**RC23583 (W0504-045) April 11, 2005**
**Computer Science**

# IBM Research Report

## Layering Advanced User Interface Functionalities onto Existing Applications

**Vittorio Castelli, Lawrence D. Bergman, Tessa A. Lau, Daniel Oblinger**
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

**Research Division**
**Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich**

# Layering Advanced User Interface Functionalities Onto Existing Applications

*Vittorio Castelli, Lawrence D. Bergman Tessa A. Lau, Daniel Oblinger*
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598 USA
Phone: (914) 945-2396
[vittorio, bergmanl, tlau, oblio]@us.ibm.com

**ABSTRACT**
We explore the topics of user-interface instrumentation functionality and design in support of advanced features such as automation, adaptive documentation, intelligent tutoring, automated testing, and programming-by-demonstration. We propose a description of the interaction between user and computer applications in terms of a state-action model, we show how it is sufficient to support advanced UI functionalities, and we analyze the main points and trade-offs in the design of an instrumentation layer based on the state-action model. We discuss how the design principles could be practically applied using as supporting examples two systems, Sheepdog and DocWizard, developed on different platforms. We finally suggest a set of guidelines, intented for designers of application platforms such as operating systems, that facilitate the task of layering advanced UI functionalities on top of existing applications.

**ACM Classification** H5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

**General Terms** Design, Algorithms, Measurement, Documentation.

**KEYWORDS:** Advanced UI features, instrumentation, automated testing, programming-by-demonstration.

**INTRODUCTION**
Intelligent tutors [9], adaptive documentation [5], automated GUI testers [1], and programming-by-demonstration (PBD) systems [3, 8] are examples of a class of systems that provide advanced UI functionality on top of existing applications. We shall refer to them using the AUIF acronym (for Advanced User-Interface Functionality).

These functionalities are often built into the applications they enhance and therefore are not easily ported to other applications. Moreover, extending an existing AUIF system to simultaneously interact with multiple applications is a arduous task. One of the main difficulties is the fact that different applications expose different APIs for obtaining data, for performing specific actions, and for obtaining event notifications. Porting AUIF system therefore requires redesigning the interface with the underlying application, which is a costly and time-consuming proposition.

In this paper, we propose a unifying framework for interfacing AUIF systems with underlying applications based on a general model of user/application interaction. Figure 1 illustrates the main components of the framework: an instrumentation layer and a state-action model of the user/application interaction. The instrumentation deals with the specifics of interaction with the application, by obtaining data and event notification, and executing actions. The state-action pair model combines the information produced by the instrumentation into a standardized (i.e., application-independent) description of the application UI and of the user/application interactions. This architecture therefore enables layering multiple AUIF systems onto an application and facilitates porting an AUIF system to different applications.

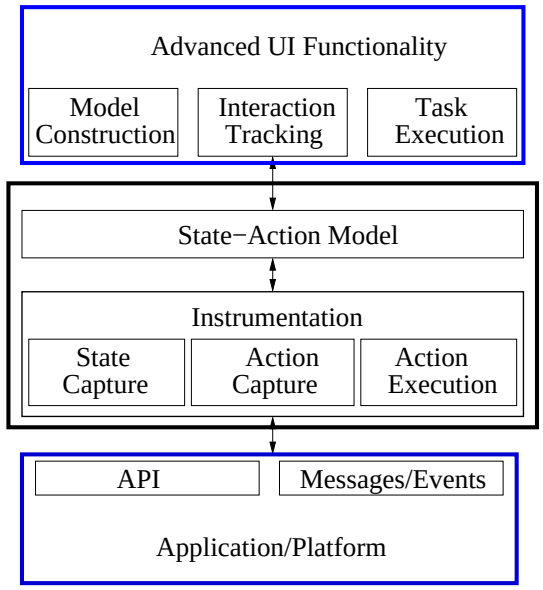


Figure 1: Interfacing an advanced-UI-functionality system with an application using the state-action model.

In order for our approach to be successful, two requirements must be satisfied. First, the general model of user/application interaction must be sufficiently rich to support the main functionalities of the systems of interest. Second, the instrumentation must support the resulting requirements on the state-action model.

We identify a set of functionalities for which our proposed framework must provide support in order to serve as a general interface for all the AUIF systems mentioned above: task-model induction or construction, matching a sequence of user/application interactions to a task model, and automatic execution of operations on a GUI. Note that, while not every system implements all three functionalities, our framework needs to accommodate all of them in order to be usable by a broad variety of systems.

To support these functionalities the instrumentation must implement three classes of operations. The first is retrieving the application GUI content. The second is observing the actions performed on an application by a user executing a procedure. The third is automatically executing actions on the user interface. In simple terms, an instrumentation layer must *see what the user sees*, *see what the user does*, and *do as the user does*. More specifically, the instrumentation must produce a consistent, broad, and detailed representation of the content of the screen; precisely capture the user operations on the application interface; and automate the execution of individual actions on behalf of the user. We devote separate sections to the corresponding components of a UI instrumentation: a data-capture layer, which extracts the content of widgets in the application UI; an action-capture layer that observes user action and, in conjunction with an abstraction layer, combines the captured data into higher-level information; and an execution layer, which interacts with the application to emulate user actions.

In this paper, we make the following contributions:

- We propose a formal description of user interaction with computer applications using a state-action model;
- We describe a framework based on the state-action model for layering AUIF systems onto existing application;
- We discuss the instrumentation required to support a state-action model, discuss alternative approaches and trade-offs, and identify difficulties and limitations;
- We present two implemented systems that make use of this model on the Windows® and Eclipse[2] platforms;
- We propose a set of requirements for platform designers to make it easier to develop intelligent interfaces.

The rest of the paper describes the issues that arise in designing the proposed UI instrumentation layer, and is organized as follows. After a brief notational digression, we introduce a framework for describing the interactions between a user and an application, the *state-action model*. We then analyze the requirements that the state-action pair model must satisfy to support task induction, action execution, and interaction tracking. We describe the main questions that arise when designing a system that sees what the user sees, sees what the user does, and does as the user does. We describe our experience in instrumenting two different platforms for programming-by-demonstation and adaptive documentation applications, and conclude the paper with suggestions for platform designers.

## NOTATION

We use the term *application* to refer to the program or set of programs with which the user interacts during the execution of a procedure; we use the term *system* to denote a system that provides advanced UI functionalities on top of an application. We also use the term *platform* to denote the environment in which the application operates (e.g., Microsoft Windows® or the Eclipse platform).

Note that both individual applications and whole platforms can support advanced UI functionalities. In the rest of the paper, we do not explicitly distinguish between interfacing with applications and with platforms: our analysis and conclusions will hold in both cases.

## THE STATE-ACTION MODEL OF INTERACTION

In this section we describe our model of user/application interaction, the state-action model. We identify the main requirements of task model induction, automatic task execution, and execution tracking. We then analyze how these requirements reflect onto constraints on the state-action model. Since PBD systems require induction, automatic execution, and tracking, we will use them as a running example.

Most existing PBD systems use only information about user actions to capture the target procedure. For example, Familiar [11] captures user actions in the Macintosh Finder application, and recognizes repetitive sequences of such actions. The main limitation of this approach is the difficulty in inferring constructs, such as decision points, that depend on the content of the UI. A few systems, like SMARTedit [6], instead capture states of an application, and build a procedure model by identifying the actions that are consistent with the observed state changes. The main limitations of these systems are the assumption that different actions produce different state changes, and the need for a very precise model of the target application (in fact, of the specific release of the target application): in order to infer the procedure model, the system must know exactly what actions are required to yield the desired change in state.

In contrast, we propose to represent the model of user interaction by using a conversational turn-taking paradigm that supersedes both approaches and overcomes their limitations. At the beginning of each turn, the system resides in a particular state; the user chooses an action to perform, such as clicking a button or selecting a menu item. In the next turn, the system updates its internal state in response to the user's action and reflects the change in the user interface, thus completing the turn. A user interaction consists of a sequence of such turns. The rationale for this model is the assumption that the user performs an action based on a mental model of the procedure, on the current location within the procedure, and on the content of the screen. The application receiving the action executes it, updates its internal (invisible) state, and provides feedback to the user by changing the content of the screen. Using a control theory analogy, the content of the screen is the observable state of the application and, if the UI is well designed, exposes all the information that the user

needs to decide which action to perform next.

Using both state and action information overcomes the above-described limitations of using just one. State information can influence a user's decisions while interacting with an interface. For example, a user may choose to perform different actions depending on what is visible on the application interface, or, equivalently, on features of the state: adding an entry to a list only if it is not present or installing a software package only if it is not already installed. Action information is required for application independence: without it a system layered on top of an existing application needs a specific model of the application to deduce user actions by comparing the states obtained before and after each interaction. Action information is therefore needed to support an architecture that decouples the AUIF system from the target application.

The resulting state-action model is somewhat similar to the STRIPS representation used in planning [4], where the state captures facts that are true in the world, and plan operators describe the possible means for altering the state of the world.

We next analyze the main components of model induction, action execution, and interaction tracking, and we show how they impose requirements on the state-action model and for the instrumentation.

## SUPPORTING ADVANCED UI FUNCTIONALITIES

The interface framework must provide the information and functions required by automatic task-model construction, task execution, and interaction tracking. In this section, we summarize the most significant ones.

*Uniquely Identifying Widgets.* The visual appearance of an application, in particular its layout, typically varies depending on user settings and environment differences. The same widget might be in different positions or have a different look. Consider, for example, the same toolbar button in two instances of the same application: it can be in different positions within the toolbar, or can even be absent; the toolbar can look different, can contain different buttons, or can even be absent; the toolbar can be in a different position relative to other toolbars or to the application frame, it can be vertical rather than horizontal, etc.

Consistently identifying the same widget across different instances of the same application is important for automatic task-model construction and for automatic execution. In task-model induction, the content of widgets is used to induce decision points. Hence it is important to identify the same widget in execution examples obtained from different application instances. Identical actions on the same widget must also be recognized as being identical irrespective of the interface configuration. In automatic execution, the execution engine must correctly find the widget to be activated irrespective of its position and look.

Similarly, different widgets should not be identified as one.

*Activating Widgets.* Automatic execution task execution on the UI requires the ability to activate UI widgets.

*Characterizing Widgets.* Task model induction requires a precise characterization of widgets. This characterization includes: the functionality of the widget, which is typically jointly determined by its type and the value of a collection of attributes; the knowledge of whether a widget state can change or is always fixed; the widget content, such as its text or, for container classes like lists, the contained widgets; the state of the widget (active, grayed, checked, expanded, collapsed, etc.). An accurate widget characterization additionally includes spatial and hierarchical relations between widgets. An example of spatial relation is the proximity of a type-in field to its "label", when the label is actually a different widget. As an example of hierarchical relationship, think of the tree representation of a directory structure in the "file open" dialogs. An accurate characterization of widgets is also essential to widget identification.

*Associating states with actions.* This is important in automatic task induction: each action should be associated with the state of the interface at the time when it started. Using this information, the inference engine tries to induce structural constructs of the task, such as loops and branches. State-action association is also required for task execution: for example, an action should not be executed until the target widget is available and ready (think of the case in which the widget is created as a consequence of the previous action), and this information is typically available in the state. Similarly, a decision point should not be evaluated during execution until the previous action has completed, an event that can typically be deduced from the state.

## REQUIREMENTS OF THE STATE-ACTION MODEL

In order to provide the information and functionalities described in the previous section, one must impose a set of requirements on the state and action representations and on the action execution.

### Requirements of State Representation

*Widget-Type Coverage.* The state representation should contain information on as many different widget types as possible. Our interface layer cannot make assumptions about what types of widgets are important, for example, to a task-model-induction algorithm or to an automated GUI testing system.

*Richness of Detail.* A rich description of the properties of each widget is highly desirable in a variety of scenarios. For example, a task-model-induction algorithm uses state information to explain user behavior, and therefore needs to accurately know the properties of the widgets that can influence the user decisions. Other properties, like the exact location of a widget, are important to automatic task-execution engines that emulate mouse and keyboard events, and to intelligent tutorial systems that highlight portions of the UI for the benefit of the user. Finally, richness of detail is required to correctly identifying widgets. For this purpose, structural information about the widget hierarchy is also often required.

*Screen Coverage.* The state representation should broadly cover the content of the application UI. In particular, all widgets that are visible (i.e., not occluded or minimized) can be both targets of execution and sources of information for automatic task-model induction. Although not visible, occluded widgets could be targets of execution.

*Dynamic Updates.* Changes in the UI should be quickly reflected in the state representation. This requirement is important, for example, to automatic execution, where an action should not be performed before the effects of the previous one are visible on the screen.

*Synchronization with Actions.* The state-action model of user/application interaction is predicated on the assumption that the user makes decisions based on a mental model of the procedure, the current "place" within the procedure, and what is visible on the UI just before the action is performed. It is therefore imperative that the state representation satisfy this synchronization assumption.

#### Requirements of Action Representation

*Action-type Coverage.* Representations should be available for all types of user actions, for every type of widget.

*Richness of Detail.* The representation of an action should be sufficiently detailed to allow automatic discrimination between similar actions. It should also contain sufficient information to uniquely identify the widget or widgets involved in the action, as required by automatic task execution and model induction. Finally, there should be enough details to instruct the execution engine as to how to execute the action.

*Semantic-level Representation.* Actions should be represented at the same level of abstraction used by a human to describe the task. This substantially simplifies the induction of a task model and the process of matching a sequence of interactions to a model, by retaining the essence of what the user does while hiding the low-level details of how it is done. For example, many actions can be equivalently performed with the mouse or with hot keys: considering these low-level details introduces a source of variability that makes induction and tracking more difficult without adding any benefit to either.

*Uniformity of Representation.* The descriptions of different types of actions must be at the same or at a close semantic level.

#### Requirements of Action Execution

The requirements of the action execution layer are diverse and contrasting, and strictly depend on the goal of exeuction. We can distinguish three classes of goals: 1. performing a procedure on behalf of the user, were no further restriction on the action execution mechanism is imposed; 2. performing a procedure in cooperation with the user, where the user can take over at any point in time, namely, in "mixed initiative"; and 3. emulating the user interaction with the application.

*Visibility.* Intelligent tutoring systems and adaptive documentation systems that can demonstrate to the user how to interact with an application need support to execute action in a way that is visible to the user. A somewhat more stringent requirement is imposed by automatic GUI testing systems, which actually need actions to be performed by means of mouse and keyboard emulation. To execute a procedure in mixed-initiative mode, the user must see the system actions. Therefore, visibility is a requirement for goals number 2 and 3.

*Uninterruptibility of individual actions.* Systems that completely automate task execution for the user should be impervious to accidental user intervention (such as moving the mouse). A similar requirement holds for intelligent tutoring systems and adaptive documentation systems that demonstrate multi-step interactions with applications. The only allowed violation of this requirement occurs when, by design, the user is allowed, in mixed initiative mode, to interrupt the system execution at any point in time.

*Interruptibility of action stream.* To operate a mixed initiative mode, where the user can take over the execution at any point in time, the action execution component should allow the user to interrupt the execution flow between actions. In other scenarios, such as intelligent tutoring, the execution should probably not be interruptible.

*Speed.* For purely automatic task execution, the speed of execution of long procedures can contribute to enhancing the user experience.

### STATE CAPTURE

The purpose of the state-capture component is to "see what the user sees". It must satisfy five main requirements (widget-type coverage, richness of detail, screen coverage, dynamic updates, and synchronization with actions) without negatively impacting the user experience. The first five requisites can result in a complex and computationally heavy state-capture layer.

To capture the state of a widget, the instrumentation relies on exposed APIs that range from methods on a public interface of the widget (such as the methods on the `IAccessible` interface in Windows®, or the `GetChecked()` method on an Eclipse check box), to messages (like the `WM_GETTEXT` message that retrieves the text of a window in Windows®). Widget-type coverage and richness of details are the result of an extensive state-capture layer.

The last requisite, for a good user-experience, limits the applicability of simple-minded, brute-force strategies (that yield good screen coverage, dynamic updates, and synchronization with user actions) to applications with simple UIs (e.g., command-line-based programs). Two examples of these strategies are: intercepting each user action and capturing the entire state of the UI before allowing the action to be received by the application; and sampling the entire content of the UI at a high enough sampling rate (e.g., every $1/20$ of a second). Although, when feasible, both strategies satisfy most of the requirements, they are bound to severely impact the user experience on the vast majority of applications.

Therefore, to ensure a high-quality user-experience the instrumentation must adopt strategies to adaptively monitor parts of the UI, and to judiciously capture selected content. The decisions of what to capture and when to capture must try to satisfy all the desired requirements.

**What to Capture**

The advantage of capturing the entire GUI is that no information is disregarded: this yields complete screen coverage and richness of detail. The disadvantages are the computational burden, which can be substantial and therefore affect the user experience, and the presence of information that is irrelevant to the task. In particular, if the model of the procedure is inferred automatically, via machine learning techniques, the well-known problem of the curse-of-dimensionality comes into play, and statistical variations in the appearance of the UI can make irrelevant information appear to be highly predictive of user decisions. The advantages of capturing restricted parts of the GUI are the smaller computational burden and the fact that many (but not all) irrelevant features are automatically discarded. The disadvantage is that the relevant features could be discarded too. To implement partial capture, the instrumentation must adopt a strategy for adaptively selecting which windows to monitor. To further improve the user experience, the system could use different strategies at procedure-demonstration time and at procedure-playback time.

At demonstration time, the instrumentation must trade off computational cost for risk of discarding relevant data. The most conservative strategy, short of recording the state of the entire UI, consists of including all the windows that are not minimized, including those partially occluded by other windows. The idea is that the user cannot make decision based on information that is not visible. We use this approach in both our systems described in later sections. At the opposite end of the spectrum we find a strategy consisting of capturing only the widget being interacted with (a strategy that we do not advocate, as the risk of missing relevant information is high). A somewhat safer approach consists of considering the windows containing the widgets where the current and previous actions are performed.

During playback time, when a procedure model exists which is either executed automatically or used to track user actions, the system knows which widget content is relevant at each point in time, for example to determine which path of a branch needs to be taken. The system could therefore request from the instrumentation only the content of these specific widgets. This selective approach to state capture typically requires the retrieval of a small amount of information in response to each request, and therefore it has a small impact on the user experience.

**When to Capture**

Even when it captures the entire GUI, the instrumentation does not need to obtain the content of all tracked windows at the same point in time. A strategy that substantially improves the user experience is to maintain an internal representation of the content of the UI, which can be thought of as a cache of the visible widget states, and to update it incrementally, by appropriately selecting the portions of the UI to be queried. We use the term "World Model" to denote this cached representation.

We distinguish two main approaches to deciding when to capture the content of selected portions of the UI and update the corresponding parts of the World Model, respectively called "widget-notifies" and "instrumentation-requests". In both cases, the instrumentation registers callback functions with specific widgets or in a system-wide fashion. Different types of callback functions are invoked when different aspects of the state of a widget changes and when user actions are detected. The notification mechanisms are different depending on the platform. For example, Windows® allows a user to register a callback for Accessibility events which is invoked when a relevant event is generated. Example of events that trigger an accessibility callback are: changes in object text, state, and location, the creation and destruction of an object, changes in selection within a container object, etc. The forms of the callback functions also differ depending on the environment. In Java, each class of events is associated to a specific "listener" interface (derived from `EventListener`) that specifies one method per each event of the class: the PopupMenuListener has three methods invoked respectively when a popup menu become visible, when it becomes invisible, and when it is canceled. In Windows® callbacks are just regular functions with specific signatures.

*The widget-notify paradigm* In a widget-notifies technique, instrumentation is *reactive*: whenever a callback function is invoked, the instrumentation retrieves the offending widget state by sending it appropriate inquiries. In this approach the World Model is updated one widget at a time.

Provided that the platform supports callback functions that are both exhaustive (that is, the instrumentation can be notified of all desired changes for all types of widgets) and consistent (that is, notifications are both deterministic and happen under all circumstances), and provided that the widgets respond to the inquiries under all circumstances, this approach satisfies the dynamic updates and synchronization with actions requirements. In general, however, a designer cannot assume that the notification mechanism is well-behaved in the sense just described, and should carefully verify that these conditions hold. Even if the conditions hold, the downside of the approach is the potentially high computational cost: typically, the stream of notifications is substantial and most captures performed by the instrumentation are unnecessary.

*The instrumentation-request paradigm* In a technique of this class the instrumentation is *proactive*. The instrumentation observes a stream of events from the application widgets, exactly as in the widget-notify approach. Instead of performing a capture whenever an event notification is received, the instrumentation selectively decides when to capture part of the content of the application UI. The retrieved information can include content of widgets other than those generating the observed events. A variety of strategies can be used to implement an instrumentation-request approach. The most simplistic one consists of performing the capture at predetermined, equally spaced intervals, irrespective of the received

messages. This approach is simple to implement, but results in unnecessary state captures, and is either computationally expensive (if the sampling frequency is too high) or has a high risk of missing relevant information (if the sampling frequency is too low). It is therefore limited to applications with a simple or small UI (e.g., text), for which event-notification support is poor.

A more reasonable strategy consists of capturing the GUI when the user performs an action, by preventing the action from being executed until the state-capture is completed. As discussed previously, this strategy yields a poor user experience unless the state capture is targeted to a limited number of widgets. For example, the instrumentation could keep track of all the messages indicating widget-state changes during the interval between adjacent user actions, and retrieve only the content of the affected widgets. This approach guarantees that all the changes in the content of the UI are correctly recorded, but it does not ensure that the number of widgets to be queried is sufficiently small to guarantee a good user experience.

A third approach consists of performing an initial capture when the system is activated, when new windows or widgets are created, and after the results of a user action are visible on the UI. The last event is difficult to assess unless the instrumentation has an accurate model of the application, or if the application has a notification mechanism for completed actions (as is the case, for example, for a database). A heuristic that approximates the detection of the completion of an action consists of waiting for "quiescence": the instrumentation would observe notification of changes to the GUI after the user performs an action, and performs a capture when no change is observed for a heuristically selected period of time.

In general, the instrumentation could adopt a hybrid between the second and third strategy: it could capture the GUI when predetermined events occur (e.g., startup and window creations); when specific sequences of event notifications are detected (e.g., the sequence window-deiconify, window-focus, window-repaint); when a large number of widget-state changes is detected; and when a user performs an action. These hybrid approaches have the advantages of limiting the interference with the user experience (by incrementally acquiring small amounts of data and ensuring that the cost of state-capture when an action is detected is small), of keeping the computational cost under control, and of yielding a consistent model of the UI content. The downside is the complexity of the design. In particular, the types of events that trigger the state capture are in general widget-dependent (and different parts of the state can be available when different notifications are issued), and a careful design is required to match notifications and available aspects of the state.

## ACTION CAPTURE

The second main function of the instrumentation is "seeing what the user does". Recall the desiderata of the action representation: type coverage, richness of detail, semantic-level representation, and uniformity of representation. The requirement on the richness of detail, in particular the availability of a low-level step-by-step description, is in contrast with the need for a high-semantic-level representation. The solution is to distribute the burden between the action-capture and the action-execution module: the action-capture module is responsible for generating the high-semantic-level representation, while the action execution module is responsible for decomposing a high-level action into a sequence of low-level interactions with the application.

An action-capture architecture that yields both high-level and uniformity of representation consists of an data-gathering component, which retrieves action information, and an abstraction component, which combines the gathered data into high-level, coarse-grained description. In this section, we first describe the data-gathering component, then we describe the abstraction subsystem and discuss the trades in the common design of these two elements.

### The Action-Data Gathering Component

The basic mechanism of action-data gathering consists of registering appropriate callback functions with individual widgets or in a system-wide fashion. These callback functions are invoked upon the occurrence of events caused by the interaction of the user with the UI. Event notifications and callback functions are conceptually identical to those discussed in the data-capture section.

The information obtainable from the action-data-gathering component can be categorized in terms of cause and effect. Certain callback functions are invoked when an action performed by the user is detected, for example, when the user clicks the left mouse button. Other callback functions are invoked when the direct effects of the user action are observed, for example, when the system menu pops up in response to the left mouse button click. It is important to note that many state-change events are not direct consequences of user actions, but are rather due to reactions of the application. For example, consider an e-mail program with an inbox and a preview window: when the user selects a message in the inbox, the content of the message appears in the preview window. Say that two events are generated: the first signals that the selection in the inbox list has changed, the second signals the change in content of the preview window. The first event is a direct consequence of the user action, and should be captured by the action-capture layer; the second is an indirect consequence, and should not be captured by the action capture layer.

In practice, callback functions that detect user actions typically yield low-semantic-level information, such as mouse moves and clicks, and key presses. In contrast callback functions that detect the effects of user actions typically yield information at a higher semantic level. For example, depending on the characteristic of the widget and on the mouse pointer location within the widget, a mouse click could result in a button press, a checkbox check, a tree-item expansion, a window minimization, etc.

In an ideal application platform, callback functions would exist that yield information at the same semantic level that an expert would use to describe interactions with the GUI. This, unfortunately, is usually not the case: for example, double-clicking on a desktop icon can result in launching an application or opening a document, depending on the nature of the

icon; the existing notifications do not provide sufficient information to discriminate between the two cases. Hence, the available callbacks that provide information at this desirable high semantic level collectively do not satisfy the coverage requirement, and therefore the action-data gathering cannot by itself satisfy coverage, high-semantic level, and uniformity of representation. These requirements, however, can be met by means of an abstraction layer.

### The Abstraction Layer

The purpose of the abstraction layer is to combine the information retrieved by the data-gathering component into a comprehensive, uniform high-level descriptions of the user actions. The designer must distribute the burden across abstraction layer and data-gathering instrumentation.

At one end of the spectrum, we find a solution consisting of collecting only the lowest-level user actions, and making the abstraction component responsible for most of the work. This approach has two advantages: the data-gathering instrumentation is very simple (one can construct a full instrumentation by registering just the callback functions for the following four events: mouse-down, mouse-up, keyboard key down, keyboard key up), and it is uniform across applications and widget types. The downside is the need for a very complex abstraction mechanism to produce the desired high-level description of the task from a sequence of mouse clicks and key presses.

As an example of the complexity involved in this approach, consider the simple task of detecting double clicks from mouse-down/mouse-up events (abstracting more complex mouse actions, such as hover and drag-and-drop, is substantially more involved). The abstraction mechanism must consider a variety of information: the event time stamps, the mouse location, the system settings for the double click speed and maximum mouse movement, and the type of the widget receiving the action. If the widget reacts to a double click, and if the individual clicks were sufficiently close in space and in time, the abstraction layer declares a double click and proceeds to further analyze the action. Here, a detailed description of the widget and platform-specific information on how different widgets operate are necessary: for example, in Windows® double clicking a desktop icon results in launching an application or opening a document, while double-clicking the titlebar of a window maximizes the window or restores its pre-maximization size.

At the opposite end of the spectrum, we find an approach where the instrumentation gathers information at the highest possible level of abstraction available, which, as mentioned, varies across widgets and types of user actions, and use the abstraction layer to "equalize" the results. The lack of uniform coverage and of uniform interfaces provided by the available callback functions can reduce the effectiveness of this strategy. In the worst-case scenario one needs to separately deal with each type of widget and each type of user action, thus making the instrumentation potentially complex and difficult to maintain.

Based on this discussion and on our practical experience, we can reach some broad conclusions. An approach that favors simplicity of the data-capture layer is better suited for very diverse environments, such as the Windows® operating system. In contrast, an approach that favors capturing actions at a high-level of abstraction is very effective for environments with a restricted, well-defined widget set, such as the Eclipse platform.

## ACTION EXECUTION

The third main function of the instrumentation is to "do as the user does". Recall that automatic execution is somewhat different from the other two functionalities, since its actual mode of operation strictly depend on the goal of the execution. We identify three such goals: performing a procedure on behalf of the user, executing a procedure in mixed-initiative, and emulating the user interaction with the application. Recall that we also identified visibility, interruptibility of individual actions, interruptibility of action streams, and speed as additional requirements.

Broadly speaking, there are three approaches to automatic interaction with an application: via an API that exposes the internal operations of the application, via an API that exposes the UI functionalities, and via mouse/keyboard emulation.

Using an API that operates on the internal data structure of an application or a platform (e.g., programmatically modifying the registry) is not suitable for most types of system considered in this paper, and therefore we will not elaborate on it.

Interacting with an application by calling appropriate methods of the widget, or, equivalently, by sending appropriate messages to the widgets, is an approach with a variety of benefits. In particular, the execution is typically faster than with mouse/keyboard emulation and is not susceptible to errors due to human interference. The main downsides are complexity and size of the instrumentation, which requires specific methods to interact with specific widget. Additionally, this approach is entirely unsuitable for emulating the user interaction, for instance for automated testing, and poorly suited for tutoring purposes.

Mouse-and-keyboard emulation is a fairly general purpose approach to action execution. Its main benefits are the simplicity of the action execution layer, and the ability to show the user how to perform the task. A downside is the somewhat slower execution, especially of "actions" that require multiple elementary interactions, like selecting an item in a combo-box. Also, this method is vulnerable to a user's destructive interference, which can be caused by accidentally moving the mouse or clicking a key. A potential solution for the last problem consist of "completely taking over the machine", namely, of temporarily disabling mouse and keyboard while each action is executed. More complex solutions involving retries are also possible.

## EXAMPLES

We now illustrate the principles described in the previous sections in the context of two actual systems, Sheepdog and DocWizard.

### Sheepdog

The Sheepdog system [7] is a Microsoft® Windows® 2000-based application that learns technical support procedures from multiple expert demonstrations. Sheepdog's inference engine combines sequences of state-action pairs (called *execution traces* or simply traces) gathered by its instrumentation layer into a probabilistic model of the procedure. This model belongs to a special class of Hidden Markov Models, and is induced, roughly speaking, by finding the best possible alignment of state-action pairs from different traces based on both similarity and the usefulness to predict the next action (more details can be found in [10]). The inference algorithm relies on a collection of statistical classifiers that take as inputs the state and predict the action, and on a distance function that measures the similarity between the state-action pairs.

Sheepdog is intended as an application-independent (in fact, cross-application) system that can perform technical support tasks in cooperation with the user, namely, in mixed initiative.

The state-capture component is intended to work with any Windows-based application. As a consequence, a fundamental design point was to select uniform strategies for deciding when to capture the UI content and what to capture in the UI content.

To understand the design decisions of the selected architecture we need to briefly review how Windows® supports instrumentation. To capture the content of windows, Windows® offers two main mechanisms: inquiry messages and Active Accessibility® functions. A user can send a specific inquiry message to a specific window requesting a specific piece of information: the message is queued on the message queue of the window and processed asynchronously. Active Accessibility® is a framework for enabling the use of a Windows®-based application to people with disability, which allows the instrumentation to programmatically query the content of windows and widgets using a uniform, widget-independent API. Active Accessibility® (which also provides a collection of callback functions, discussed later) is a very appealing framework, but has several limitations: not all standard window classes are accessible, not all the content of standard window classes is visible through Active Accessibility® , and third-party applications that use proprietary widgets are not required to provide Active Accessibility® support to be Windows® compliant. The inquiry messages, on the other hand, have a typically wider coverage than accessibility methods both in terms of widget types and of retrievable content. However, each class of widget responds to specific messages and this makes the instrumentation complex and onerous to maintain. Windows® provides a mechanism for registering platform-wide callback functions, called hooks. A user can register a variety of hooks, including Active Accessibility® hooks, which together yield a partially redundant set of notifications of user actions and application responses. Some of the hooks callback provide generic low-level information. Those that provide higher-level, specific information generally require knowledge of the internal data structure of the specific application with which the triggering event is associated in order to be useful to the instrumentation. Registering all the available hooks also yields a voluminous stream of callbacks.

The Windows® instrumentation support has several limitations. First, the system messages and calls do not necessarily behave as specified in the documentation (the main example is the call that retrieves the handle of the window receiving a mouse click, which ofter returns a sibling of the window). Also, callback functions not called uniformly and consistently: for example a double click sometimes results in four callbacks (mouse down, mouse up, mouse down, mouse up) and sometimes in three (mouse down, mouse down, mouse up); callbacks of specific events are known to be unreliable; the Active Accessibility® hooks work only with accessible windows, and sometimes in an non-uniform fashion (drag-and-drop events are produced by applications but not by the system).

Since Sheepdog is cross-application, it needs to capture the content of all system and application windows. To minimize its impact on user experience, Sheepdog's instrumentation must perform state capture selectively and incrementally. The volume, lack of uniformity, and unreliability of the notifications led us to discard a reactive (widget-notify) approach to state capture. We selected instead a proactive (instrumentation-request) approach in which only selected hooks are registered. The instrumentation observes the stream of callbacks, and decides accordingly when to selectively retrieve the content of windows. The rules for deciding when to retrieved were constructed to be general (i.e., application and widget-type independent), and the rules for selecting which windows to capture when a rule fires were constructed to avoid missing widget state changes for which no consistent notification exists.

The action-capture instrumentation of Sheepdog detects mouse clicks and keyboard key presses. Detecting events at higher levels of abstraction would require a very complex analysis of the stream of widget change notifications, as well as actual knowledge of the internal data structures of individual applications, and therefore was not deemed feasible. As a consequence, an abstraction layer is used in Sheepdog to combine low-level user-generated events into high-level action descriptions.

Generality and uniformity were again the guiding principles in designing the action execution layer of Sheepdog. As a consequence, a mouse-and-keyboard-emulation approach was selected. This approach actually allows seamless exchange of initiative between the user and the system, which proved to be effective and useful to incrementally construct a model of the task.

Sheepdog's instrumentation suffers from a variety of limitations. Timing issues are somewhat problematic: window messages used to retrieve window contents are not processed synchronously, and there is no guarantee that the response reflects the actual window state at the time the request was issued. Inconsistent behavior of the notification mechanisms requires a variety of strategies to overcome the resulting limitations. The content of certain windows, including the

client areas of many applications, cannot be obtained using the mechanisms described above (the alternative is to take a bitmap of the screen, segment it, and apply OCR techniques to extract the text). Capturing the content of certain windows is extremely time consuming. In particular, it is possible to capture the content of an HTML page displayed by a browser using Active Accessibility® calls, but if the page is large, this might take several seconds, which totally destroys the user experience.

#### DocWizard

The second system we consider is "Follow-me Documentation Wizard" or DocWizard, an intelligent documentation system authored by demonstration. DocWizard is implemented on the Eclipse platform, a Java-based environment for developing and running applications. During an authoring phase, DocWizard observes one or more experts performing a procedure, captures the content of the Eclipse-based applications, performs selected bitmap captures of appropriate windows, and constructs a model of the procedure representable as a script. During playback, DocWizard observes a user performing the recorded procedure, matches the actions to the procedure model, and suggests the next actions to take. The user is presented with a script-like high-level procedure representation, and is presented in-context documentation. This documentation includes the appropriate screen shots obtained during recording, where the widget to be activated is highlighted. At the same time, DocWizard identifies and highlights the widget on the GUI of the actual application. DocWizard can also perform part or all the procedure on behalf of the user, except for steps that explicitly require user input.

The Eclipse environment differs from the Windows® environments in two main respects. First, the number of widgets provided with the standard widget toolkit (SWT) is limited. Second, the source code is available and modifiable, and bugs can be clearly identified using the debugger, reported, and quickly corrected through a formal process.

Eclipse supports registering system-wide callback functions, which work consistently and uniformly. Therefore, DocWizard can use a reactive (widget-notifies) approach to deciding when to capture GUI content. The exception is when DocWizard starts operating: here it actually captures the entire GUI. To retrieve the content of widgets, the DocWizard instrumentation uses Java introspection. Timing issues are entirely avoided by executing the callback function in the GUI thread: the capture of the widget state is therefore synchronous with the operation of the GUI and the instrumentation can obtain a description of the widget content at the time of the callback invocation. Since the number of standard widgets is limited, the DocWizard instrumentation contains widget-specific state-capture methods. The main design decisions were the selection of the events that trigger state capture, which in general are widget-dependent. Most of the problem encountered with the Sheepdog state-capture instrumentation did not appear in DocWizard, because of the more restricted widget set, and of consistency of the notifications and of the general behavior of inquiry functions.

Action captures are done at the highest semantic level available from the callback functions. In particular, actual mouse clicks are ignored; instead their effects (list selection, checkbox checks, menu popups, etc.) are observed and used as action description. A lightweight abstraction layer is used to create actions corresponding to typing strings, selecting multiple items, and navigating a sequence of nested menus. The availability of reliable notifications of high-level events that exhaustively cover all SWT widgets was the enabling factor that allowed the design of an efficient and compact action-capture instrumentation.

Action execution relies on an open-source GUI test framework called Abbot for SWT. Abbot emulates mouse clicks and keyboard presses on the SWT widget set, and provides an almost full coverage of the desired operations. Keyboard and mouse emulation is desirable, since a main intended use for DocWizard is intelligent tutoring. The principal limitation of the execution layer is brittleness to user interference. This behavior is partially by design: DocWizard is meant as a mixed-initiative cooperative tool where the user can take over the execution at will. An open problem is how to discriminate between an intentional and an accidental interference of the user with a procedure execution.

## SUGGESTIONS FOR PLATFORM DESIGNERS

Based on our experiences with instrumentation we can propose a set of requirements for future operating system and application platform development. Fulfilling these requirements would greatly enhance the ability to layer intelligent systems on top of existing applications.

1. Remove the need for keeping a separate world model by providing hooks at the right points. This would require an ability to register callbacks that are to be invoked prior to any effects of actions on the UI (prior to all application callbacks, as well as any changes to widget state).

2. Provide high-level events. It should be possible to register for events such as "select menu item" in addition to lower-level mouse and keyboard events.

3. Provide a uniform introspection mechanism for the widget hierarchy. All widget types should expose all state information, with a consistent API for information access.

4. Provide high-level mechanisms for specifying hierarchical UI structure, including specification of semantic relationships between entities. For example, the API should allow the developer to specify that a particular label is the caption of an input field. Note that web-based UI-specification languages such as XForms [2] do exactly that.

5. Provide an application model that exposes high-level actions and application state. The ability to capture actions at the application level would allow semantically equivalent UI actions (e.g., either typing a hot key sequence, or selecting a menu item) to be identified as equivalent. Making state available would allow for richer interpretation of user actions; non-visible information as well as visible information could be used to inform inferences of user behavior.

6. Provide consistent high-level ways of actuating widgets. Actuation at the same semantic level as that reported by the API (described in 2) would allow good support for mixed initiative, minimize problems with user interference with automated execution, and simplify the planning task for automated execution modules.

## DISCUSSION AND CONCLUSIONS

We have described a framework for supporting advanced user-interface functionalities. This framework is centered on a state-action model of the user/application interaction. It provides a uniform view of the user operations and system responses while hiding the specific details of how individual applications (or even a platform) actually works.

We have identified the characteristics required of the state-action model to support a diverse set of advanced UI features, including intelligent tutoring, automated GUI testing, programming-by-demonstration and adaptive documentation. We have translated these characteristics into requirements for an instrumentation layer, and we have discussed the trade-offs involved in how to satisfy these requirements. Finally, we have described two systems, the Windows®–based Sheepdog and the Eclipse-based DocWizard, that rely on the state-action model.

We believe that the state-action model is very general and widely applicable. It has, nevertheless, some intrinsic limitations. The most important, which has impact on the instrumentation, is that states and actions need not be synchronized in the way assumed by the model. In particular, an advanced user might be able to perform a procedure more quickly than the application can display the result of the actions. The instrumentation is therefore notified of a user action before the effects of the previous one are visible. An action would then be associated with a state in flux. Short of heavily interfering with the user, the instrumentation cannot typically correct problems of this nature, and other components of the system must address them.

We finally remark that an interface such as the one described in this paper can be more than just an enabling technology: in some classes of systems it is the key component. In particular when automatic task-model induction from observations is involved, one can safely bet that a mediocre induction algorithm provided with high quality data will consistently outperform a high quality induction algorithm provided with mediocre data: thus, the interface layer plays a role that is even more important than that of the induction mechanism. Similarly, when a good user experience is desired, the interface should be sufficiently optimized and efficient to minimize its interference with the user/application interaction. A poorly designed instrumentation, for example, can produce a low-quality user experience even if the rest of the design is exemplary.

## ACKNOWLEDGMENTS

We would like to thank Dr. John J. Turek for the numerous insightful discussions.

## REFERENCES

1. The Rational approach to automated testing. White Paper TP-303, Rational Software Corporation, Cupertino, CA, Jan 1999.

2. Eclipse platform technical overview. White paper, www.eclipse.org, February 2001.

3. A. Cypher, editor. *Watch What I Do*. The MIT Press, Cambridge, MA, U.S.A., 1993.

4. R. Fikes and N. Nilsson. STRIPS: a new approach to the application of theorem proving. *Artificial Intelligence*, 2:189–208, 1971.

5. E. Horvitz, J. Breese, D. Heckerman, D. Hovel, and K. Rommelse. The lumiere project: Bayesian user modeling for inferring the goals and needs of software users. In *Proc. Fourteenth Conf. on Uncertainty in Artificial Intelligence*, pages 256–265, Madison, WI, U.S.A., July 1998. Morgan Kaufmann.

6. T. Lau, S.A. Wolfman, P. Domingos, and D.S. Weld. Programming by demonstration using version space algebra. *Machine Learning Journal*, 53(1–2):111–156, 2003.

7. Tessa A. Lau, Lawrence D. Bergman, Vittorio Castelli, and Daniel Oblinger. Sheepdog: Learning procedures for technical support. In *Proc. of 2004 Int. Conf. on Intelligent User Interfaces (IUI 2004)*, pages 106–116, 2004.

8. H. Liebarman, editor. *Your Wish is My Command*. Morgan Kaufmann, San Francisco, CA, U.S.A., 2001.

9. Tom Murray. Authoring intelligent tutoring systems: an analysis of the state of the art. *International Journal of Artificial Intelligence in Education*, 10:98–129, 1999.

10. D. Oblinger, V. Castelli, T.A. Lau, and L.D. Bergman. Similarity-based alignment and generalization: A new paradigm for programming by demonstration. Technical Report RC23140, IBM Research Division, 2004.

11. G.W. Paynter. *Automating iterative tasks with programming by demonstration*. Ph.D. dissertation, The University of Waikato, New Zeland, 2000.