SOFTWARE INSTRUMENTATION FOR INTELLIGENT EMBEDDED TRAINING

Brant A. Cheikes and Abigail S. Gertner
The MITRE Corporation
Bedford, Massachusetts

Abstract

Software applications play a critical role in many work environments. These applications may be general purpose, such as word-processing and spreadsheet tools, or tailored to specific mission functions, such as systems for air-traffic management and military "command and control" (C^2). End-user training is critical if these applications are to be adopted and used effectively. With OPTEMPO up, and training budgets under constant pressure to "do more with less," it is more important than ever to bring training to the users and enable them to learn whenever they have time, wherever they may be. One way to accomplish this is to embed a training system in the mission application itself. Such *embedded training systems* (ETSs) have been used to varying degrees throughout the military services, and the United States Army has mandated the use of embedded training techniques for all new systems it procures.

Experience has shown that application-operation skills are learned best when trainees are given extensive "hands-on", interactive coached practice on the mission application to be used on the job. Our research at MITRE focuses on developing ETSs that approximate the advantages of one-to-one expert human tutoring through the use of intelligent computer-assisted instruction (ICAI) techniques. For ICAI-based ETSs to support interactive coached practice, they must have some means of observing both the trainee's actions on the mission application, and the application's response(s) to those actions, and also a way to take control of the mission application for the purpose of demonstration or to set up the initial environment for training.

To provide this service, MITRE has developed a technique called *software instrumentation*, whereby we non-invasively modify the mission application's computing environment (rather than the application itself) and thus gain the required forms of access for our ETS. We discuss this general technique and our implemented software instrumentation tools for X-Windows and standalone Java applications.

Biographical Sketches:

Brant A. Cheikes is a Principal Scientist and Associate Department Head at The MITRE Corporation, in the Information Technology Center's department of Information Management & Instructional Systems. He holds a B.S. degree in Computer Engineering from Boston University, and M.S.E. and Ph.D. degrees in Computer and Information Science from the University of Pennsylvania. Since joining MITRE in 1993, Dr. Cheikes has both supported and directed research in advanced instructional technologies, including student modeling, modular training-system architecture, and intelligent embedded training. His current research focuses on the design of an intelligent embedded training agent that coaches trainees using spoken natural-language dialogue combined with on-screen pointing gestures.

Abigail S. Gertner is a Lead Scientist at The MITRE Corporation, in the Information Technology Center's department of Information Management & Instructional Systems. She received her A.B. in Psychology from Harvard University, and her M.S.E. and Ph.D. in Computer and Information Science from the University of Pennsylvania. Her research interests are primarily in the areas of plan recognition, user modeling, and cooperative response generation in decision support and intelligent tutoring systems. She has pursued these interests in the context of a decision support system for emergency medicine, and an intelligent tutoring system for university Physics. She is currently developing an intelligent embedded training agent.

SOFTWARE INSTRUMENTATION FOR INTELLIGENT EMBEDDED TRAINING

Brant A. Cheikes and Abigail S. Gertner
The MITRE Corporation
Bedford, Massachusetts

INTRODUCTION

Software applications play a critical supporting role in most military work environments. These applications may be general purpose, such as word-processing and spreadsheet tools, or tailored to specific mission functions, such as systems for airspace management or military command and control (C^2). End-user training is critical if these applications are to be adopted and used effectively in the field. With OPTEMPO up, and training budgets under constant pressure to "do more with less," it is more important than ever to bring training to the users and enable them to learn whenever they have time, wherever they may be located. One way to accomplish this is to embed a training system in the mission application itself. Such *embedded training systems* (ETSs) have been deployed to varying degrees throughout the military services, and the United States Army has mandated the use of embedded training techniques for all new systems it procures (Sherman, 2000). (Although we say the training system is "embedded" in the mission system, we generally mean that the mission system and the training system are distinct, interoperating software packages running in the same computing environment, and resident on the same computer workstation.)

Experience has shown that software-operation skills are learned best when trainees are given extensive "hands-on", interactive coached practice on the mission application to be used on the job. Our research at MITRE focuses on developing ETSs intended to approximate the effectiveness of one-to-one expert human tutoring through the use of *intelligent computer-assisted instruction* (ICAI) techniques. We believe that artificially-intelligent ETSs will be able to coach novice users to expertise, and do so quickly, conveniently, reliably and, for sufficiently large student populations, at lower cost per student than human-led methods.

Can ICAI Replace Human Instructors?

In any discussion of ICAI systems, the question inevitably arises of whether such systems can effectively replace human instructors. ICAI systems certainly promise a number of advantages. First, they can be available at any time, and deployed anywhere. Second, they can produce consistent and measurable positive learning outcomes. Third, they can optimize their instructional services to best fit the needs of each learner, whereas human instructors in classroom settings are forced to teach to the mythical average student, thus leaving some students to struggle while others are insufficiently challenged. Fourth, by optimizing their instruction, ICAI systems offer training services that are significantly more engaging and motivating than conventional presentation-oriented Computer-Based Instruction (CBT), and often can help students achieve desired training results in markedly less time than classroom approaches permit.

Despite these advantages, over the near term at least, replacement of human instructors with ICAI systems is neither desirable nor likely. Expert instructors bring unique knowledge, skills and experience to the learning process, and can motivate and mentor students in ways both subtle and profound, well beyond the capabilities of current and emerging ICAI methods. Yet this teaching expertise comes at a stiff cost: contrary to the adage that "those who can, do; those who can't, teach", expert instructors are typically experts in the subject matter, and the time they spend teaching others, however personally rewarding it may be, is time taken away from doing what they do best.

Rather than replacing expert instructors, we see ICAI systems as complementing them when they are available, and providing a helpful substitute when they are not. In their complementing role, ICAI systems can serve as "force multipliers" for expert instructors. For example, in a classroom setting, students can work in a self-paced manner with an ICAI system while the instructor is free to observe and provide one-to-one support as needed in areas where the system's instructional ability is limited or inadequate. During periods when instructors are unavailable, the ICAI system still can provide valuable learning support.

ICAI—A Highly Interactive Approach to Training

ICAI systems have two key features from which their instructional advantages are derived: *interactivity* and *adaptivity*. (A high degree of interactivity is partly what distinguishes ICAI approaches from commercial CBT delivery systems.) ICAI systems focus on helping students *learn by doing*. Students perform realistic

practical exercises with coaching and feedback from the ICAI system. During these exercises, students interact with appropriate problem-solving tools and with the ICAI courseware (e.g., to request help or advice). For example, the LISP Tutor (Anderson *et al.*, 1989) teaches computer programming using the LISP language; students write LISP programs in a text editor, while the tutoring component corrects their syntax and debugs their algorithms. As another example, the Sherlock tutoring system (Katz *et al.,* 1992) teaches device troubleshooting; students attempt to diagnose and repair a simulated piece of test equipment while the tutoring component coaches them on their diagnostic technique.

ICAI system interactivity is supported by sophisticated *student modeling* algorithms, which develop detailed assessments of students' learning progress and achievements, based on continuous observation of their problem-solving activity during practical exercises. ICAI systems use their student models to adapt their instruction to best fit each student's learning needs. This adaptation may involve both course sequencing—selecting and sequencing lessons and exercises to keep students challenged and working in their "zone of proximal development" (Vygotsky, 1978)—and strategic alterations of instructional behavior, such as increasing or reducing direct coaching support, or allowing students more freedom to explore (and learn from) unproductive problem solving approaches.

In ICAI systems, interactivity and adaptivity work hand in hand. Interactivity provides the tutoring component with many opportunities to observe students in detail as they apply knowledge and skills to specific problem-solving tasks. These observations feed the development of the system's student model, which drives the system's instructional adaptations.

Supporting Interactivity and Adaptivity in ETSs

For ICAI systems to observe students while they perform practical exercises, the problem-solving tools must be able to report, more or less in real time, each problem-solving step performed by the student. Ritter and Koedinger (1995) have argued that problem-solving tools need to be *observable*, *inspectable*, and *scriptable* if they are to be used as teaching tools by an ICAI system. They define a software product as *observable* if it is able to report to an external application all actions taken by the user. They define software as *inspectable* if it is able to respond to requests for information about its internal state, and s*criptable* if it can execute all available command functions from an external application as if they were invoked by the user.

In ETSs that employ ICAI methods, the mission application itself serves as the problem-solving tool, and thus it needs to be observable, inspectable, and scriptable. Unfortunately, few military software applications today are engineered to expose suitable interfaces to external systems such as ETSs. (Indeed, few software applications in general are so engineered; Microsoft's suite of office-automation applications are among those that have extensive interfaces supporting interaction with external applications.) To overcome this obstacle, MITRE has developed a technique called *software instrumentation*, whereby we non-invasively modify the mission application's computing environment (rather than the application itself) and thus gain the needed forms of access for our ETS. In this paper, we discuss this general technique and describe our implemented software instrumentation tools for X-Windows and standalone Java applications. We conclude with some examples of how we are applying software instrumentation techniques to develop ICAI-based embedded training system prototypes.

SOFTWARE INSTRUMENTATION

The mechanisms that provide observation, inspection and scripting services can be thought of as comprising a distinct software layer, which we call the *application interface* layer. Thinking of these services as forming a layer allows us to ask whether the layer is part of the mission application, or external to it. In Ritter and Koedinger's tool-tutor model, this layer is implemented internal to the problem-solving tool. That is, the tool incorporates a programming interface that allows external applications to subscribe to reports of user actions on the tool, to issue queries on tool state variables, and to execute tool functions. When integrated into the tool, the interface layer permits communication between the tool and an external tutoring application 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 applications could refer to individual cells by their row and column addresses.

Building an interface layer into a problem-solving tool requires a significant amount of engineering labor above and beyond the design and implementation of the tool's core functionality and user interface. 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 tools), the lack of standards in this area leads to a situation in which interfaces differ greatly across both tools and vendors.

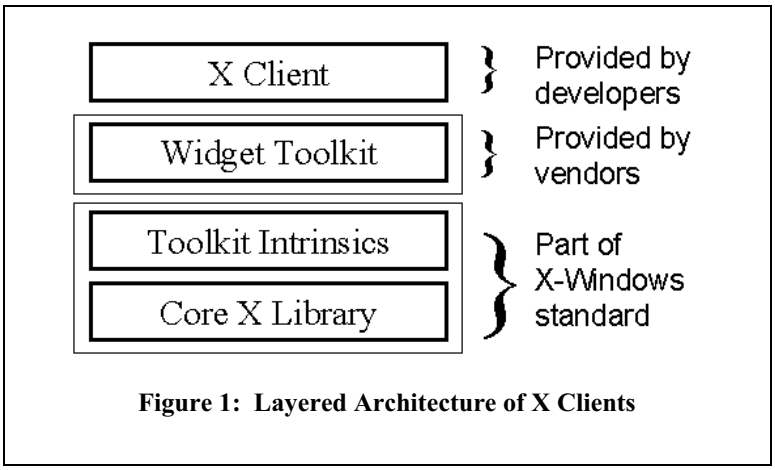

Figure 1: Layered Architecture of X Clients

We have been investigating an alternative approach, namely, implementing the application interface layer as an independent component external both to the mission application and to the tutoring application (in our case, an ETS). Our technique is based on the idea of exploiting software "hooks" in the mission application's computing environment, which we will discuss shortly. This approach has two advantages: (1) it has the potential to apply to any mission application that runs in the computing environment, and (2) the interface exposed would be consistent across all monitored applications. The disadvantage is that communication between the mission application and the ETS can no longer occur in application-specific terms, since the requisite knowledge resides only inside the 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, has been sufficient to enable our prototype ETSs to provide detailed guidance down to suggestions of specific user-interface gestures.

In the following subsections, we describe prototype software instrumentation tools that MITRE has developed for X-Windows applications under Unix, and for standalone Java applications. We conclude with a discussion of some industry trends that promise to make software instrumentation services ubiquitous in the computing industry.

Software Instrumentation for X-Windows Clients

To understand how software instrumentation of X-Windows applications can be accomplished, it is necessary first to understand a bit about the X-Windows system itself. X-Windows is a portable, network-transparent windowing system that has become the base technology for the implementation and management of graphical user interfaces for software applications running in multitasking computing environments such as Unix (Mansfield, 1993). X was created at the Massachusetts Institute of Technology, and early development was sponsored by the Digital Equipment Corporation and International Business Machines. Since the release of X-Windows Version 11 (X11) in 1987, the standard has been further developed by the MIT X Consortium, the X/Open group, and most recently by X.Org, an industry consortium of hardware and software vendors committed to maintaining and enhancing the X-Windows platform. All major vendors of Unix systems (e.g., Compaq, Hewlett-Packard, IBM and Sun Microsystems) supply a version of X as part of their base operating system software.

X-Windows applications (commonly referred to as "X clients") are built on several service-providing layers, some of which are provided standard with X-Windows software releases, others of which are provided by third-party vendors. This layered architecture is illustrated in Figure 1.

The lowest layer of X-Windows services is the core X protocol library, also called Xlib. This layer provides basic mechanisms for accessing displays and attached devices such as the keyboard and mouse, and for managing multiple, independent and possibly overlapping regions of screen "real estate" called windows. It supports a standard protocol for inter-process communication over a network between the client and a special system process called the X Server. The X Server carries out the client's requests for window-based operations on the display screen, and passes back to the client notification of all hardware events such as mouse actions and keypresses. Xlib services are neutral by design to the "look and feel" of the client application, i.e., the appearance and operation both of the visual "decorations" used to move, resize, iconify, and otherwise control windowed applications, and of the suite of interface elements that comprise an application's GUI.

The concept of an interface element—also called a *widget*—is introduced at the *Toolkit Intrinsics* layer, also called Xt. This layer provides a collection of functions that may be used to define classes of widget types (such as buttons, menus, checkboxes, and so forth) and specify their appearance and behavior. It is important to understand that this layer does not actually define a particular set of widget types, but rather provides only the mechanisms need to define widget types. These mechanisms are implemented through appropriate sets of calls to Xlib services. Both the Xlib and Xt layers are part of the X-Windows standard, and are implemented as external function libraries to which client applications are linked.

Collections of widget types sharing a common look and feel are called *widget toolkits*. These toolkits are created by calling services provided by Xt. Although there are no truly standard widget toolkits, the *Motif* widget toolkit, originally created by the Open Software Foundation (OSF) and now licensed by The Open Group, has become a *de facto* industry standard. Motif is included by most Unix platform vendors in their base operating system releases, and is widely used throughout industry.

A software developer's X-client (e.g., a military command and control application) typically constructs and manages its GUI by directly calling upon the services of a widget toolkit such as Motif. These calls to the widget toolkit are translated into calls to Xt services, which in turn call upon Xlib services.

Assuming one had access to the source code of any of these service layers, it would be technically possible to add programming instructions to them and thus effectively intercept the stream of GUI-management service calls. This is the approach to X-Windows software instrumentation we are pursuing. To prove this concept, we have developed a software instrumentation tool called WOSIT—Widget Observation, Scripting and Inspection Tool.

WOSIT

It turns out that a sample implementation of the X-Window system is maintained as open-source software by X.Org. The latest release of the X-Windows sample implementation is X11 Release 6.6, which can be downloaded free of charge from the X.Org website (X.Org, 2001). This implementation contains source code for Xlib and Xt. Although the proprietary implementations of X provided by Unix platform vendors may vary in quality and efficiency, because X has been standardized to support interoperability, all commercial implementations provide the same collection of services and conform to the same functional interfaces. Now consider what this means for an X client running in, e.g., Sun's Solaris operating environment using Sun's implementation of X. Because of the standardization of the X protocol, it is possible to substitute X.Org's open-source version of Xlib or Xt in place of Sun's proprietary implementations, with no significant impact on the X client's operation. Moreover, given the standard practice of dynamically linking X client applications with required system libraries (including the X libraries that implement Xlib and Xt), it is possible to make this switch transparently to the client itself, without access to the client's source code, and without recompiling or relinking the executable image. (Statically linked clients will require relinking, but will not need to be recompiled.)

WOSIT is an X client application that runs independently of the mission application. It works by modifying the X-Windows environment in which the mission application operates, and thereby gains access to all the X-Windows functions that the mission application itself can perform. WOSIT can intercept widget state-change notifications, query widget states, and even generate widget events (e.g., button clicks) as though they were performed by the user. It alters the X environment by substituting a modified version of Xt for the standard version supplied by the computing platform vendor. For dynamically linked applications, this substitution occurs at runtime. MITRE has released WOSIT to the public; it may be downloaded free of charge from MITRE's website (WOSIT, 2001).

The modified version of Xt required by WOSIT extends the X standard in two ways: (1) it adds a general-purpose rendezvous mechanism (Xtea—for Xt "external agent") enabling WOSIT to discover WOSIT-compatible X clients running in the environment and establish a direct communication channel with them, and (2) it defines a protocol called RAP—the Remote Access Protocol—that WOSIT uses to issue requests to and receive responses from the applications to which it connects as an external agent. Both Xtea and RAP were originally developed at Georgia Tech as part of the Mercator research project (Edwards, Mynatt & Rodriguez, 1993; Mynatt & Edwards, 1992), and were later revised by Will Walker at Digital Equipment Corporation. Both Xtea and RAP have been publicly released, and MITRE has since made a number of bugfixes and extensions to them as part of the WOSIT development effort.

WOSIT Services

Once WOSIT has established a RAP connection to the X client to be monitored, it exposes a simple text-based message interface of its own using network sockets. In

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

Figure 2: WOSIT Observation Stream

essence, WOSIT serves as a piece of "middleware" that mediates between an application to be monitored (such as a military command and control application) and one or more service-provider applications (such as an ETS).

WOSIT's most useful service is its ability to observe and report the details of user actions involving the widgets in a monitored application's GUI. Each time WOSIT notices that a GUI widget has undergone a significant state change, it sends a descriptive message to all connected service providers. Figure 2 illustrates a sequence of observation messages issued by WOSIT.

Each line in the figure reports a single widget event on the GUI of the monitored application. The first token (outside the parentheses) indicates the kind of widget that was manipulated by the user; the first line, for example, indicates that a pull-down menu was used. The second token indicates the type of event that occurred, e.g., a *press* event on a pull-down menu. The third token indicates the window in which the event occurred (using the text in the window's title bar), and the fourth uniquely identifies the widget within that window. The fifth argument indicates whether the event was initiated by the user or by the system, and the sixth is a timestamp in seconds counting from the time WOSIT began operating. So the first line in the figure reports that the user pressed the "Open…" item in the "View" menu, in a window titled "State Tools", and did so five seconds after WOSIT began running. Should a message contain additional arguments beyond the timestamp, these provide additional information unique to the event type. For example, the penultimate line in the figure reports that the user selected the item "tango2.view" from a list widget identified as "F_List" (file list), which was contained in the "Select File" window, and performed nineteen seconds after WOSIT began operating.

As the figure illustrates, WOSIT uses symbolic labels to refer to both 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 monitored application (assuming the structure of the GUI remains unchanged), and unique within each window.

WOSIT provides extensive support for inspection. Using its inspection services, WOSIT clients may issue a wide variety of queries pertaining to an application's GUI, including these:

- list the currently open windows;
- list the widgets contained in a specific window;
- report the current value of a widget (e.g., the contents of a text field, or the on/off state of a radio button);
- report the size and location of a widget.

WOSIT's support for scripting is currently limited. We have implemented one demonstration capability: the ability to cause selected widgets on the GUI of the monitored application to flash briefly (in order to draw the user's attention to them).

Full details on WOSIT, including a User's Manual, is available from the WOSIT website (WOSIT, 2001).

JOSIT

JOSIT is a variation on the WOSIT theme, an implementation of the GUI instrumentation concept for standalone Java applications (as opposed to applets running within a web browser). It differs in the details of its implementation, but provides the same services (observation, inspection, and scripting) using the same communication methods and protocols. The hooks that JOSIT exploits are provided by Sun Microsystem's accessibility API.

Like WOSIT, JOSIT has been publicly released and can be downloaded free of charge from the JOSIT website (JOSIT, 2001). Interested readers are referred to the website for complete details on JOSIT, including extensive documentation.

Related Trends in Industry

Recent Federal regulations require agencies of the United States Government to provide accessible Electronic and Information Technology (E&IT) products to their employees who are individuals with disabilities, as well as to ensure that individuals with disabilities who are members of the general public have access to

agency products comparable to that of individuals without disabilities (Access Board, 2000). A crucial feature of the rules is that they both mandate Government compliance, and provide avenues for affected individuals to seek recourse through civil litigation. Consequently, the Federal rules are acting as a "big stick" driving the E&IT industry, in concert with vendors of assistive technology, to offer accessible products. Both Microsoft and Sun have undertaken high-visibility efforts to develop new designs and technologies that will make their products broadly accessible to individuals with disabilities (Microsoft, 2001; Sun Microsystems Corporation, 2001), and many other vendors are following suit, making significant investments in accessible product design.

For individuals with low vision or blindness, assistive technology is crucial for them to make effective use of modern information systems. A common assistive device for visually-impaired individuals is a "screen reader", a software application which converts visual displays into descriptive spoken-language utterances (using commercial speech-synthesis technology). For individuals with various forms of physical impairment, voice-navigation tools enable them to navigate displays and manipulate software applications.

It turns out that assistive technologies such as screen readers and voice-navigation systems need at least the same forms of access to GUI state and event information that tools like WOSIT and JOSIT provide.

Driven by the urgent need to meet Federal accessibility requirements, industry is rapidly moving towards a set of emerging technical standards that promise to make GUI instrumentation services ubiquitous. For example, Microsoft's Active Accessibility effort aims to support a model in which "applications, referred to as servers, provide information about the contents of the computer screen [… and] Accessibility aids, referred to as clients, use Active Accessibility to obtain information about the user interface of other applications and the operating system" (Microsoft, 2001). Sun has provided extensive support for designing and building accessible Java applications (Meloan, 2000), and the GNOME Accessibility Project is actively developing accessibility support for the next-generation window-oriented graphical desktop environment for Unix and Linux platforms (GNOME, 2001). Although the details of the interfaces differ, these accessibility-support mechanisms are providing WOSIT/JOSIT-like observation, inspection, and scripting services.

Today, adding GUI instrumentation to a complex software application can only be done using niche tools like WOSIT and JOSIT, or by employing costly application-specific methods. This, we believe, has severely restricted the opportunities for widespread development and deployment of intelligent embedded training systems. As support for E&IT accessibility matures, we believe that the associated operating-system services will become widely available, thereby removing a major impediment to the widespread development of embedded training tools.

Summary

This section introduced and described the concept of software instrumentation, and presented in detail a technical approach to instrumenting the GUI of an X-Windows client application. We described WOSIT, a prototype instrumentation tool for X clients, and JOSIT, a related tool for use with standalone Java applications. We concluded with a discussion of industry trends that promise to make GUI instrumentation services widely available.

In and of themselves, neither WOSIT nor JOSIT are embedded training tools. Rather, they are critical enabling technologies for "intelligent" embedded training systems, ETSs that employ ICAI techniques to provide highly interactive and adaptive training support to users of complex software applications. In the next section, we discuss two ETS prototypes we have developed that take advantage of the GUI instrumentation services provided by WOSIT and JOSIT.

EMBEDDED TRAINING SYSTEM PROTOTYPES

As we have suggested, GUI instrumentation is a key enabling technology for ICAI-based ETSs. Using the WOSIT instrumentation tool, we have created a simple ETS demonstration prototype called the *Active Checklist*, which we describe next.

Active Checklist

Checklists are useful job aids, particularly when workers are expected to follow approved, validated problem-solving procedures. These checklists, when they exist, typically are found in printed manuals. Web technology offers the potential for checklists to be widely deployed in easy-to-use, browsable form, and indeed, some military C2 systems are arriving with such web-based checklists. One problem with online checklists is that users must navigate them manually, not only selecting the appropriate checklist for their task, but also keeping the checklist synchronized with their progress in the task. GUI instrumentation, combined with some simple, well-understood artificial intelligence techniques (task modeling and plan recognition), offers the potential to develop active, "intelligent" checklist tools

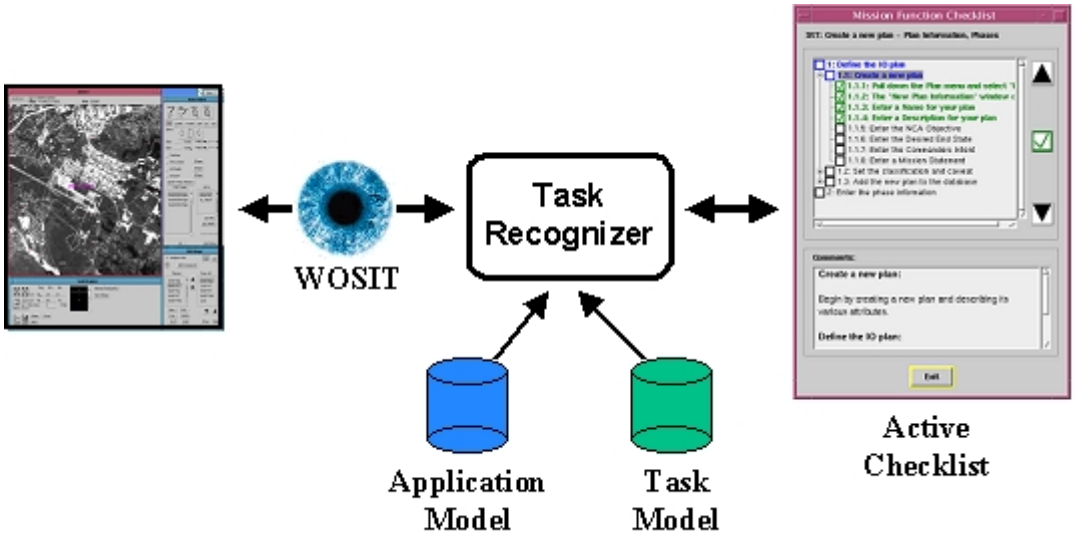

Figure 3: Active Checklist Architecture

that automatically track the problem-solving activities of system operators and keep themselves synchronized with the operators' location in the task.

To prove this concept, we have developed a prototype system having the architecture illustrated in Figure 3. At the beginning of a procedure, the user selects a checklist from the Checklist Browser, after which the Active Checklist appears (shown on the right in the figure). The Active Checklist tool displays the problem-solving process in hierarchical form, as shown in Figure 4. When a step in the Checklist is selected, explanatory text associated with that step is displayed in the lower panel. Icons beside each step indicate whether the step is mandatory, optional, or for information only. As the user performs the procedures on the mission application, WOSIT reports the activity to the Active Checklist, which updates its display accordingly. When a step is begun, the Checklist highlights it in blue. When a step is completed, the Checklist checks it off and highlights it in green. If a step is performed incorrectly, the Checklist highlights it in red. Steps that have been skipped are easily identified by the absence of markings and coloring.

How it works. A detailed description is beyond the scope of this paper. The mission application (shown on the left in the figure) is instrumented using WOSIT, which reports user actions to the Task Recognizer. The Task Recognizer refers to two databases in the process of tracking the user's progress in the procedure: the *application model* and the *task model*. The application model maps patterns of user-interface gestures (as would be reported by WOSIT) to more abstract event

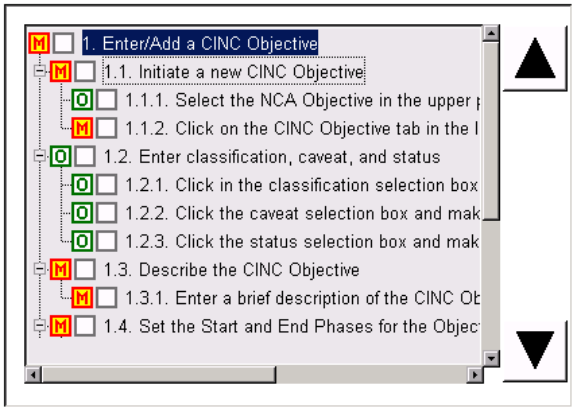

Figure 4: Checklist Hierarchy

types called *application commands*. For example, this sequence of events:

1. User performs: menu press File->Open
2. System performs: window labeled "Open" appears.
3. User performs: select "myfile.doc" in File list.
4. User performs: button press Open.

could be mapped to an application command called **OpenFile.** Application commands define the basic units of activity in a software application; the user-interface gestures are, in essence, the "words" that users combine to create requests for action from the application. The application model is constructed using a companion authoring tool, and need be done only once per application (although maintenance may be required if the application's GUI is modified).

The task model defines tasks in increasing degrees of abstraction, building upon application commands as their foundation. For example, the task model might define the **PrepareProject** task as a sequence of application commands: **OpenProject**, **DisplayNavaids**, **DisplayFixes**, **DisplayRunways**, **LoadARTSData**. The task model is also constructed using a companion authoring tool, suitable for use by subject-matter experts.

Training concept. The Active Checklist can be used either as a job aid or as a training tool. When used as a training tool, users may select from the checklist catalog a procedure they need to learn, then manually click through each step displayed by the Checklist, performing each step as described. The Checklist will automatically update its display to indicate correct or incorrect completion. Once a user has achieved basic familiarity with a procedure, they may practice it again, this time with the Checklist iconified, i.e., active but off-screen. They may practice the procedure to the extent they are able, while the Checklist tracks them in the background. If they get stuck, they can re-display the Checklist, and it will be positioned where they currently are in the procedure, with correct, incorrect and skipped steps visually indicated.

The Active Checklist is a simple example of an ETS that exploits GUI instrumentation. It is relatively easy to author new checklists for a given application. The learning support it provides, however, is limited. Training effectiveness depends on user initiative and motivation, as the tool provides only indirect tutoring support. To explore the design of ETSs that provide significantly more tutorial benefit, we are currently developing a more interactive *embedded training agent*, described next.

Embedded Training Agent

A major obstacle to the design of effective training systems is the design of the user interface. Many ICAI systems require students to learn a specialized user interface, which can place an unacceptable cognitive load on a student who is already struggling to learn a new subject. This problem is particularly salient in the case of embedded training because the task that the student is engaged in learning is precisely the use of a complex GUI. Having to learn one user interface in order to learn how to use another seems quite infelicitous. (This is another disadvantage of the Active Checklist approach.) Natural language dialogue is an attractive potential solution to the user interface problem for embedded training because it allows students to interact with the tutor in a familiar manner, thus reducing the overall cognitive load. The use of speech input and output (made possible by the system's ability to carry on a dialogue) would further ease the burden of communication.

With this in mind, we are developing an embedded training agent that engages the student in a multimodal, mixed-initiative spoken natural-language dialogue, and pursues explicit instructional goals to provide individualized coached practice on realistic problem-solving exercises (Cheikes & Gertner, 2001). Other motivations for using dialogue in our ETS include both the ability to implement a wider range of instructional strategies than would be possible with a structured user interface, and the ability to change strategies smoothly by informing the student of what is happening using natural language cues (e.g., "No, that's not quite right. Let's try a different approach…"). In addition, the ability to refer to a model of the ongoing discourse provides the agent with the opportunity to adapt its tutoring to the discourse context. The agent can refer to the discourse model, which incorporates both natural language utterances and GUI actions, to help select an instructional strategy, to determine whether its current strategy is succeeding or failing, and to decide whether and when to change strategies.

To provide a discourse model for our tutorial agent, we are using Collagen, an application-independent collaboration management platform developed by the Mitsubishi Electric Research Laboratory (Rich & Sidner, 1998). Collagen provides tools for dialogue management which, when given a task-specific *recipe library*, will perform plan recognition, track the focus of attention in the tutor-student interaction, and maintain an agenda of actions that could complete the plan. In addition, Collagen provides interface elements to support dialogue interaction between an agent and a user, in-

cluding optional support for speech recognition and generation.

At this point, Collagen does not do any natural language understanding, so the user is required to produce utterances that can be interpreted as part of an artificial discourse language (Sidner, 1994) that includes the types of utterances people use when collaborating on tasks. The user can construct these utterances using a structured interface provided by Collagen, or by speaking in speech-enabled Collagen applications. Collagen also provides facilities that allow pointing gestures to be integrated with the agent's repertoire of communicative acts.

For the initial prototype of our ETS, we are working with an application called TARGETS (Terminal Area Route Generation, Evaluation, and Traffic Simulation), a tool for designing new arrival and departure routes into and out of airports. TARGETS is a Java application developed by MITRE's Center for Advanced Aviation Systems Development. To use Collagen for embedded training on TARGETS, we instrumented the TARGETS application's GUI using JOSIT.

Instrumentation enables a number of the tutorial behaviors of the Collagen agent. First, using the observation capability of JOSIT, the embedded training agent is able to track the student's behavior on the application interface, and integrate new actions into its model of the current domain plan. JOSIT's inspection mechanisms are used for two main functions: first, to determine the location of elements of the TARGETS GUI on the screen so that the agent can point to them with its movable pointing hand, and second, to maintain a model of the application state which the agent uses to determine the applicability of potential actions. Finally, the scripting capability in JOSIT is used to allow the agent to actually perform actions on the TARGETS interface. This may be done to demonstrate a new procedure to the user, as well as to change the application state to get it ready for a new tutoring exercise.

CONCLUSIONS

This paper has described ICAI-based embedded training systems, and discussed a key enabling technology called GUI instrumentation. We have presented technical details on two GUI instrumentation tools called WOSIT and JOSIT, which apply to X-Windows clients and Java applications respectively. Finally, we summarized two demonstration prototypes we have created at MITRE that illustrate how GUI instrumentation services might be employed to support the delivery of highly interactive training services to users of complex mission applications.

The key conclusion of our work is that GUI instrumentation makes intelligent, interactive ETSs technically feasible. We have shown how GUI instrumentation may be added to a mission application without requiring any supporting modifications to the application itself. This means that mission applications may be augmented with ETSs at any time, and that the ETSs may be developed by any qualified, experienced third party, not necessarily by the prime contractor of the mission application. This will become even more true as accessibility-support services become widespread in the computing industry.

Limitations

Although GUI instrumentation tools such as WOSIT and JOSIT make ICAI-based ETSs possible, there remain technical limitations that may affect the ETSs ability to provide detailed tutorial support on all aspects of mission-system operation.

Recall that GUI instrumentation tools work by intercepting system messages pertaining to GUI widget states and behaviors. What this means is that consumers of GUI instrumentation data only obtain a surface-level view of the mission application. For example, a GUI instrumentation tool cannot "see" into (i.e., directly access) any databases used by the mission application; it can only see portions of those databases if and when they are displayed somewhere on the application's GUI.

So-called "situation displays" pose a similar problem. Situation displays are typically implemented using a generic "canvas" widget. The contents of the situation display (e.g., multi-layered maps, imagery, etc.) are bit-mapped renderings of entities known only to the application itself. A GUI instrumentation tool only has access to the pixel-by-pixel representation of the situation; only the application knows that it is displaying, e.g., imagery over Sarajevo with iconic indications of recent ELINT emission collections.

User-settable preferences and configuration parameters may also be inaccessible to GUI instrumentation tools if they are not continuously displayed somewhere on the application's GUI (and most are not). If these settings have pervasive effects on the way the application operates, an ETS may have difficulty in interpreting user actions properly if it lacks access to the settings.

Thus far we have found that we are able to work around most of these limitations through clever instructional design. Since the purpose in using an ETS is to achieve training goals, we can design our exercise scenarios in a way that forces the trainee to reveal through their actions those elements of information that are inaccessible

to the GUI instrumentation tool. It is also possible to design the ETS to directly question the trainee as needed. Such questioning need not be intrusive, and can even serve instructional objectives, such as having the trainee explain their actions in order to verify they understand the relevant concepts.

It remains an open question whether GUI instrumentation as described in this article can truly support all the needs of ICAI-based ETSs, and ongoing MITRE research is investigating this issue. We suspect, however, that even if a "hybrid" solution should be required, in which GUI instrumentation is combined with application-specific modifications, GUI instrumentation would still dramatically reduce the overall cost of enabling embedded training as compared to the cost of implementing a complete, application-specific set of interfaces internal to the mission application.

REFERENCES

Access Board. (2000). *Electronic and Information Technology Accessibility Standards: Final Rule*. (2000). Federal Register, **65**(246): 80500-80528. http://www.access-board.gov/news/508-final.htm.

Anderson, J. R., Conrad, F. G., & Corbett, A. T. (1989). Skill acquisition and the LISP Tutor. *Cognitive Science*, **13**, 467-506.

Cheikes, B. A. and Gertner, A. S. (2001). Teaching to Plan and Planning to Teach in an Embedded Training System. In Moore, J. D., Redfield, C. L., and Johnson, W. L., editors, *Artificial Intelligence in Education*. Amsterdam: IOS Press, pp. 398-409.

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.

GNOME. (2001). *Gnome Accessibility Project*. http://www.sun.com/access/gnome/.

JOSIT. (2001). JOSIT Home Page. http://www.mitre.org/tech_transfer/josit.

Katz, S., Lesgold, A., Eggan, G. and Gordin, M. (1992). Modelling the student in Sherlock II. *International Journal of Artificial Intelligence in Education,* **3**, 495-518.

Meloan, S. (2000). *Accessibility and the Java Platform*. http://java.sun.com/features/2000/03/accessibility.jplat_ns.html.

Microsoft Corporation. (2001). *Microsoft Active Accessibility*. http://www.microsoft.com/enable/msaa/details.htm.

Mynatt, E. D. & 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.

Rich, C. and Sidner, C. L. (1998). COLLAGEN: A collaboration manager for software interface agents. *User Modeling and User-Adapted Interaction,* **8** (3-4):315-350.

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.

Sherman, K. O. (2000). Intelligent Tutor for Today's Army. In *Military Training Technology*, **5**(5). http://www.mt2-kmi.com/Archives/5_5_MT2/5_5_Art6.cfm.

Sidner, C. L. (1994). An artificial discourse language for collaborative negotiation. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, Menlo Park: AAAI Press, pp. 814-819.

Sun Microsystems Corporation. (2001). *Sun Accessibility Program*. http://www.sun.com/access/.

Vygotsky, L. S. (1978). Mind in Society: The Development of Higher Psychological Processes. Cambridge, MA: Harvard University Press.

WOSIT Home Page. (2001). http://www.mitre.org/technology/wosit.