge of the "look and feel" of the prime application's WIMP GUI;
- Understanding of the domain concepts, procedures, and events that pertain to the job tasks to be performed (independent of any particular information system to be used);
- Understanding of the information-system concepts, procedures, and events implemented by the prime application, and their mappings to domain concepts, procedures, and events.

Consider, for example, the knowledge and skills necessary to use a software application such as Quicken[1] to balance one's checkbook. First, one needs to have a basic familiarity with the computing environment in which Quicken will be run, e.g., Microsoft Windows or MacOS. This knowledge enables users to, e.g., locate and run software applications in that environment. Next, one needs to be able to recognize and manipulate the common elements of the application's GUI, for example, the bars, buttons and borders (generically referred to as *widgets*) that one uses to raise, lower, move, and resize windows. Then one needs to be able to recognize and manipulate the *application-specific* GUI widgets, for example, Quicken's unique resource toolbar. To balance a checkbook, one needs to know what checkbooks and bank statements are, understand the basic principles of checking accounts, and know how and why in general one might reconcile one's own records against a monthly report provided by one's bank. Finally, one needs to understand the relevant concepts implemented by Quicken, e.g., that account information is stored in a file, that one may select "tabs" in a graphical "notebook"

[1] A product of Intuit, Inc. See http://www.quicken.com.

to display data on the associated account, that one initiates a "reconcile" operation in Quicken by clicking a graphical button having a particular graphical label on Quicken's toolbar, and so forth. Note that mappings between domain and information-system concepts must also be understood, e.g., the mapping between the user's concept of "balancing a checkbook" and Quicken's implementation of that concept as "reconciling an account."

We have chosen to focus on teaching information-system concepts, procedures, and events in the context of domain concepts, procedures and events. Thus we assume that the students in our target audience will be familiar with the prime application's computing environment and WIMP interface, and will have been trained to perform their job tasks independent of any particular software application. Although these assumptions might not be valid for general user populations, we believe them to be reasonable for our anticipated population of military intelligence analysts. Military analysts undergo extensive training at government facilities and are schooled at length in the business of intelligence analysis. As personal-computer technology has spread through the population at large, recruits increasingly enter military service already familiar with computers and WIMP GUIs. Once these analysts complete their specialization training and deploy to their units, sooner or later they are required to use software applications for which they never received training at the schoolhouse (e.g., because the applications were under development while they were in training). At present, the training support available to deployed users ranges from poor to none.

The objective of our embedded training system is to bring these users rapidly to basic proficiency in operating an unfamiliar software application given that they already know how to do their job. We are not trying to create expert operators from novices; rather, we want operators to learn at least one approved method of using the tool to carry out their assignments. Once training is over, they are free to experiment and improve upon what they have learned.

To achieve this instructional objective, an embedded training curriculum needs to incorporate presentation-oriented teaching materials and to provide opportunities for trainees to apply and refine their knowledge by performing realistic problem-solving exercises. We have chosen to focus our research on the technologies needed to guide and support trainees during practical exercises. We envisage that a fully functional ETS would employ conventional CBT modules (developed following a validated instructional design process) to present concepts and procedures, and an ICAI system to manage practical exercises.

An ICAI system could allow users to perform practical exercises either on a simulation of the prime application, or on the application itself. Simulations have the general advantage that they can highlight the abstract concepts to be learned, excluding details of system operation that might complicate or confuse the instructional objectives of a given exercise. They also can prevent students from exploring garden paths and floundering. In the case of complex information systems, however, these advantages come at a high cost: simulations of each application screen must be developed and made sufficiently "live" and realistic to meet the needs of the curriculum. Each time the system itself is updated, the affected simulations must be updated as well.

Using the application itself for practice eliminates the development and maintenance costs of a simulation, and also supports direct transfer of operator skills to the real job. But there are disadvantages: for example, when using the application itself for training, a mechanism might be needed to switch the application into a non-operational training mode (for example, to disable real-world effects like missile firings during training exercises). Furthermore, it may be difficult or impossible to prevent students from wandering from the desired learning path. These considerations suggest that neither approach is "right" in general; rather, the decision must be guided by the particulars of the prime application. In our case, the prime application is a complex information-analysis tool. Users are unable to affect the external world through their actions on the application, so a "training mode" is unnecessary. Because of the application's complexity, the estimated cost of developing a simulation was sufficiently high to sway the decision in favor of hosting training on the application itself.

Theoretic framework

The design of our embedded training system has been influenced by Norman's (1988) "stages of user activity" model. This model characterizes the process by which people operate human-computer interfaces as a cycle of planning, acting, evaluating results, and replanning. This model and its extension by Abowd and Beale (1991; cf. also Dix, Finlay, Abowd & Beale, 1993) became our framework for defining the instructional objectives of an ETS.

Norman describes two conceptual "gulfs" that explain why users have difficulty operating GUIs. The *gulf of execution* arises from the mismatch between the user's conception of the steps required to achieve a goal, and the actions permitted by the software system. The *gulf of evaluation* arises from the mismatch between the system's external presentation of its internal state, and the user's conception of the expected results of his actions. Both gulfs result from difficulties that users encounter in translating between the *task language* - defined as the concepts and operations that users use to describe their tasks - and the *core language* - defined as the concepts and operations understood by the system.

The premise of our work is that an ETS should facilitate users' acquisition of knowledge and skills that enable them to bridge these gulfs. Each gulf, however, correlates with a distinct set of problems which manifest themselves in different ways. We examine these problems and their symptoms in depth, starting with the outline of a theory that forms the foundation of our analysis.

Fundamental theory

The core language of a software application can be modeled as a set of functions taking zero or more arguments. For example, a text-editing application might provide a function that opens a named file and displays it in an editing window. We might refer to this as the **`OpenFile`** function, and represent it using this notation:

```
OpenFile(path, file)
```

In this expression, **`path`** and **`file`** are the arguments to the function. Associated with each core-language function is a set of *prerequisites* and *effects*. Prerequisites are conditions that must be satisfied in order for the function to be executable. Effects are state changes that result from successful execution of the function.

Prerequisites and effects refer to states of the application (including its computing environment). For example, one of the prerequisites of **`OpenFile`** is that the specified path and file together denote an actual file in the computer file system. One of the effects of **`OpenFile`** is that the application changes its state such that the content of the specified file is displayed in an editing window.

Many of the widgets of an application's GUI enable users to *invoke*, *specify*, *submit* or *cancel* core-language functions. Invoking a function involves indicating an intention to specify its arguments. For example, selecting the **Open** option from a word processor's **File** menu invokes the **`OpenFile`** function. The application responds by displaying a dialog window. This window allows the user to specify the desired values of the function's **`path`** and **`file`** arguments. When finished specifying the arguments, the user may press the **Ok** button to submit the fully-specified function to the application for execution, or may press the **Cancel** button to request that the application return to the state it was in before the function was invoked.

We view a GUI as providing tools that enable users to construct functional expressions in the core language of the application, and to submit them for execution. This view helps us more precisely characterize the gulfs of execution and evaluation, which in turn helps us derive instructional objectives for an ETS.

Understanding the gulf of execution

As the "stages of user activity" model suggests, each cycle of operation begins with the user formulating an intention in the task language. For example, at the beginning of a word-processing session, the user might adopt the intention to "open the IJAIED article". To cross the gulf of execution, the user must construct a plan - a sequence of executions of core-language functions - that will fulfill this intention. This involves many mental steps, including:

- identifying a goal state - a desired configuration of application states that, when reached, cause the task-language intention to be satisfied;
- identifying a core-language plan - a sequence of function(s) that, when successfully executed, will put the application into the goal state;
- ensuring that the prerequisites of each function in the plan will be satisfied by the time the function is submitted for execution;
- identifying the arguments to each function;
- identifying the appropriate values to be assigned to those arguments;
- locating the interface elements that must be manipulated to invoke, specify, and submit the functions;
- recalling the specific manipulations that invoke, specify, and submit each function.

A user's inability to complete any of these steps results in an impasse. Without assistance, users typically begin to flounder. For example, if a user does not know how to decompose the task-language intention into a core-language plan, he might start searching the interface, opening up menu after menu in search of a function to invoke. If the user knows how to invoke the right function but doesn't recall how to specify its arguments, he might try different values or simply cancel the invocation.

This analysis suggests that ETSs should organize their lessons around task-language goals. During these lessons, users should be taught how to decompose their goals into core-language plans, how to verify prerequisites (and/or to execute subplans that serve to satisfy prerequisites), and how to manipulate the user interface to invoke functions, specify their arguments, and submit them for execution.

Understanding the gulf of evaluation

Once a user properly invokes, specifies, and submits a function for execution, the application will attempt to perform the requested operation, and will either succeed or fail. The application typically updates its interface to indicate success or failure. If the operation succeeds, the application will present its results in some manner. If the operation fails, the application will display some indication of the nature of the error (or so one hopes).

To cross the gulf of evaluation, the user must interpret the updated interface and try to answer these questions:

- Has the system finished processing my request?
- Did my request succeed or fail?
- If my request succeeded, where and how are the results presented? How do those results compare to the results I wanted?
- If my request failed, why did it fail?

If users do not know how to recognize that the system has completed processing their request, they may try to take further actions while the system is busy, or may sit idle, waiting for a signal that will never come. If they cannot recognize that their request failed, they may proceed as if it succeeded, possibly to be confused and frustrated when the system behaves

unexpectedly later. If they do not know where results are displayed, or how to interpret those results, they will become blocked. A blockage will also result if users are unable to determine the cause of a failed request.

Thus an ETS must teach users how to bridge both the gulf of execution and the gulf of evaluation. Users must learn how the application responds to their requests, how to interpret its displays, and how to diagnose failures.

Related research

ICAI systems described in the literature typically are standalone software artifacts designed from the ground up to address a chosen instructional task. The system described here differs from other ICAI efforts by using a pre-existing, external application as a tool to support practical exercises.

There is a rich literature on simulation-based approaches to education and training, e.g., Van Joolingen & De Jong's (1996) work on discovery learning, and Munro, Johnson, Pizzini, Surmon, Towne & Wogulis's (1997) RAPIDS/RIDES system for teach of device-operation skills. We are not aware of simulation-based approaches to teaching information-system operation skills.

The most closely related effort is described in Ritter and Koedinger's (1995) article on tutoring agents. They discuss how to add tutoring components to Microsoft's Excel spreadsheet and Geometer's Sketchpad construction tool to create systems that teach algebraic problem-solving and geometric construction, respectively. In both cases, users are being taught problem-solving skills that are independent of the particular tools being used to do the work. Our work complements theirs, focusing less on abstract skills and more on the specifics of how the tool is used to perform tasks.

Another closely related effort is that of Paquette, Pachet, Giroux & Girard (1996) on EpiTalk. Borrowing a term from botany, the authors describe an "epiphyte" approach to augmenting existing information systems with advisor tools. While similar in spirit, our work differs from theirs both in focus and in detail. We focus on the specific issues involved in the teaching of information system operation skills, whereas Paquette *et al.* discuss advisory systems more generally. In addition, we delve into the details of how an ETS might interface with a prime application, and discuss how to algorithmically interpret user actions based on data provided by our interface technology.

Lentini, Nardi and Simonetta (1995) describe an ICAI system that teaches users how to operate a "spreadsheet application" - a pre-programmed spreadsheet meant to support a particular task such as creating a cash-flow report. Unlike our system, Lentini *et al.*'s tutoring component is implemented within the spreadsheet tool; moreover, their research focuses on automatically extracting tutoring knowledge from the contents (formulae) of the spreadsheet application.

More recently, Zachary, Cannon-Bowers, Burns, Bilazarian and Krecker (1998) have described an ETS for tactical team training. In this system, teams of users collaborate to perform real-time tasks in a dynamic simulated environment. There is much in common between our effort and theirs; indeed, the functional architecture of their Advanced ETS includes three components (Automated Data Capture, Automated Assessment & Diagnosis, and Automated Instructional Analysis & Feedback) that map directly to the three-layer architecture we use to describe our system. They address a problem that we have yet to deal with (team training), and skim over the individual operator training that is our chief concern.

Overview

We have sketched a broad vision of the scope and purpose of an ICAI-based ETS. Our development efforts to date represent a step toward realizing that vision. We have focused on designing key infrastructure elements that enable intelligent coaching during practice exercises. Our ETS prototype currently offers one form of coaching during exercises: task-related hints

on demand. When the trainee requests a hint, the ETS identifies what it believes to be the trainee's current goal, and recommends a step the trainee should take next to achieve that goal.

These hints are intended to address one aspect of the gulf of execution, namely, the difficulty that users encounter in decomposing their goals from task-language terms to core-language functional steps. Each hint consists of two parts: a description of the user's current goal, and a description of the next step to be taken towards that goal. Depending on the state of the user's progress in the exercise, the hint may be phrased entirely in task-language terms (Figure 2), or in a combination of task- and core-language terms (Figure 3).

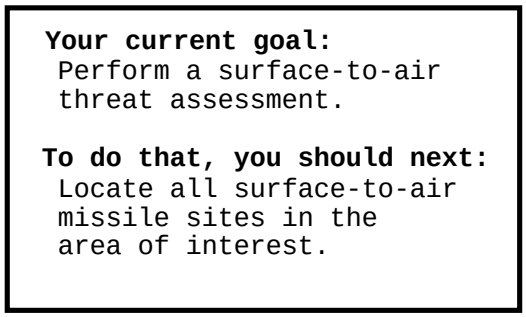

Figure 2. A hint in task-language terms

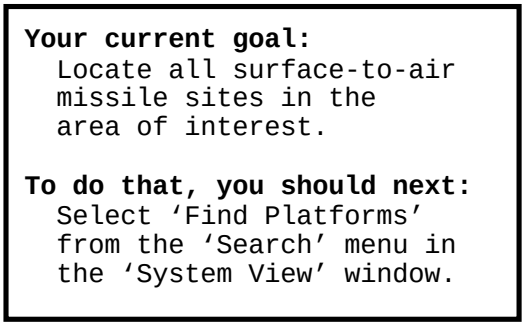

Figure 3. A hint combining task- and core-language terms

We discovered three technical hurdles that an ETS must overcome if it is to produce even relatively simple forms of feedback such as the hints shown above. First, the ETS requires special access to the prime application to monitor users while they solve exercises. Second, the ETS must interpret its observations of user behavior and "understand" how that behavior compares to the desired behavior. Third, the ETS must choose instructional interventions based on its interpretation of the user's behavior and its overall pedagogic objectives.

The mechanisms required to overcome these hurdles can be grouped into three information-processing "layers". The next section summarizes these layers and their interactions, after which we delve into the details each layer's current implementation.

LAYERS OF PROCESSING FOR LEARNING SUPPORT DURING EXERCISES

We group the mechanisms by which ETSs support users during problem solving into three layers: (1) an *application interface* layer, (2) an *action interpretation* layer, and (3) a *training services* layer (Figure 4). Labeled arrows indicate the source and type of communication between layers.

Application interface layer: overview

Ritter and Koedinger (1995) note that for tutoring agents to use off-the-shelf software products (like spreadsheets and word processors) as teaching tools, the software products need to be *observable*, *inspectable*, and *scriptable*. A software product is *observable* if it is able to report to an external application (like a tutoring agent) all actions taken by the user. Software is *inspectable* if it is able to respond to requests for

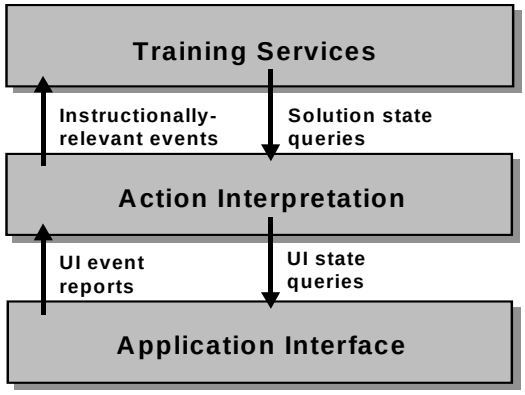

Figure 4. ETS Processing Layers

information about its internal state. *Scriptable* software can execute commands from an external application as if they were initiated by the user. To support an ETS, the prime application must be observable, inspectable, and scriptable.

The elements that provide observing, inspecting and scripting services can be thought of as comprising a distinct software layer, which we call the *application interface* layer. In Ritter and Koedinger's tool-tutor model, this layer is implemented by the prime application itself. That is, the prime application incorporates a programming interface that allows external applications to subscribe to reports of user actions, to issue queries on application state variables, and to execute application commands. When integrated this way into the prime application, the interface layer permits communication between "tutor" and "tool" to occur in the language of the tool. For example, a spreadsheet tool might report user actions in terms of operations like "inserting" and "updating" of "rows", "columns", and "cells". Tutoring agents could refer to individual cells using row and column addresses.

Building an interface layer into an application requires a significant amount of engineering labor. In today's competitive market, this work gets done only if a compelling business need exists. Even when such a need exists (e.g., for office-automation software), the lack of standards in this area leads to a situation in which interfaces differ greatly across both tools and vendors. (See the LTSC Home Page for an IEEE-sponsored project to develop standards for educational technology.)

The approach we take is to implement the application interface layer as an independent component external to the prime application. Our solution exploits software "hooks" in the prime application's computing environment, specifically, in the window-management environment. The advantage of this approach is that it has the potential to apply to any application that runs in the computing environment. The disadvantage is that communication between the prime application and the ETS can no longer occur in application-specific terms, since the requisite knowledge resides only inside the prime application. Instead, the communication takes place in more generic terms of user-interface gestures and states, like inserting text into an edit field, clicking a button, or selecting a menu option. This level of description, however, enables our ETS to provide detailed guidance down to suggestions of specific user-interface gestures.

Given this background, the arrow labeled "UI event reports" in Figure 4 can be interpreted as real-time reports of the trainee's actions on the prime application's UI. The arrow labeled "UI state queries" refers both to requests for information about the state of the UI (e.g., states of toggle buttons, contents of edit fields) and to commands that affect the UI state (e.g., commands to set the values of edit fields, to select menu items, to press buttons, etc.).

Action interpretation layer: overview

The action interpretation layer of an ETS builds and maintains a model of the trainee's problem-solving activity during practice exercises with the prime application. This model is accessed by the training services layer to support the production of guidance and feedback. Each of the trainee's UI gestures is reported to the interpretation layer by the interface layer, whereupon the interpretation layer updates its model. The interpretation layer may also need to inspect the state of the prime application's UI (hence the arrow labeled "UI state queries" in Figure 4). The interpretation layer may alert the training services layer if it notices an instructionally-relevant event, e.g., if the user does something unexpected.

The information needs of the training services layer are the primary drivers of design requirements on the interpretation layer. Cataloging the full set of requirements is an open research task. In the prototype described here, the interpretation layer builds and maintains a model that tracks the user's progress through the assigned problem-solving exercise. This model provides enough information to enable the training services layer to offer task-oriented hints on demand.

```
menu(press,"View;Open...","State Tools",user,5);
scroll(set,"directory_scroll","Select File",user,9,"13");
scroll(set,"file_scroll","Select File",user,11,"10");
list(select,"Dir_List","Select File",user,13,"training");
list(select,"F_List","Select File",user,19,"tango2.view");
button(press,"OK","Select File",user,24);
```

Figure 5. WOSIT observation stream

Training services layer: overview

The training services layer contains the mechanisms that implement the ETS's instructional strategy. Informing the decision-making processes at this level is the system's interpretation of its observations of user actions. We have proposed that an ETS's training services should help users learn to bridge the gulfs of execution and evaluation. Our work on training services has thus far focused on techniques that help users bridge the gulf of execution. We have implemented one service: hints on request (cf. Figure 2 and Figure 3). These hints are intended to help users locate their "place" in the problem-solving process, and to identify the next problem-solving step to take.

Implementing this service has helped us to identify design requirements for the action interpretation layer. Specifically, we have found that the model developed by the action interpretation layer must be hierarchical, and explicitly relate task-language concepts and actions to core-language concepts and actions. That is, the upper levels of the model should be expressed in task-language terms, and the leaves in core-language terms. Each path from root to leaf contains a transition from task-language representation to core-language representation.

Summary

This section described three layers of computational mechanisms that comprise the within-exercise support component of an ETS. We discuss our current implementations of these layers next.

APPLICATION INTERFACE LAYER: IMPLEMENTATION

An ETS must be able to observe, inspect, and script the prime application in order to employ ICAI techniques. We have developed a software tool that provides the needed access without requiring modifications to the prime application. This section describes our implementation of this tool, called WOSIT (Widget Observation, Scripting and Inspection Tool).

Overview

WOSIT is a Unix application that runs independently of both the prime application and the ETS. It works by modifying the window-management environment in which the prime application operates. This technique is window-system specific; the version of WOSIT described here applies to any Unix application built with the X-Windows "Motif" widget set. (We are currently prototyping a version of WOSIT that observes, inspects and scripts Java applications, and a commercial company is developing similar technology for Windows applications, cf. http://www.guruinc.com.) The only requirement for using WOSIT is that the prime application must be *dynamically linked* with the X-Windows libraries. (This is standard practice.)

By modifying the windowing environment, WOSIT gains access to all the X-Windows functions that the prime application itself can perform. WOSIT can intercept widget state-

change notifications, query widget state, and even generate widget actions (e.g., button clicks) as though performed by the user. WOSIT also can modify the prime application's GUI; for example, it can add a menu of tutor-related commands to the prime application's menu bar (to foster a perception that the ETS is fully integrated with the prime application).

WOSIT and the ETS communicate with each other over network sockets. Each time WOSIT notices that a GUI widget has undergone a significant state change, it sends a descriptive message over the socket to the connected ETS. At any time the ETS can ask for information about the interface or send a scripting command. Figure 5 illustrates a sequence of WOSIT reports. Figure 6 shows a series of inspection and scripting commands from the ETS and the corresponding WOSIT responses.

In these reports, the function name indicates the type of widget involved in the event (e.g., button, edit field, list), and the first argument indicates the type of event that occurred. The second argument identifies the specific widget, and the third argument identifies the window in which the widget is located. The fourth argument indicates the initiator of the event (*user* or *system*). The fifth argument is a timestamp, and any additional arguments serve to further describe the event.

As the figures illustrate, WOSIT uses symbolic labels to refer to windows and widgets. Whenever possible, WOSIT uses the text from windows and widgets for their labels. WOSIT ensures that these labels are constant across executions of the prime application, and unique within each window.

Technical approach

Three technologies enable non-invasive observing, inspecting and scripting of X/Motif applications: The InterClient Exchange (ICE) library, X-Windows hooks, and the Remote Access Protocol (RAP). ICE and the X-Windows hooks have been part of the standard X11 package since Release 6.1. RAP was originally developed at Georgia Tech as part of the Mercator research project (Edwards, Mynatt & Rodriguez, 1993; see also the Mercator home page). It was later revised and publicly released by Will Walker at Digital Equipment Corporation. Figure 7 illustrates how these elements are combined.

The ICE library enables programs to communicate across address spaces. ICE acts as a more efficient version of a socket connection. Functions to send and receive data, and to open and close connections are all standardized within the ICE library. The X-Windows hooks provide the means for a software component within the same address space as a client program to subscribe to widget-level changes in the client.

RAP provides the "bridge" enabling widget notifications to be passed across address

Commands

```
list(windows);
list(widgets,"Select File");
query(widget,"Select File","Dir_List",value);
flash(widget,"Select File","OK");
```

WOSIT Responses

```
group(windows,"Select File","ARC","Spatial Analysis");
group(widgets,"OK","Dir_List","F_List");
list(inspect,"Dir_List","Select File",{"training"});
flash(widget,"Select File","OK");  /* OK Button flashed */
```

Figure 6. WOSIT command/response stream

spaces from the prime application to WOSIT. It implements a suite of procedures to send and receive widget information using X-Windows hooks. It comes in two parts: a client-side library and an agent-side library. The client-side library resides in the prime application's address space, interfacing with the X-Windows hooks and sending data via ICE to the agent-side library. The agent-side library resides in WOSIT's address space, acting as a buffer between the WOSIT application and the RAP client library. The WOSIT application processes messages coming from the prime application via RAP, and provides a socket-based reporting and query interface to client applications such as the ETS.

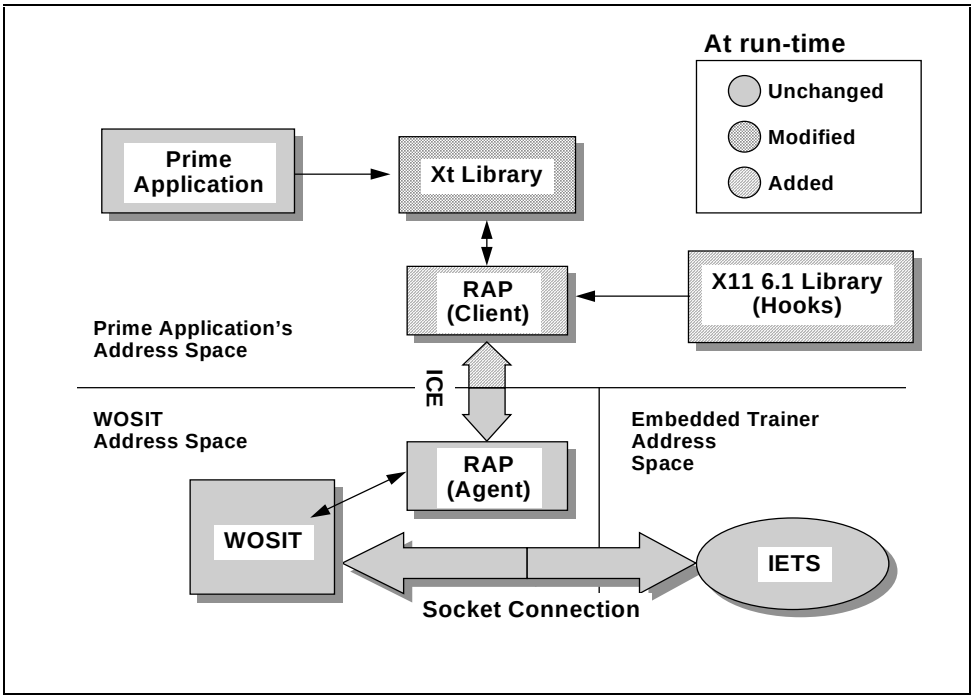

Figure 7. WOSIT technical approach

Limitations

The X11 hook mechanism is widget-set independent. However, to interpret widget changes and report them, WOSIT must make widget-specific assumptions. For example, to recognize that the selected item in a list has changed, WOSIT needs to know which of the widget's state variable(s) should be monitored. Thus WOSIT's interpretation code must be extended for each new widget or widget set to be monitored. WOSIT provides an easy-to-configure text file for output modifications and to allow reporting of new widgets. This makes it possible to extend WOSIT without additional programming. In the future, an authoring tool should be developed to further simplify the configuration task.

Loading the client-side RAP library into the prime application's address space requires it to be linked with the prime application. To avoid forcing the prime application to be recompiled, we have patched the Xt library in the standard X11 release to have a dependency on the RAP client library. Thus the prime application needs only to have its Xt library replaced to function properly with WOSIT. For applications built with versions of X11 prior to Release 6, both the Xt and the standard X libraries must be replaced. Replacing these libraries only requires changing an environment variable as long as both libraries are dynamically linked with the prime application.

Status

Our implementation of WOSIT has been tested under Solaris versions 2.5 through 2.7. Currently, the Unix version of WOSIT fully supports all Motif widget observations, provides a configurable text file for widget modifications, responds to generic inspection requests, and has the ability to flash any GUI widget.

WOSIT may be downloaded at http://www.mitre.org/technology/wosit/ or by contacting the first author.

ACTION INTERPRETATION LAYER: IMPLEMENTATION

The action interpretation layer in our prototype ETS receives a stream of reports from the interface layer (cf. Figure 4) describing the trainee's actions on the prime application's GUI. This section describes how the interpretation layer processes this input to build and maintain a model of the trainee's problem-solving activity.

Overview

Norman's (1988) "stages of user activity" analysis implies that our action interpretation layer would need to perform plan recognition. However, plan recognition mechanisms are expensive both to develop and to run. In our view, embedded training is a structured process, and that structure should allow us to use reasoning and inference mechanisms that are significantly less powerful and less expensive than plan recognizers.

Recall that embedded training consists of presentation-oriented learning activities and interactive problem-solving exercises. Before assigning an interactive exercise to a trainee, an ETS should present the new knowledge to be practiced. (Conventional multimedia approaches are adequate for the presentation portion of embedded training.) Exercises are used primarily to reinforce and make concrete the concepts and methods conveyed by the presentations. Exercises also provide opportunities for trainees to apply and become comfortable with skills that were studied earlier in the curriculum. If the ETS carefully prepares students for each exercise, it is reasonable for it to have strong expectations about the problem-solving process they will follow. These expectations can drive the system's interpretation of trainee actions.

Based on these ideas, we have developed an *expectation-driven* approach to action interpretation. At the start of each exercise, the action interpretation layer of our ETS constructs an *expectation model* that encodes the problem-solving process that the trainee is expected to follow. (It constructs this model from elementary units stored in a knowledge base; these are described in the next section.) The expectation model can be thought of as the system's "expert model" for each exercise, in the sense that it represents the problem-solving approach against which the student's approach will be compared. As the student's problem-solving actions are observed, the interpretation layer performs a classification procedure to determine whether the observed action is consistent with the set of expected actions.

The Expectation Model

The expectation model is a data structure built and maintained for the purpose of tracking students through a problem-solving exercise. It is organized as a tree, similar to a plan tree. The root node of the tree represents the most general goal or objective of the exercise, and is typically encoded using terms and concepts from the task language. An example from our prototype is the goal "Perform a surface-to-air threat assessment in the Tango2 area". The leaves of the tree are GUI event reports passed up from the interface layer. Between the root and the leaves are arbitrarily many intermediate layers of subgoal nodes. Figure 8 illustrates portions of an expectation model for the surface-to-air threat assessment exercise.

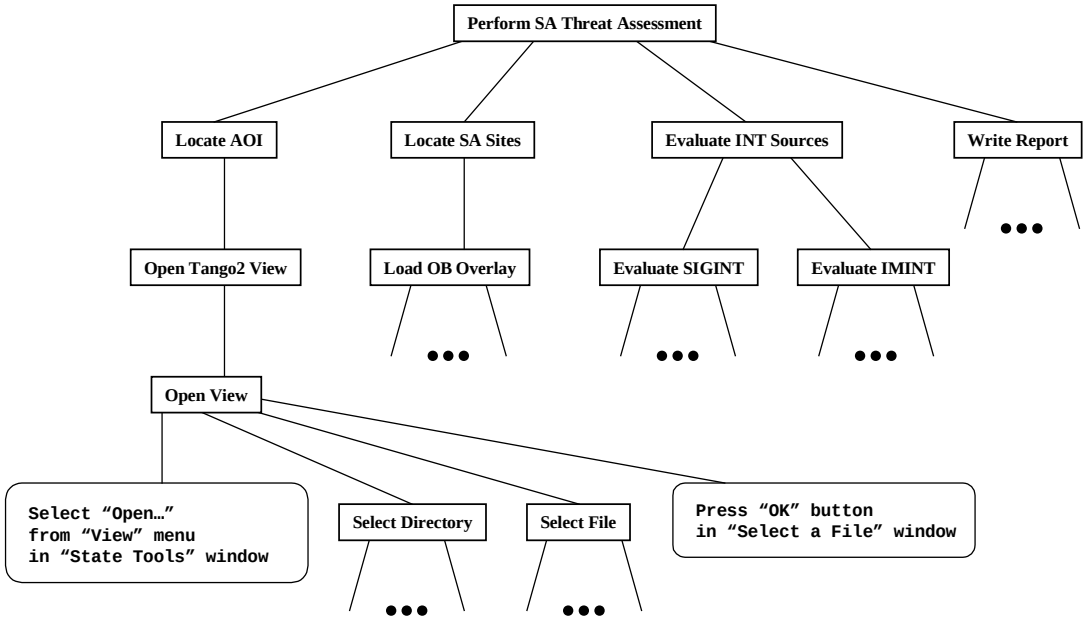

Figure 8. Hierarchical expectation model

In the figure above, the top-level goal is decomposed into four subgoals: Locate the Area of Interest, Locate Surface-to-Air Missile Sites, Evaluate Intelligence Sources, and Write Report. Locating the AOI breaks down further, crossing over from task-language terms into core-language terms like opening a view in a geographic information system. Opening a view requires the operator to first perform the GUI action of pulling down the "View" menu from the prime application's "State Tools" window, then selecting the option "Open". The system then displays the "Select a File" dialog, and the operator chooses a file in the computer file system. Once the file has been chosen, the operator completes the "Open View" step by pressing the "OK" button on the dialog window.

Only the leaves of an expectation model represent observable actions. All other nodes in the model are intended to represent meaningful abstractions over those actions. Teaching these abstractions is one of the ETS's duties, since they are the building blocks that help students bridge the gulf of execution.

Technical requirements

The illustration in Figure 8 makes no commitments regarding the order in which the actions at the leaves are to be performed. We have found that in most realistic exercises, a wide variety of paths through the solution process are permissible. Yet there are constraints, some enforced by the prime application (e.g., most of the tool's functions are disabled until a view is opened), and others by "best practices" or doctrine in the task domain. Thus it is important to encode these constraints explicitly in the expectation model.

In our work thus far, we have found it necessary to distinguish five ordering relations among subgoals:

- InOrder
- AnyOrder
- ZeroOrMore
- OneOrMore
- PickOne

These relations establish conditions on the order in which the immediate subgoals of a goal node should be achieved. The *InOrder* relation means both that all subgoals must be achieved for the goal to be achieved, and that they must be achieved in sequence. For example, to properly achieve the "Open View" goal, the operator must first select the "Open" item from the "View" menu, then select a directory, then select a file, and then press the "OK" button. Knowledge of InOrder relations is essential for suggesting appropriate next steps.

The *AnyOrder* relation means that all subgoals must be achieved, but their order is arbitrary. For example, when evaluating intelligence sources, it does not matter whether the operator looks at signals (SIGINT) or imagery (IMINT) data first. The *AnyOrder* relation also permits subgoals to be pursued in parallel, an important feature of window-based applications.

The *ZeroOrMore* relation means that a goal is achieved by achieving its subgoals zero or more times. This provides a means of describing optional behavior. If more than one subgoal is defined, the order of achievement is arbitrary. For example, after loading a map overlay into the prime application's situation display, the user might pan or zoom the display. The *ZeroOrMore* relation allows us to record these actions in the expectation model (for student modeling purposes) without requiring that they actually be performed.

The *OneOrMore* relation means that a goal is achieved as soon as any one of its subgoals is achieved, but in addition, subgoals may be achieved more than once and they may be achieved in any order. As a simple example of this, the "Write Report" goal is achieved when the operator sends an e-mail message to a supervising officer, but the operator may also choose to disseminate the information more widely by sending additional e-mail messages.

Finally, the *PickOne* relation allows us to define alternatives. A *PickOne* goal is achieved when exactly one of its subgoals is achieved.

By augmenting the expectation model with these five ordering relations, we transform it into a compact representation of a space of valid problem-solving sequences. In the next subsection, we describe how our expectation models are implemented.

Implementing the expectation model

Expectation models in our system are implemented as trees of active processing elements called *recognizers*. There are six types of recognizers, one corresponding to each ordering relation defined above as well as a *BasicMatch* type. Each type has specialized behavior. All six types have a *description*, a set of *arguments*, a *body*, and three Boolean state variables: *repeatable*, *satisfied* and *violated*. Recognizers receive GUI event reports as input, and may change their state and/or pass the reports on to subordinate recognizers in response. Processing begins when a GUI event report is input to the recognizer at the root of the expectation model, and ends when the root recognizer returns a result.

The *description* of a recognizer is a symbolic value denoting the goal which the recognizer represents. The *arguments* modify the goal; for example, the *PerformSAThreat* goal takes one argument representing the area of interest to be analyzed. The *body* of a recognizer is either a pattern to be matched against GUI event reports (for the *BasicMatch* recognizer type) or a list of subordinate recognizers (for all other recognizer types). The *satisfied* and *violated* state variables indicate whether the recognizer's goal is satisfied or violated, respectively. The *repeatable* state variable indicates whether the recognizer may be satisfied repeatedly. In the following subsections, we summarize the unique behavior of each type of recognizer.

BasicMatch recognizers

BasicMatch recognizers comprise the leaf nodes of the expectation model structure. Their *satisfied* and *violated* state variables are both initialized to *false*, and their *repeatable* variable is initialized to *true*. The body of a BasicMatch recognizer is a pattern to be matched against incoming GUI event reports, and looks similar to a WOSIT observation message (cf. Figure 5). For example, the pattern

```
button(press,"OK","Select File",user)
```

will match a report of the user pressing the "OK" button in the "Select File" window. The pattern

```
list(select,"F_List","Select File",user,?file)
```

will match any report in which the user selects an item from the "F_List" (file list) of the "Select File" window, and as a side effect will bind the variable `?file` to the name of whatever file the user selected.

When a BasicMatch recognizer receives a GUI event report, it compares the functor and first four arguments against the pattern in its body. If any of these fail to match, the recognizer simply ignores the report. If they all match, the recognizer then compares any additional arguments against corresponding elements of the pattern. If the pattern element is a variable, then the recognizer binds the variable to the value in the event report. If the pattern element is a value, then the recognizer compares it to the value in the event report. If the values fail to match, the recognizer sets its *violated* state variable to *true*. If all pattern match arguments either match or bind to corresponding elements in the event report, the recognizer sets its *satisfied* state variable to *true*.

When it completes the processing of an event report, the BasicMatch recognizer returns the values of its state variables as well as any variables bound as a side effect of a successful match.

InOrder recognizers

InOrder recognizers are initialized with *satisfied*, *violated*, and *repeatable* all set to *false*. They respond to GUI event reports by passing the reports to the first unsatisfied or repeatable subordinate recognizer in their body. Along with the event report, the recognizer passes a variable binding stack consisting of a set of variables and associated bindings. When the subordinate recognizer completes processing the report, the InOrder recognizer inspects the results. If the subordinate recognizer changed its state to *satisfied*, the InOrder recognizer updates the variable binding stack with any bindings that were passed back, then passes its own state variables and the binding stack back to its caller. When all subordinate recognizers become *satisfied*, the InOrder recognizer sets its own state to *satisfied*. If any subordinate recognizer becomes *violated*, the InOrder recognizer sets its state to *violated*.

AnyOrder recognizers

AnyOrder recognizers are initialized to be unsatisfied. They pass GUI reports on to *all* unsatisfied or repeatable subordinate recognizers. An AnyOrder recognizer becomes satisfied when all subordinate recognizers are satisfied, and violated when any subordinate recognizer is violated.

ZeroOrMore recognizers

ZeroOrMore recognizers are initialized to be both repeatable and satisfied. They pass GUI event reports to each subordinate recognizer until a change of state is detected.

OneOrMore recognizers

OneOrMore recognizers are similar to ZeroOrMore recognizers, except they are initialized as unsatisfied. They become satisfied as soon as any of their subordinate recognizers become satisfied.

PickOne recognizers

PickOne recognizers are initially unsatisfied. They pass event reports to all subordinate recognizers, and become satisfied as soon as any of their subordinates become satisfied. However, they become violated if more than one subordinate becomes satisfied.

Summary

The expectation model described in this section compactly represents a constrained space of possible problem-solving approaches. In particular, we can distinguish between sets of tasks that must be performed in a particular sequence, and those that may be performed in any order and/or in parallel. We can also encode optional action sequences and alternative methods of achieving the same objective. The next section describes how the interpretation layer uses the expectation model when processing event reports from the interface layer.

The Recognition Process

Action interpretation in our ETS prototype is carried out using a collaborative process between a governing *Recognizer Manager* and the recognizers in the expectation model. The process is inspired by the work of Chandrasekaran & Johnson on Generic Task theory (1993; see also Chandrasekaran, 1983). The Recognizer Manager mediates all interactions between recognizers in the expectation model. Together, the Recognizer Manager and the expectation model recognizers work to find an appropriate leaf node to match against incoming event reports. The key tasks of the Recognizer Manager are to recognize interleaved task execution, and to check for order violations. We illustrate the principles of the interpretation process through a pair of examples.

Example 1

Consider a simple expectation model consisting of three nodes (Figure 9). The root node (R) is an InOrder recognizer in state [-s,-v], meaning not satisfied and not violated (we ignore repeatability in these examples). Nodes N1 and N2 are BasicMatch recognizers, and both are also in state [-s,-v]. The Recognizer Manager is the action interpretation layer's interface to the rest of the system. When a GUI event report is received, the Recognizer Manager passes it first to the root node of the expectation model.

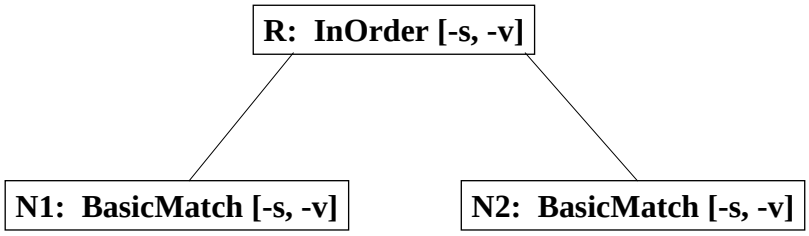

Figure 9. InOrder model

Because R is an InOrder recognizer, it issues a request back to the Recognizer Manager to pass the report on to N1, the first unsatisfied subordinate recognizer.

At this point, three scenarios are possible. The incoming report might match N1, might violate N1, or might not match N1. If the report matches N1, then N1's state will change to [+s,-v] and interpretation ends. Similarly, if the report violates N1, then N1's state will change to [-s,+v], as will R's (the violation propagates up the expectation model). If the report does not match or violate N1, the Recognizer Manager will try passing it to N2. In this case, however, if N2 becomes satisfied as a result, the Recognizer Manager will mark R as *violated*, to indicate that an ordering constraint failed (N2 was recognized before N1).

In general, if the Recognizer Manager fails to find a match for a GUI event report below a branching point, it will look for matches down other paths that may violate the ordering constraints at that branch, to determine whether the constraints have been violated. At present, if the Recognizer Manager can find no match for an event report, it simply ignores it. Of course, such an event might signal that the user is proceeding down an unexpected, incorrect path. More research is needed into how such situations should be handled.

Example 2

We now consider an expectation model whose root is an AnyOrder recognizer (Figure 10).

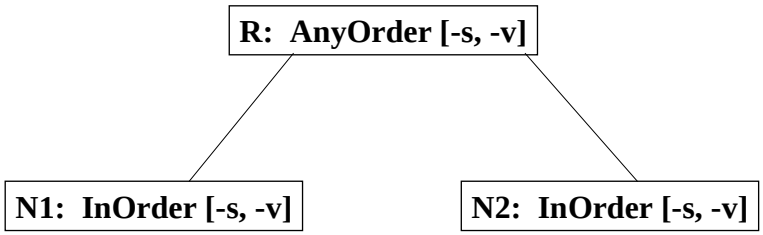

Figure 10. AnyOrder model

In this example, nodes N1 and N2 are InOrder recognizers, and we will assume they have the same substructure as R in Figure 9. The Recognizer Manager will pass incoming event reports first to R, which will in turn request that they be passed onto N1. Processing will continue as in Example 1. If a match or violation is found below N1, interpretation ends, otherwise processing continues to N2 and below. What is interesting about this example is that the model allows users to interleave their work on the goals represented by N1 and N2. That is, the user may take an action that matches a leaf recognizer below N1, then later take an action that matches a leaf recognizer below N2. However, once either N1 or N2 become satisfied, matches will no longer be explored below them. When both N1 and N2 become satisfied, R will become satisfied, and matches will no longer be explored below R. Thus the pattern of satisfaction of the recognizer nodes models the user's progress in the exercise.

Summary

This section has described an expectation-driven approach to action interpretation. The interpretation layer in our prototype ETS builds and maintains an expectation model that represents the set of acceptable paths through the solution process for the assigned exercise. The expectation model is constructed out of active processing elements called recognizers. The recognizers collaborate with the governing Recognizer Manager to find a leaf node that matches received event reports. Two kinds of violations are recognized: ordering violations, and argument violations (i.e., user actions that violate a BasicMatch recognizer, such as when the user selects the wrong item from a list widget). The expectation model allows a variety of problem-solving approaches to be compactly represented, and also indicates goals that may be pursued in parallel.

The expectation model and associated interpretation procedure were designed to permit the ETS to track students through problem-solving exercises. Furthermore, the model was designed to be used by the training services layer in the process of generating guidance and feedback. We turn next to a discussion of how our ETS generates hints using the expectation model.

TRAINING SERVICES LAYER: IMPLEMENTATION

During interactive exercises, the ETS maintains a small interface on the trainee workstation that enables students to access training services. At present, services are only provided upon request when the student clicks the ETS's 'hint' button. Our primary goal in implementing this service was to better understand the kinds of information required of the interpretation layer. More research is needed to develop a suite of training services sufficient to help students bridge the gulfs of execution and evaluation.

Technical approach

Hints are constructed from natural-language patterns attached to the nodes of the expectation model. When the student requests a hint, the hint-generation mechanism examines the expectation model and determines the *current task* **c** and the *next task* **n** as follows:

l = the last leaf recognizer to match all of the features of a UI event report.

c = the closest ancestor of **l** such that:
c is in the set of recognizers that have not yet been satisfied, and
c has attached coaching information.

n = the first (as determined by the control type) child of **c**, such that:
n is in the set of task recognizers that have not yet been satisfied.

The hint generator then fills in a template replacing any variables in the coach pattern with their values (taken from the variable binding stack), and displays the resulting text. At this point, the hint generator remembers the node corresponding to **n**, and activates the 'More Detail' button on the tutor interface. The example below shows how hinting works.

Example

Consider the portion of an expectation model illustrated in Figure 11.

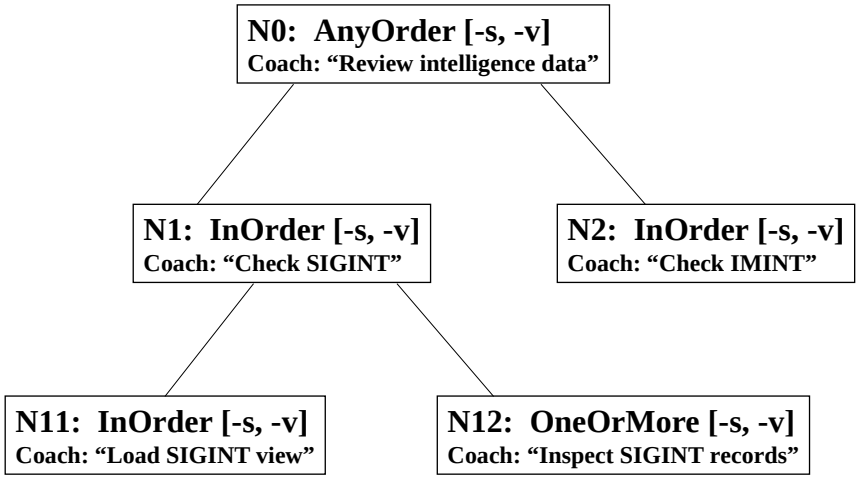

Figure 11. Expectation model with coaching data

Suppose the student performs an action that matches a leaf node somewhere below N11, and that as a result, N11 becomes satisfied. If the student were to request a hint at this point, the system would find N1 to be the closest unsatisfied ancestor of the matched leaf node, and would identify N12 as the next goal. In this case, the system would produce a hint such as:

Your current goal is: Check SIGINT.
To do that, you should next: Inspect SIGINT records.

Node N12 would be remembered as the suggested next step. If the student presses the 'More Detail' button without taking any other actions on the prime application, the hint generator "drills down" below N12 and explains how it is to be accomplished, e.g.,

To achieve the goal: Inspect SIGINT records.
You should next: Press the "All Radar" button in the "State Manager" window.

Summary

Our existing training services layer is quite simple. The implementation is intended primarily to demonstrate the ETS's ability to track users flexibly through realistic problem-solving exercises and to suggest appropriate next steps on demand. Although the expectation model is able to indicate certain types of violations, we have not yet developed feedback mechanisms that exploit this information.

OPEN PROBLEMS AND CONCLUSIONS

The work described in this article has been implemented in a prototype ETS for an information system used by military intelligence analysts. With only one training service supported (hinting), the prototype is not yet mature enough for evaluation of its training adequacy. Our formative evaluations of the prototype, however, have revealed several weaknesses in the interpretation process; we summarize the key issues below.

Open Problems

Our ETS's training services layer depends on robust tracking of students' actions during practice exercises. Our current implementation of the action interpretation layer needs to be improved to handle cancellations and restarts, "floating" goals, and unexpected behavior.

Cancellations and restarts

We found that users commonly proceed down an incorrect path, realize this, then cancel the operation before completing it. For example, a user might bring up a dialog box prematurely, start to fill it in, then realize the mistake and hit the 'Cancel' button. This has the effect of rolling the system's state back to where it was just before the user acted to bring up the dialog. However, our current interpretation layer has no notion of the 'extent' of a command, and is thus unable to properly handle cancelled command sequences. Similarly, users sometimes cancel command sequences only to restart them from the beginning. It seems clear that the action interpretation layer needs to model user activity at a slightly more coarse grain-size than widget-level events.

"Floating" goals

In some exercises, there are goals that users may try to achieve at almost any time. Panning and zooming a situation display are two such examples. Although we may want to be able to recognize when a user is panning or zooming, we cannot say exactly when these actions should occur. Thus we may need to define "floating" goals that have no specific place in an expectation model, but are available to be "spliced in" whenever the user so chooses.

Unexpected behavior

What should the action interpreter do when it observes user actions that are completely unexpected? At present, our system ignores them. Unexpected actions may indicate that the user has gotten lost in the exercise, or perhaps is off on a side trail, exploring some interesting

aspect of the system's behavior outside the scope of the current activity. It remains an open question whether the system should attempt to perform an analysis of the user's activity, or intervene directly to get the user back on an expected path. This problem is an outgrowth of the decision to support training using the application itself, rather than a simulation. A mockup of the system could disable all aspects of simulated behavior that do not directly support the problem-solving task being practiced. Users of the actual application cannot be so readily constrained. Early intervention by the ETS may be the best strategy as soon as unexpected actions are noticed; this approach remains to be investigated.

Conclusions

ETSs have the potential to provide high-quality, cost-effective operator training at or near operators' workplaces. We believe that information-system operation represents an important new domain for the application of ICAI technology. This article makes three contributions to the development of this technology. First, we have described a framework, based on Norman's "stages of user activity" model, for defining the instructional objectives of an ETS. This framework suggests that ETSs should provide guidance and coaching that enables users to bridge the gulfs of execution and evaluation. Second, we have demonstrated a non-invasive approach to instrumenting software applications. Our WOSIT tool makes X-Windows/Motif applications observable, inspectable and scriptable in a manner that does not require application-specific modifications. This allows us to add ETSs to applications that were never designed to collaborate with a training system and consequently lack the requisite programming interfaces. Third, we have described a method for interpreting user behavior during problem solving. Our action-interpretation mechanism matches incoming reports of user-interface gestures against a multilevel pattern that compactly represents the set of valid problem-solving paths for the exercise. After each user-interface gesture, the action interpreter annotates its expectation model and keeps track of where the user is in the problem-solving process. The annotated expectation model is then used by the training services layer to compute task-oriented hints on demand.

Acknowledgments

This research has been supported by the MITRE Technology Program. Comments and suggestions from Lisa Haverty and two anonymous reviewers have contributed to the quality of the manuscript and are appreciated.

References

Abowd, G. D. and Beale, R. (1991). Users, systems and interfaces: A unifying framework for interaction. In Diaper, D. and Hammond, N., editors, *HCI'91: People and Computers VI*, Cambridge: Cambridge University Press, 73-87.

Chandrasekaran, B. (1983). Towards a Taxonomy of Problem Solving Types. *AI Magazine*, 4(1), 9-17.

Chandrasekaran, B. and Johnson, T. R. (1993). Generic Tasks And Task Structures: History, Critique and New Directions. In David, J. M., Krivine, J. P. and Simmons, R., editors, *Second Generation Expert Systems*, Springer-Verlag, 239-280.

Dix, A., Finlay, J., Abowd, G. and Beale, R. (1993). *Human-Computer Interaction*. New York: Prentice Hall.

Edwards, W. K., Mynatt, E. D. and Rodriguez, T. (1993). The Mercator Project: A Nonvisual Interface to the X Window System. In The X Resource, Seastopol, CA. Issue #7.

Lentini, M., Nardi, D. and Simonetta, A. (1995). Automatic Generation of Tutors for Spreadsheet Applications. In *Proceedings of AI-ED 95 - World Conference on Artificial Intelligence in Education*, Washington DC, August, 59–66.

LTSC Home Page: http://ltsc.ieee.org.

Mercator Home Page: http://www.cc.gatech.edu/gvu/multimedia/mercator/mercator.html.

Munro, A., Johnson, M., Pizzini, Q., Surmon, D., Towne, D. and Wogulis, J. (1997). Authoring simulation-centered tutors with RIDES. *International Journal of Artificial Intelligence in Education*, 8, 284-316.

Mynatt, E. D. and Edwards, W. K. (1992). The Mercator Environment: A Nonvisual Interface to the X Window System. Technical Report GIT-GVU-92-05, Georgia Institute of Technology.

Norman, D. A. (1988). *The Psychology of Everyday Things*. Basic Books.

Paquette, G., Pachet, F., Giroux, S. and Girard, J. (1996). EpiTalk: Generating Advisory Agents for Existing Information Systems. *Journal of Artificial Intelligence in Education*, 7(3/4), 349–379.

Ritter, S. and Koedinger, K. R. (1995). Towards Lightweight Tutoring Agents. In *Proceedings of AI-ED 95 - World Conference on Artificial Intelligence in Education*, Washington DC, August, 91–98.

Regian, W., Seidel, R., Schuler, J. and Radtke, P. (1996). Functional Area Analysis of Intelligent Computer-Assisted Instruction. Training and Personnel Systems Science and Technology Evaluation and Management Committee. Unpublished white paper, United States Air Force Research Laboratory, San Antonio, Texas.

Van Joolingen, W.R. and De Jong, T. (1996). Design and implementation of simulation-based discovery environments: the SMISLE solution. *International Journal of Artificial Intelligence in Education*, 7, 253-276.

Zachary, W., Cannon-Bowers, J., Burns, J., Bilazarian, P. and Krecker, D. (1998). An Advanced Embedded Training System (AETS) for Tactical Team Training. *In Intelligent Tutoring Systems - 4th International Conference*, San Antonio, August, 544–553.