DocWizards: A System for Authoring Follow-me Documentation Wizards

Lawrence Bergman, Vittorio Castelli, Tessa Lau, Daniel Oblinger
IBM T.J. Watson Research Center
19 Skyline Dr.
Hawthorne, NY 10532,USA
Tel: 1-914-784-7946
{bergmanl, vittorio, tessalau, oblio}@us.ibm.com

ABSTRACT

Traditional documentation for computer-based procedures is difficult to use: readers have trouble navigating long complex instructions, have trouble mapping from the text to display widgets, and waste time performing repetitive procedures. We propose a new class of improved documentation that we call *follow-me documentation wizards*. Follow-me documentation wizards step a user through a script representation of a procedure by highlighting portions of the text, as well application UI elements. This paper presents algorithms for automatically capturing follow-me documentation wizards by demonstration, through observing experts performing the procedure. We also present our DocWizards implementation on the Eclipse platform. We evaluate our system with an initial user study that showing that most users have a marked preference for this form of guidance over traditional documentation.

ACM Classification: H5.2 [Information interfaces and presentation]: User Interfaces. – Training, help, and documentation.

General terms: Documentation, Algorithms**,** Human Factors

Keywords: Documentation generation, programming-by-demonstration

INTRODUCTION

Knowledge about how to do things – install printers, fill out expense reports, configure the desktop, etc. – is an important resource for the modern computer user. Capture and dissemination of such procedural knowledge is typically through one of two mechanisms.

The first is scripts or wizards. For many applications these provide an excellent end-user experience, greatly simplifying potentially complex tasks by walking the user through a process, step-by-step. However, there are several problems with these forms of automation. First, they are laborious to author, and difficult to maintain. Second, they are often not robust to unforeseen conditions. Most computer users have had the uncomfortable experience of discovering partway through a wizard that either their own knowledge or the structure of the wizard is inadequate to permit further progress. Finally, scripts and particularly wizards serve poorly as tutorials. A user who wants to perform a task similar to the automated task, but with a few differences, receives little or no guidance from the wizard or script.

The other common form of procedural knowledge transfer is through documentation. Well-written documentation provides a user with a conceptual overview of the application model, as well as sequences of operations to perform common tasks. There is a very strong tutorial nature to documentation, overcoming the limitation of wizards and scripts. The downside is that documentation is frequently more difficult to follow. The user is burdened with associating descriptive elements within the documentation (either text or images) with the actual application UI. Furthermore, users may find it difficult to keep track of where they are in the document (see [1] for a discussion). This is particularly problematic when there is control logic in the procedure (e.g., a branch based on state of the UI or on the user's goal), or when unanticipated events occur. Finally, documentation is also difficult to produce and costly to maintain.

We propose a solution to the capture and dissemination of procedural knowledge that we believe embodies the benefits of both documentation and scripts/wizards. We call our solution follow-me documentation wizards. Follow-me documentation wizards provide live documentation – they continually show the user the current position in the procedure, highlight the relevant application controls, and can even automate portions of the procedure. They are designed to address problems users have with existing documentation, such as difficulty navigating the procedure structure, inability to locate onscreen objects mentioned in the text, and trouble interpreting conditional branches in the instructions.

Though the cost of constructing follow-me documentation wizards using traditional tools may be prohibitively expensive, we have developed a low-cost method for authoring

UIST'05, October 23–27, 2005, Seattle, Washington, USA.

based on demonstrating the procedure one or more times directly on the application GUI.

The major contributions of this paper are:

- A new class of procedure documentation which we call follow-me documentation wizards;
- Algorithms for constructing follow-me documentation wizards that support evolution of the procedure through multiple demonstrations as well as manual editing;
- An implemented follow-me documentation wizard system for the Eclipse platform, which we call DocWizards;
- The results of a preliminary user evaluation of the DocWizards system.

The paper begins with a scenario describing the use of our follow-me documentation wizard system, followed by a set of additional application scenarios. We then present an overview of the system functionality and features. This is followed by a discussion of the system architecture and a discussion of the results of our user study. Sections on related and future work conclude the paper.

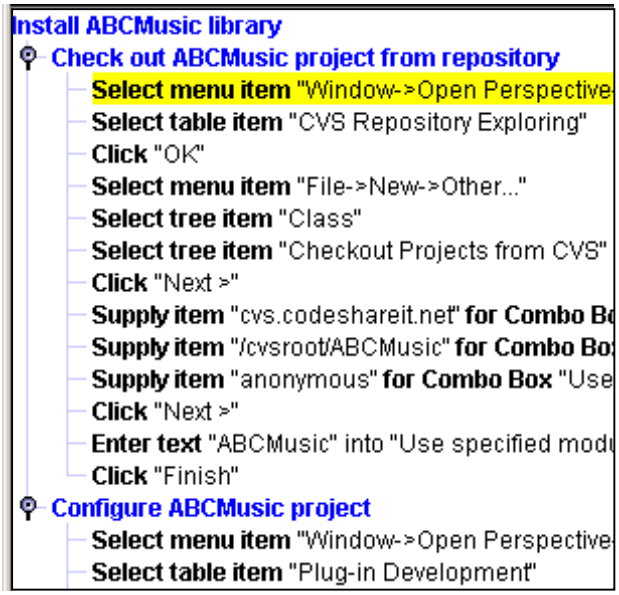
Terminology

Several terms will be used consistently throughout the paper. The term *application* will be used to refer to the software on which a procedure is being demonstrated or replayed. By contrast, the term *system* will be used to refer to our follow-me documentation wizard software. The terms *script* and *procedure representation* will be used interchangeably to refer to both the visual format of the procedure produced by the follow-me wizards system as well as its internal representation.

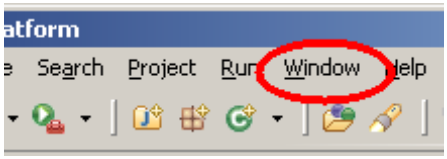
SAMPLE SCENARIO

We will illustrate the use of our follow-me documentation wizard system, which we call DocWizards, through a simple example. We begin by describing the process of authoring a follow-me documentation wizard, and then will describe the use of that wizard.

Authoring. Jane is lead developer for a newly-formed team tasked with developing a multi-media plug-in for the Eclipse application platform. Over the past several weeks, Jane has identified and installed on her own system several plug-ins and libraries that her team will be using as a development base. One of these installs is the ABCMusic utility library, an open-source project that her team intends to modify. The project is available from a CVS source repository at the open-source CodeShareIt website. In a single session, Jane records the steps to create a CVS repository location for CodeShareIt, the steps to check out the ABCMusic source code, and the steps to build the ABCMusic library. As she is recording, she adds some comments to the script, indicating the major subtasks. Once she has completed the task, she makes the script available in her team's local repository.



(a)



(b)

Figure 1. Initial recording and playback of a DocWizards script. (a) Portion of the initial script at the start of the second recording session. Note the highlighted step, showing the first predicted action to be performed. (b) Highlighting in the application UI of the widget corresponding to the predicted action.

Playback with additional authoring. Paul is the first member of the team to install ABCMusic. He grabs the script from the team repository and loads it into DocWizards.

Since he is the first to try the script, he decides to follow it fairly carefully, with recording turned on. When he starts playback of the script, the first statement in the script is highlighted (Figure 1a), and the "Window" menu item associated with that step is highlighted in his Eclipse application (Figure 1b). He notices the comment at the top of the script that this portion will check out ABCMusic from the code repository (located at codeshareit.net). Realizing that he needs to perform these steps, Paul presses the "Do single step" button in the DocWizards control panel, letting the system do the menu selection for him. The DocWizards system performs the menu selection, and then highlights the next line of the script, and the corresponding on-screen widget.

Paul continues in this manner, letting the DocWizards system performs actions until he notices that the next set of steps in the script will set up a CodeShareIt repository location. Since he already has one defined, he takes over, and opens his predefined repository location. As soon as the

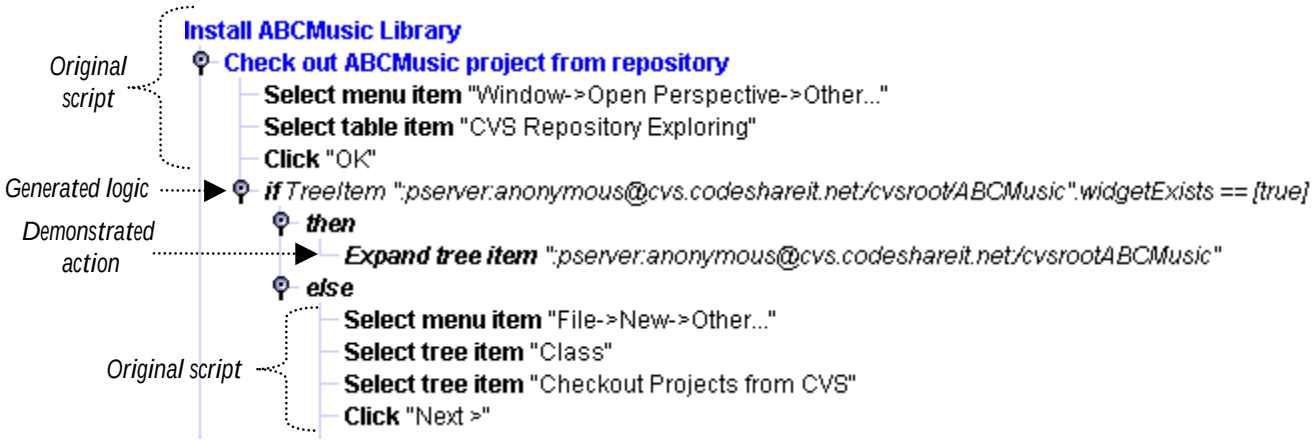


Figure 2. DocWizards script with automatically generated control structure. This script fragment shows the top portion of the script from Figure 1a, after adding an off-track action in a subsequent recording. The if-then-else statement is automatically inserted (and italicized) by the DocWizards learning component.

DocWizards system sees an performed action that does not correspond to the most recent predicted action, it modifies the script, inferring a conditional based on differences between the GUI state during Paul's demonstration and the original demonstration by Jane (Figure 2). In this particular case, DocWizards notices a tree entry describing the CodeShareIt repository location which was present during Paul's demonstration, but not during Jane's, and uses the existence/absence of this widget to distinguish the two demonstrations.

Note that this form of incremental update can be used whenever the existing script does not adequately cover current conditions, including recovery actions for unanticipated error conditions.

Paul continues to disregard DocWizards recommendations as he proceeds through a check-out process different than that demonstrated by Jane. Once the checkout is complete, he performs the same action that Jane did to begin configuration of the newly checked-out software. As soon as DocWizards sees an action that corresponds to the existing script, it moves forward in its predictions. Paul now returns to using the "do single step" control and quickly steps through the procedure with no further deviation from the original. At the successful conclusion of the procedure, Paul saves the updated script and stores it back into the team repository.

Playback. Joe is a new member of the development team. He checks out the ABCMusic installation script from the team repository and plays it back in DocWizards. The script has been through several authoring cycles at this point, and the accompanying note says that it is fairly well tested. Since he is relatively new to the Eclipse environment, however, he decides to go through the script slowly, performing the actions himself, to get a feel for what it is doing. He reads each line in the script as it is highlighted, along with associated comments that have been added by team members as the script has evolved. He easily locates the controls for each step in the script, and performs the action at each step. Because DocWizards tells Joe the action to be performed at each step in the procedure, he finds it very easy to get through the installation. Since he is particularly interested in learning, from time to time he does some exploration not described in the procedure, for example, navigating through installation components, occasionally opening and examining contents. DocWizards continues to suggest the next "on track" action while Joe does this, and when Joe returns to the path and performs the suggested action in the script, next-step predictions continue.

Ellen is a seasoned member of the team. She checks out the ABCMusic installation script from the repository, notes that it has been in use for a while, and simply presses the "do all" button. The script runs successfully and quickly to completion.

OTHER SCENARIOS

There are a variety of scenarios for which we think follow-me documentation wizards are particularly appropriate as replacement for traditional documentation, scripts, or wizards. These scenarios all assume an expert author, who is comfortable with reading and editing scripts and who is intentionally authoring the procedure. The end-user, on the other hand, can range from a complete novice to an expert who might assume an authoring role on-the-fly. Possible uses include:

Groupware procedures. The sample scenario described above can be characterized as development of a groupware procedure. Frequently groups within organization have procedures that are particular to that group. Such procedures are typically shared using written descriptions disseminated through email or collaboration tools, such as teamrooms or wikis. Often no one person is tasked with developing group procedures, and the resources to develop such procedures may be minimal. By providing for lightweight initial authoring combined with on-going evolution of the procedure, follow-me documentation wizards may provide an excellent alternative to traditional documentation.

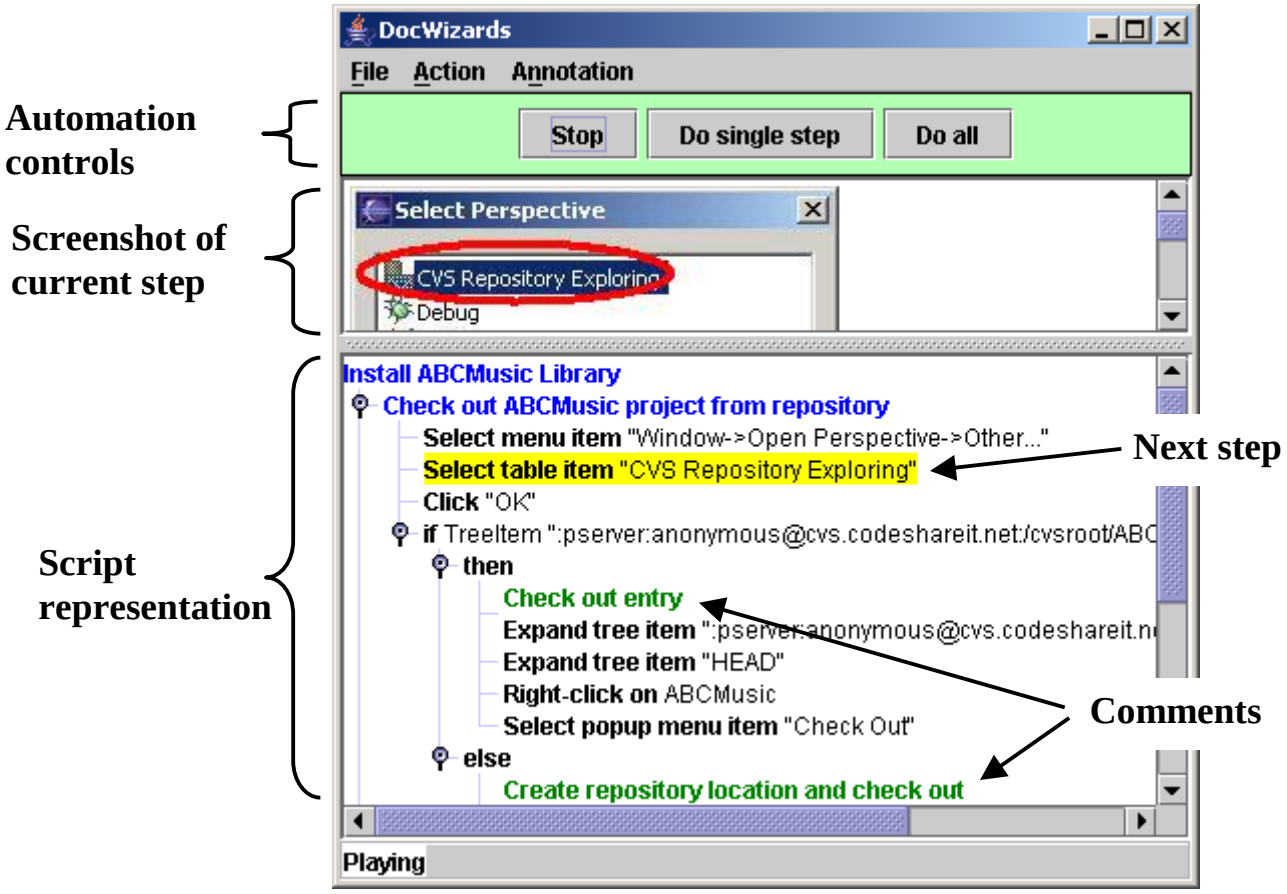


Figure 3. The DocWizards GUI. A previously authored script is loaded in playback mode. Notice that the suggested next step in the procedure is highlighted, with a corresponding screenshot that illustrates that step.

Guided walk-throughs. A common form of documentation is tutorials. Tutorials often instruct the user on how to perform a pre-specified goal by walking them through a sequence of operations. Doing this in-context, on the application interface itself, minimizes the need for a user to translate from what they are seeing during the tutorial to what they will be doing when using the application for real work.

Technical support. Technical support may be facilitated when the support personnel can operate directly on the user's desktop. A method for further enhancing this would be to record procedures for performing particular installations or repairs and then distributing these to end users. The author would be the expert technician, who would record and annotate several demonstrations of the same procedure in different end-user environments. Novice end-users would benefit from documentation that is easier to follow than traditional documentation and includes some automation, yet permits interleaving of "off-track" troubleshooting by support personnel.

SYSTEM FEATURES

The system we are describing, DocWizards, is a follow-me documentation wizard system implemented on the Eclipse platform [2]. The system is capable of recording and replaying actions performed on SWT widgets within Eclipse. Figure 3 shows a screenshot of the DocWizards system during playback of a previously authored procedure.

Learning from multiple demonstrations. Learning from multiple demonstrations is a key feature of DocWizards. By incorporating new demonstrations when the application UI changes (with new version releases, for example), or when previously unseen error conditions are encountered, procedures can evolve and remain current.

After the first demonstration of the procedure, subsequent demonstrations result in on-the-fly modificationd of the procedure structure that keeps it consistent with all prior demonstrations (unless the script has been manually edited, as we will discuss in the Architecture section). The immediate feedback of seeing the procedure representation update as actions are performed is a critical part of the authoring process. In the Architecture section we will discuss how multiple demonstrations are used to generate and maintain the script representation.

The wizards within our system are represented as textual scripts. Each script contains a set of actions such as "click

X" or "select list item Y". In addition, there may be control logic such as conditional branches or loops.

Editable procedure representation. Although the scripts are automatically generated, they are also editable. A learning system is often unable to perfectly infer the intent of an author from just a few demonstrations. To make the system full usable, the author must be able to change the script, either during the recording process, or at a later time. Edit operations currently supported include deleting steps, moving steps, and adding annotations. Our algorithm for updating the script representation ensures that manual edits are retained during learning. This will be discussed in detail in the Architecture section. Additionally, there is a multi-step undo facility that reverses the effects of the last action on the script modification process.

Partial and complete automation. During playback, the system provides the user with a facility for either complete or partial automation. The user can choose to have the system execute the next suggested step by pressing a "do single step" button. The user can also request automatic completion of the rest of the script at any point in time by pressing a "do all" button.

Highlighting. A system that is designed to guide users through procedures needs to provide as much feedback as possible about *what* the user (or system) is to do next, and *where* that action is to be performed.

Two forms of highlighting are presented during playback. The first is highlighting of the next step to be performed in the script, by coloring the background of that step (see Figure 1a). This gives users a visual cue as to where they are in the procedure execution. The other form is highlighting within the application UI of the widget on which the next step is be performed (see Figure 1b). This form of highlighting is currently displayed as a colored oval around the target widget.

Mixed initiative. An important feature of DocWizards is the mixed initiative model – a facility which allows the user to perform portions of the procedure, while permitting the system to perform other portions of the task automatically, with the decision to retain or yield control fully at the discretion of the user. When the user is performing portions of the task manually, an important feature is an ability of the system to follow along, helping the user to retain context by showing them where their actions are located with respect to the rest of the current procedure.

While playing a procedure, the user is free to perform any steps manually, even steps that diverge from the script. The system continually monitors the user's actions and compares them with the procedure representation. While the user's actions are consistent with the script, the system will follow, suggesting the appropriate next action at each point in time. When inconsistencies are detected, the script will be modified if required (i.e., if recording is turned on), and the system will scan the script looking for portions that might be consistent with the user's actions. When such correspondences are detected, the system resumes prediction of possible next actions.

Generalization. A common feature in programming-by-demonstration system is a facility for generalizing (i.e., variabilizing) entities within a procedure. There are two forms of generalization supported within the DocWizards system. The first is variablization of entities used within loops (discussed in the Architecture section below). The other is generalization of text input and selection operations. The author of a procedure can specify that a particular entity (for example, a string entered into a text input field) is to be generalized. This is accomplished by selecting an individual step within a script, and then invoking a "parameterize step" menu item. On replay, the user will be free to enter any value (or select an entity if the step specifies selection from a list, tree, or table). This is quite useful for applications such as user name/password entry.

We are currently implementing general-purpose variablization of repeated entities, a feature that was supported by our earlier SheepDog system [1].

Full-feature documentation. Since DocWizards is intended to serve as a complete replacement for more traditional forms of documentation and help systems, it includes all the important features of such documentation. These features include textual descriptions, listings of actions to be performed, human-readable control logic, multiple levels of detail, and labeled screen captures.

The script representation and display in DocWizards is designed to allow it to serve as a replacement for more traditional forms of documentation. The procedure scripts are presented as tree structures. Portions of the script (such as the bodies of loops or of if-then-else statements) can be collapsed or expanded by the user to provide different levels of detail. Annotations can be manually inserted into the script, allowing for explanatory comments or section headings. Screen-captures can be automatically created during recording, with highlighting overlaid to illustrate UI action. These screen-captures can be presented on a per-step basis during playback, or can be automatically inserted into an HTML representation of the script, providing a complete document that can be displayed or printed.

ARCHITECTURE

Figure 4 shows the major components of the DocWizards system, and the data-flow relationships between them. In this section, we will describe each of the components and how it contributes to the functionality of the DocWizards system.

State/action instrumentation. To facilitate learning, we employ a model of events and state that we call the *state-action pair model*. The fundamental idea is to capture the *state* of the GUI just prior to each user *action*, on the assumption that the user chooses which action to perform based on the prior actions they have taken and on the visi-

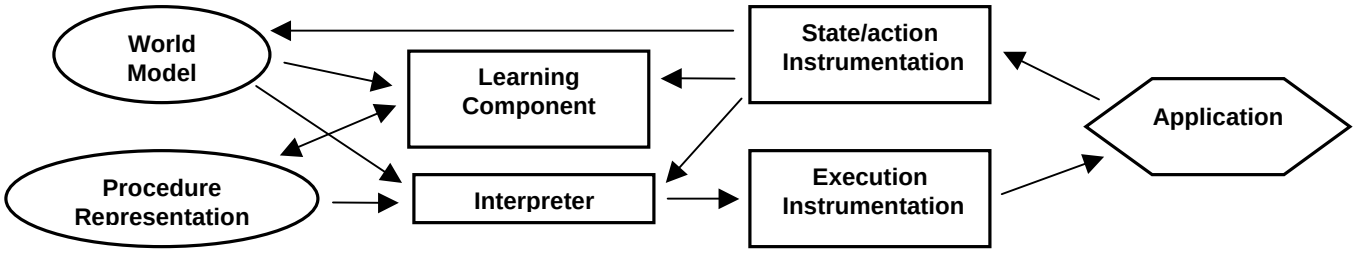


Figure 4. Architecture of the DocWizards system.

ble state of the on-screen interface at the time of the action (see [3] for a more detailed discussion of the state-action pair model). Of course, this is a simplification, but our experience to-date has been that it works remarkably well.

The state/action instrumentation provides information from SWT widgets within Eclipse [2], and is used to build the state-action pair model. The state consists of the hierarchical relationships between widgets and widget contents, including the values within lists, the checked/unchecked state of radio buttons, the text of labels, etc. This state information provides a continually updated view of what is visible on-screen. The actions describe each step taken by the user, for example, clicking a button, or entering a value in a text input field.

State is retained in an internal structure called the *world model* that mirrors the on-screen widget hierarchy. As needed, the system performs a *snapshot*, traversing a portion of the hierarchy, querying each widget for its state, and using that information to update the world model. At the start of recording or playback operations, we snapshot the entire GUI. Thereafter, we update the world model incrementally, by snapshotting the current window at the start of each action. We also snapshot whenever a new window or menu is displayed, by registering callbacks with SWT for display events.

Actions are also received by registering callbacks with SWT for particular events. Most of these notifications are fairly high-level, for example, "list item selection" or "button click". The state-action pair instrumentation provides these user actions to the learning component during recording, and to the interpreter during playback, allowing them to update the procedure structure, and predict next steps, as appropriate.

Learning component. The learning component is responsible for maintaining the script representation during recording. The learner ensures that the script continually remains consistent with all demonstrations of the procedure that have been seen (editing operations may invalidate this rule; we will discuss editing later in this section)

As new actions are received, the action is checked against the previously generated script. The learning algorithm maintains a "current position" pointer; if the newly received action (which we will refer to here as the *new action*) corresponds to the action at that place in the script, the script is consistent and no changes are made. On the other hand, if the action does not correspond to the action in the script (which we will refer to here as the *inconsistent action*), the learning algorithm needs to adjust the script to be consistent.

For example, consider the transformation of the script in Figure 1a into the script of figure 2. At the point that the learning algorithm was expecting the action "Select menu item File->New->Other...", the user actually performed the action: "Expand tree item pserver:anonymous@ cvs.codeshareit.net/cvsroot/ABCMusic". This difference caused the learning algorithm to create an if-then-else structure which is consistent with both demonstrations.

Consistency adjustment is done by creating a set of hypotheses for potential structures that are consistent with all demonstrations. Rather than doing this from scratch, which cannot be done in real-time, the current script structure is used as a starting point. The learning component creates a set of variations of a local region of the script that contains the inconsistent action. Possible hypotheses include creating a conditional branch, creating a loop, or adding the new action to an existing conditional branch or loop. A set of heuristics are used to score each potential hypothesis. Simplicity of structure contributes heavily to this scoring function. The lowest scoring hypothesis is selected, and the newly modified script is displayed.

Loop detection looks only at the actions within a demonstration. We propose a Foreach loop whenever a common set of action is performed for each item within a list, table, or tree, and these items are processed in ascending or descending order. Within the loop body, each instance of the item is automatically replaced by the iteration variable.

In addition to dynamically determining script structure, the learning algorithm must also propose conditions for branches. Branch detection begins by binning analogous actions. The question is then asked, "What features distinguish the state associated with actions in these different bins?" We use an iterative algorithm which employs heuristics to limit the scope of the state that must be examined. We start by looking for absence/existence of widget involved in the action (i.e., one action has the widget available, the other does not). Next we look at features of the widget involved in the action. The scope is then widened to look at all siblings of the widget. Finally all widgets within

the window are considered. At each iteration, a classification tree is employed to produce a "best" explanation of the differences between bins, and to score the explanation. In addition to how well it explains why different actions were taken, the score also considers simplicity of the explanation. Broadening the range of the search reduces the score. Anytime a perfect score is computed, the search is terminated; otherwise the best score from all levels of the search is selected. The classification tree associated with the best score is readily converted into a rule involving features in the state and their values. Examples include "list item X exists", "button Y is unchecked", and "tree item Z is selected".

The predicate for a conditional is a Boolean function of simple features of UI widgets (e.g., entire text string, whether or not selected, etc.). More advanced hypothesis, such as "last item in the list", or item containing the substring ".exe" could readily be accommodated, but will require additional input from the author to select between multiple hypotheses.

Editing operations introduce additional constraints to the learning process. The problem is to ensure that manual editing operations are not undone by automated script modifications. Our answer is to "lock" a script that has been manually edited, making it impervious to future automated changes. In other words, a script that has been edited can have steps added to it (including surrounding control structures), but its internal structure can never be altered. We do this by constraining the hypothesis generation process, so that only hypotheses consistent with such locking are generated. Note this is an extremely conservative approach. We have ensured that manual edits will never be undone, but at the cost of rigidity in the script structure. For example, editing of a possible iteration prior to inferring the loop may result in an inability to ever create the loop structure, even though it might be obvious to a human being. See [4] for a more detailed description of the learning algorithm.

Interpreter. The interpreter executes the script during playback. In addition to the script itself, there are two sources of information required by the interpreter. First is the action performed, either by the user or by system automation, which is received from the state/action instrumentation. This information tells the interpreter whether the last predicted action has been performed or if the user has gone "off track". The second is snapshots of application state at appropriate points in time, received from the world model. The state information is used to make "next step" decisions.

We begin with an "on track" scenario. When an action is received by the state/action instrumentation, it passes the action to the interpreter with a call to an "action received" method. The interpreter recognizes that the action matches the predicted last action, and prepares to advance its prediction (essentially, advancing a program counter). In order to determine the next step, however, a state update is required. The state cannot be properly updated until the effects of the action on application state have completed. We make a simplifying assumption, that the state updates will be complete when no state changes have been received by the state/action instrumentation for a fixed period of time, which we call a *quiescence interval*. This interval is currently 0.5 seconds, during which no SWT update events (e.g., window creates, window deletes) are received. The interpreter starts the timer when an action is received. When the quiescence interval has been satisfied, a state snapshot is generated, and sent to the interpreter's "get next step" method. This method makes a next step decision. If the next step in the script is a simple action (e.g., "click button X"), the interpreter simply returns that step. If it is a conditional, the interpreter evaluates the condition using the current state information. If the step contains variables, these variables are instantiated using a set of stack frames. The interpreter continues to move forward, evaluating logic statements, until a simple action is reached, which is then returned as the next prediction.

If an "off track" action is received, a bit of additional processing is required. We want to generate a best guess as to where the user might be in the script. We are currently using a simple-minded alignment algorithm that looks only at the current action. The script is scanned for an action that matches the one received. If a match is found, that action is proposed as the next step. If no such match is obtained, the previous action is retained as the next step proposal. In other words, the program counter is not moved forward.

Execution instrumentation. The execution instrumentation performs two basic functions. The first is highlighting widgets on-screen. The second is performing actions automatically on the interface.

Highlighting consists of simply querying the widget to be highlighted for its on-screen location, and drawing the appropriate overlay on the graphics context for the containing window. A bit of work is required to maintain the overlay when the window repaints (after being minimized, or occluded, for example). We register (with the Display) for paint events, and re-draw the overlay when they are received. Since the overlay may extend beyond the boundaries of the widget being highlighted (we draw an enclosing rectangle outside the widget bounds), we currently use a conservative approach – doing a re-draw for all paint events regardless of which widget generates the event. Although less computationally intensive schemes (but more difficult to implement) are evident, we have not noticed a performance penalty with this naïve approach.

The bulk of the execution automation is performed using an open source package called Abbot-for-SWT [5]. This package provides a library of automation techniques by widget type, including operations such as "select menu item", "select list item", "click button", etc. The automation library is invoked with a pointer to the desired widget,

and parameters that describe the action (for example, the string of the list item to be selected). The library translates the command into low-level mouse and keyboard events. Since the automation package simulates user input, the task of managing mixed-initiative input is greatly simplified – automated actions look exactly the same as manual actions to our state/action instrumentation.

USER EVALUATION

We conducted an initial evaluation of follow-me documentation wizards as a substitute for traditional documentation. The evaluation had three basic goals:

- Determining how well people operate with DocWizards compared to traditional documentation
- Determining whether people would use DocWizards as an alternative to traditional documentation
- Gaining feedback on how well the playback features within DocWizards work, including highlighting and next-step automation

The evaluation group consisted of researchers and developers at IBM TJ Watson Research Center. The twelve participants had experience using the Eclipse platform for code development ranging from only having tried Eclipse once or twice to several years of extensive use. The participants had individual sessions, each lasting thirty to forty-five minutes.

Participants were asked to do an installation and configuration task by following a printed document. The document contained descriptive text, lists of actions to be performed, and several screen shots showing desired outcomes, or the location of hard-to-find UI elements.

Following a brief training session, participants also performed the same task using the DocWizards system. Participants were paired by experience level, with one member of the pair using the printed documentation first, and the other using the DocWizards system first. The participants were further divided into two groups. One group (six participants) was presented with a version of DocWizards that tracked their actions and suggested a next step, but had no automation features. The other group (six participants) was presented with a version of the system that had a "do single step" button in addition to the automatic tracking feature. Participants in this group were shown both automation and tracking features, and were told that they could perform each step in the procedure themselves, with the system following along, or using the automation feature.

One notable difficulty arose in use of the DocWizards system. Although participants were told to follow the script as closely as possible, a number of them did things differently from the instructions, causing the DocWizards system to run into trouble tracking them. In several cases, we guided the users back "on track". We felt that this invalidated our timings, so we are not reporting comparative statistics. To our surprise, the time for completion for the printed documentation and using DocWizards was roughly equivalent, even when automation was available.

Overall impressions. Participants were generally enthusiastic about the follow-me documentation wizard. In response to a question as to which they preferred, traditional documentation or the DocWizards system, eight users strongly preferred the DocWizards system, three users somewhat preferred it, and a single user somewhat preferred traditional documentation (and one non-respondent).

Features that participants found particularly helpful included the highlighting of widgets within the application, reporting that it helped them to locate the relevant portion of the interface to be operated. Participants also liked the highlighting of lines within the script, reporting that it helped them to keep track of where they were. Participants were more divided on the automated script following. Although most participants liked the fact that the script was being automatically evaluated, several, particularly in the group who had the "do single step" control available, noted that they were not really learning from stepping through the script, and if they had to do it on their own later, would probably be lost. One the other hand, two of the participants stated that they felt that DocWizards was particularly effective in helping them to learn the task. Both stepped through the script manually, and examined the logical structure of the script as they were performing the task.

A number of participants asked why we didn't simply have a "do all" button that would perform the entire script automatically (in fact, the system has such a capability, which we disabled for this evaluation). On the other hand, several participants said that they liked having the full script, including conditional expressions, available for them to evaluate as they performed the procedure. One participant stated that he typically does not trust automated systems of this sort, and that having another layer of automation was undesirable. This was the only user to state such a view, however.

There were several common experiences and opportunities for improvement that we noted during the study:

Confusion. We noticed several participants becoming confused between what the system was suggesting, and what the system was doing for them. In several cases, when the system highlighted a widget, indicating that the next step was to select that widget, the user incorrectly assumed that the highlight indicated that the selection had already been performed. At the time of the study, we were highlighting by drawing a rectangular bounding box. Our conjecture, backed by user comments, was that the rectangular shape was too similar to the rectangular shading that the UI uses to indicate widget selection. Since then, we have begun to use oval-shaped highlighting which seems to lessen this confusion.

Tracking problems. A source of difficulty was the inability of the DocWizards system to track users when they per-

formed sub-goals, such as navigation, using action sequences other than those within the script. For example, one experienced user clicked on a toolbar shortcut to navigate to a different Eclipse perspective, rather than the menu selection sequence that was recorded in the script. Our system is currently unable to recognize these actions as being equivalent, and loses track of the user's position within the script.

Several of the participants also experienced difficulty when unexpected events occurred while performing the task (i.e., events not handled by the script, such as popup error dialogs).

Conditionals: We noticed that a number of participants struggled with evaluating the conditional logic in the printed documentation. These conditionals are based on the state of the UI, for example, "if the list does not contain the entry X, then you will need to check-out project X using the following steps". When using DocWizards, most of the participants simply allowed the system to do the evaluation for them, without checking or even realizing that a decision had been made, and hence had no problem with identity of the conditional elements. A small number checked what the script was doing, either to learn what was going on, or to validate the system. For these users, DocWizards was no more helpful than the printed documentation in evaluating the conditionals, since it provides no visual indicator of the UI entity used to make the decision. Such indicators might be desirable, particularly in tutorial settings.

Context. A number of users stated that DocWizards did not give them sufficient context while performing the procedure. One form of context that was lacking was information about sub-goals and where current actions were located in the larger task structure. A number of participants stated that more comments in the script (or perhaps context-sensitive information) would be very helpful. One participant suggested highlighting the block currently being executed and displaying comments for that block.

Another form of context that was lacking was information about completion of sub-procedures. A common sentiment was that showing expected results, perhaps using screen captures, at key points in the script would be very helpful (note that to simplify the user experience, we removed the screen capture display from the DocWizards system for the purposes of this study).

Summary: Most participants really liked DocWizards. A number of them stated that they would love to have DocWizards for installations or repetitive tasks. As a result of this study, we have a clear sense of issues that need to be addressed, and additional features that need to be implemented to enhance the usability of the system. These are all quite doable, and we have a good indication that an improved DocWizards would be useful to end-users.

RELATED WORK

Today's documentation authoring systems (e.g., Macromedia's RoboHelp [6]) provide the ability to create static documentation using editors such as Microsoft Word or Dreamweaver. These help systems are oriented towards describing the functionality of widgets in an application window or dialog box, and provide little or no support for capturing paths through an application.

Systems such as RWD's Info Pak Simulator [7] create tutorials based on recordings of a user interacting with an application. DocWizards extends this work, by actually executing the resulting procedure rather than running in a simulated environment (in a browser window), and by supporting model updates from addition demonstrations.

DocWizards' technique of procedure authoring through the use of demonstrations follows in the traditional of classical programming by demonstration (PBD) systems [8, 9, 10]. In earlier work we developed a batch learning approach that generates a procedure from a set of demonstrations [1]. DocWizards, by contrast, is an incremental learner, so demonstrated actions may cause immediate updates to the learned procedure. Further, unlike the earlier system which was more like a wizard, DocWizards generates a fully transparent (i.e., human-readable) procedure. DocWizards also extends the learning capabilities of earlier PBD systems such as Tinker [11] by automatically inferring the predicates for conditional statements.

Considerable support for the DocWizards approach can be found in the intelligent tutoring systems (ITS) literature. Work such as Electronic Performance Support Systems [12] focused on the reduction in documentation required when documentation is provided in context, either through scenarios or by limiting the functionality available to novice users. The documentation provided by DocWizards is similar in spirit, since it provides information contextualized by the current step in the procedure. Palmiter and Elkerson discovered that animated demonstrations with users in a passive role were less effective than text-only explanations for long-term learning [13]. DocWizards combines in-context text-based help documentation with user-controlled animation. We plan to test whether this combination provides a "best of both worlds" kind of performance.

The notion of tracking a user's performance on a task against a known model is one that has been extensively studied in Intelligent Tutoring Systems [14]. Most intelligent tutoring systems are programmed either conventionally, or by encoding domain knowledge for a problem solver [14]. We note that there are examples of ITS systems (see [13]) that use learning to acquire examples. DocWizards extends prior art by learning the underlying model from demonstrations, and presenting it in a human-editable form.

Because of the difficulty of programming these models, some work has been done in ITS to facilitate the authoring

process [15]. The WITS system [16] for example, relies on demonstrations in order to "program" the underlying model. In order to generalize beyond a specific demonstration, it allows the author to manually generalize the recorded procedure. DocWizards uses multiple demonstrations to automatically learn these generalizations. Selker's COACH system [17] provides in-context guidance based on observing user behavior, using examples acquired by demonstration. DocWizards differs in using the demonstrations to learn a task model (which is pre-coded in subject frames in COACH).

CONCLUSIONS AND FUTURE WORK

We have presented a new form of procedural knowledge capture, follow-me documentation wizards, which provide many of the benefits of traditional documentation as well as traditional wizards or scripts. Follow-me documentation wizards provide a complete textual description of the procedure including all control logic and optional screen-captures. They also provide live guidance by walking the user through the script representation while highlighting the associated application controls, and provide partial or complete UI automation.

We have also described DocWizards, a follow-me documentation wizard system in which the procedures are authored through demonstration. DocWizards supports incremental development of procedures through multiple demonstrations as well as manual editing.

An initial evaluation of DocWizards showed strong enthusiasm for this form of procedure documentation, compared to traditional documentation. We also identified several opportunities for improvement, including providing more context within the scripts and better support for synonymous action sequences.

One feature that DocWizards is clearly lacking is a means for the author to specify where user inputs are required. One of the strengths of traditional wizards is that they aggregate all of the required (and optional) user inputs. We will be developing light-weight techniques for an author to indicate that particular UI elements require input from the user, with the option of either requesting these directly on the application UI, or in separate generated dialogs.

Another feature that we are beginning to explore is modularization of follow-me documentation wizards. Procedures often contain subtasks that are used in a variety of contexts. A facility for culling out and parameterizing those subtasks would allow authors to build up reusable task libraries. Using the same form of action alignment that is used to track user actions, we believe we can automatically identify small candidate sets of subtasks, thereby facilitating the problem of searching a task library.

REFERENCES

1. Tessa Lau, Lawrence Bergman, Vittorio Castelli, Daniel Oblinger, Sheepdog: Learning Procedures for Technical Support. In *Proceedings of IUI 2004*, Madeira, Portugal, January 2004, pp. 109-116,
2. http://www.eclipse.org
3. Castelli, V, Bergman, L, Lau, T., and Oblinger, D., Layering advanced UI functionalities on existing applications, IBM Technical Report RC23583, 2005.
4. Daniel Oblinger, D., Castelli, V., Bergman, L. and Lau, T. Similarity-Based Alignment and Generalization. To appear in *Proceedings of ECML* 2005.
5. http://sourceforge.net/projects/abbot/
6. http://www.macromedia.com/software/robohelp/
7. http://www.rwd.com/products_services/enterprise_learning_solutions/products/infopak_simulator/
8. Allen Cypher, ed. *Watch What I Do: Programming by Demonstration.* (1993). MIT Press, Cambridge, MA.
9. Henry Lieberman, ed. *Your Wish is My Command: Programming by Example.* (2001). Morgan Kaufmann.
10. Safonov, A., Konstan, J.A., and Carlis, J.V., Beyond Hard-to-Reach Pages: Interactive, Parametric Web Macros, In *Proceedings of HFWeb 2001*.
11. Henry Lieberman, Tinker: A programming by Demonstration System for Beginning Programmers., In *Watch What I Do: Programming by Demonstration.* (1993). MIT Press, Cambridge, MA.
12. Carroll, J.M. and Kay, D.S. (1988). Prompting, feedback and error correction in the design of the scenario machine. *International Journal of Man-Machine Studies*, 28:11-27.
13. Palmiter, S. & Elkerton, J. (1991). An evaluation of animated demonstrations for learning computer-based tasks, In S.P. Robertson, G.M. Olson, & J.S. Olson (Eds.), *Human Factors in Computing Systems: CHI'91 Conference Proceedings.* NY: ACM, pp. 257-263.
14. Anderson, J. R., Boyle, C. F., Farrell, R., & Reiser, B. J. (1987). Cognitive principles in the design of computer tutors. In P. Morris (Ed.), *Modeling Cognition*, Wiley.
15. Anderson, J.R., & Pelletier, R. (1991). A development system for model-tracing tutors. In *Proceedings of the International Conference of the Learning Sciences*, 1-8.
16. Farrell, R. and Lefkowitz, L. Supporting Development of On-line Task Guidance for Software System Users. In *Facilitating the Development and Use of Interactive Learning Environments*, C. P. Bloom & R.B. Loftin, (Eds.), 1998.
17. Ted Selker, Coach: A Teaching Agent that Learns. In *Communications of the ACM*, July, 1994. Vol. 37, No. 7, pp. 92-99.