A Task-based Architecture for Application-aware Adjuncts

Robert Farrell
Peter Fairweather
T J Watson Research Center
Yorktown Heights, NY 10598 USA
+1 914 945 {3398,2138}
{robfarr, pfairwea}@us.ibm.com

Eric Breimer
Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY
+1 518 276 6907
breime@cs.rpi.com

ABSTRACT
Users of complex applications need advice, assistance, and feedback while they work. We are experimenting with "adjunct" user agents that are aware of the history of interaction surrounding the accomplishment of a task. This paper describes an architectural framework for constructing these agents. Using this framework, we have implemented a critiquing system that can give task-oriented critiques to trainees while they use operating system tools and software applications. Our approach is generic, widely applicable, and works directly with off-the-shelf software packages.

Keywords
Adjunct, agent, architecture, critic, event, graphical user interface, plan recognition, task model.

INTRODUCTION
Help systems for software applications typically provide information on operating an application's complex commands and options without giving the user help completing tasks [8]. While computer-based instruction [2] may provide effective support for new users executing common tasks, more advanced users are left without adequate assistance. To address this shortcoming, we have been experimenting with "adjuncts" to software applications that can be easily authored to handle new tasks. Our adjuncts are user agents that execute alongside an application or set of applications, take task advice from the user, and provide task-specific support in the context of the user's work. Unlike traditional methods of training and assessment, our adjunct monitors actual behavior with live software applications.

PROBLEM
We are building a prototype adjunct, the Task Critic, for Networking System Administrators. Trainees will download the system following a web-based course or classroom instruction. When they need practice or assistance with particular job tasks, they can perform them with the help of the Task Critic. The critic provides an individualized evaluation of the user's solution steps as well as hyperlinks to help pages and supporting training materials.

Let's take an example. Suppose a new system administrator needs to find the speed of the COM1 port on an employee's computer. He needs help, so he brings up Task Critic and searches for this task. Once he selects the task, the system immediately starts monitoring his actions on the desktop, in command shells, and within certain applications.

The administrator selects the Printers icon. The task critic explains that the Printers tool shows available printers, not information about COM ports. Next, the administrator starts up the Notepad text editor and loads the autoexec.bat file. The task critic explains that autoexec.bat lists commands run at startup while he wants a file that lists configuration settings. Finally, he loads the win.ini file and finds the speed of the COM1 port. The system stops monitoring and the user continues his work.

ARCHITECTURE
Our proposed architectural framework (see Figure 1) supports the interaction between users, applications, and adjuncts. Our adjuncts are user agents that are conjoined to applications, attend to their events, and instead of being active, are subordinate to the user.

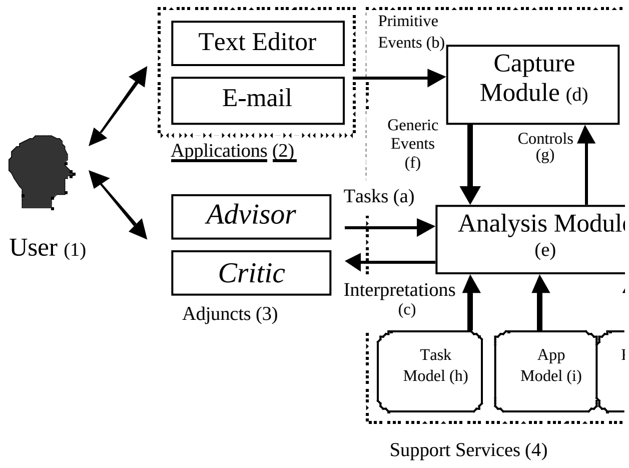


Figure 1: Architecture for Adjunct Agents

Users (1) can work on more than one target application (2) (e.g., a Text Editor and E-mail) to accomplish a task and can interact with more than one adjunct (3). Support Services provide both task and application tracking.

To receive application-aware communications from an adjunct user agent, a user must register their identity with the agent and optionally select a high-level task or tasks (a). As the user works, the Support Services interpret the events being generated (b) as applying to these tasks. The agent can use this information (c) to provide in-context assistance, performance summaries, directed feedback, or detailed critiques.

Support Services
Support Services comprise two major components: the Capture Module (d) and the Analysis Module (e).

Capturing Events
Our Capture Module can "spy" on events being generated by one or all applications. It can capture both user events and application events. User events ("inputs") are the result of actions the user actually took on the application: key presses, mouse movements, dialog use, and window controls. Application events ("outputs") are generated by the application to control itself, typically in response to user events: creation and destruction of user interface widgets, painting of windows, and population of default values in text fields are all examples of such events.

We chose to represent primitive events as n-tuples where the first element of the tuple is an enumerated event type (e.g., ActivateWindow, SelectMenu, PressKey) and the other elements are strings of characters, usually representing names of windows. "Filtering controls" (g) allow the capture module to discard certain classes of events from consideration and enable agents to suspend and resume the capture process.

Our implementation uses the "hooks" provided by the Microsoft Windows operating system to capture events. Our event processor is essentially a tree where each node tests a different attribute of the Windows event structure. An event that passes all of the tests is converted in a *generic event* (f) and is passed on to the analysis module.

Analyzing Events
The Analysis Module receives events from one or more Capture Modules on different computer systems. If the computer supports multiple users, the Capture Module also report the user's identity, implemented as a Windows login name.

The Analysis Module adds a time stamp to each incoming generic event, then performs a bottom-up reconstruction of the user's hierarchical task structure, starting with the incoming generic events.

Each generic event is matched against the available recognition rules. The recognition rules typically do four things:

1. Determine the context, if any (e.g., entering a window).
2. Match events of various event classes
3. Check semantic constraints (e.g., are the window titles the same?) and temporal constraints (is the action before the result?)
4. Create a new inferred event, justify the event with the matched supporting events, and remove the supporting events from further processing.

The Analysis Module alternates between reading events and matching recognition rules. Events that do not match the current context are put into buffer to be matched later. Otherwise-equivalent more-recent events will be preferred during matching.

Once a new inferred event has been added, the system stores the parent-child relationship to the supporting events. The system immediately matches inferred events against other recognition rules. Thus, event recognition proceeds depth-first. The growing "forest" of inferred events is called an *interpretation* of the generic events (see Figure 2).

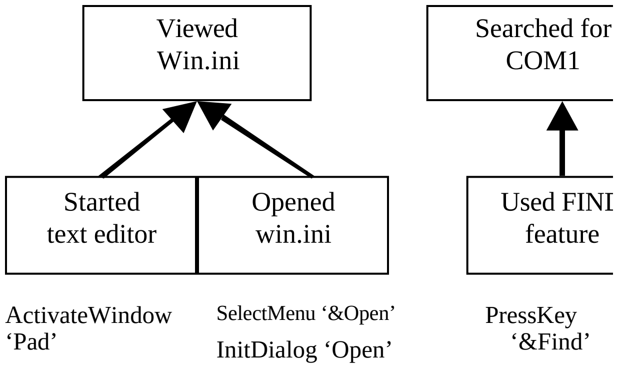
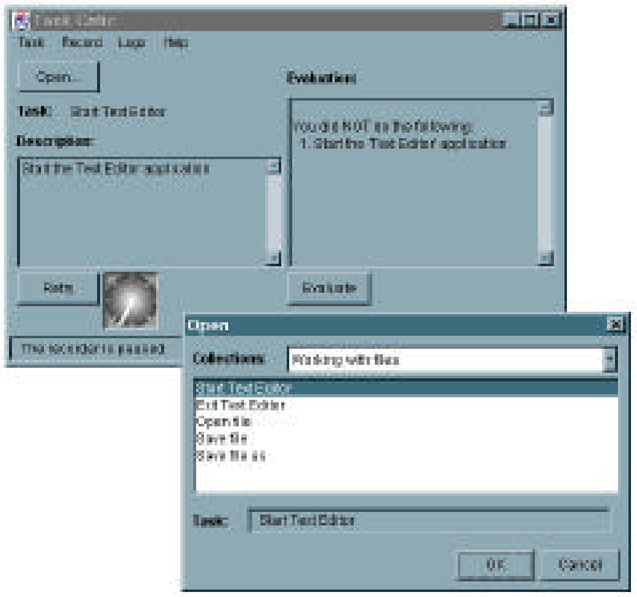


Figure 2: An Interpretation of the Event Stream

The work done by the Support Services is similar to that done in many programming by demonstration systems (for example, see [7]). However, the emphasis in our system is on building hierarchical plans from events, not in generalizing from examples.

PROTOTYPE AGENT: Task Critic
We have tested our architectural framework by building a system for critiquing users of basic Windows operating system tools and desktop applications. Our Task Critic (see Figure 3) is a separate performance support tool that is available as an icon on the trainee's desktop. It allows the trainee to choose a relevant task collection (e.g., Working With Communications Ports), select a task (e.g., Determining COM Port Characteristics), and then perform that task using the applications installed on their desktop.



Task: Start Text Editor

Description: Start the Text Editor application

Evaluation: You did NOT do the following:
1. Start the 'Text Editor' application

The recorder is paused

Collections: Working with files

Start Text Editor
Exit Text Editor
Open file
Save file
Save file as

Task: Start Text Editor

Figure 3: Task Critic

The Task Critic consists of the adjunct user interface, a task model, an application model and an evaluation module that computes the critique.

Adjunct User Interface
The Task Critic user interface consists of a single primary window with popup dialogs for browsing tasks and help. The task browser allows the user to select their desired task from a list of task collections. A 'tape recorder' graphic indicates that monitoring has begun. The small, resizable critic window allows the user maximal screen real estate for performing the task on the application, while the dual-pane design allows the user to see both the task description and the system's evaluation simultaneously. The critic provides on-demand, incremental evaluation.

Task Model
Tasks are represented as hierarchical AND/OR trees of goals and subgoals (see

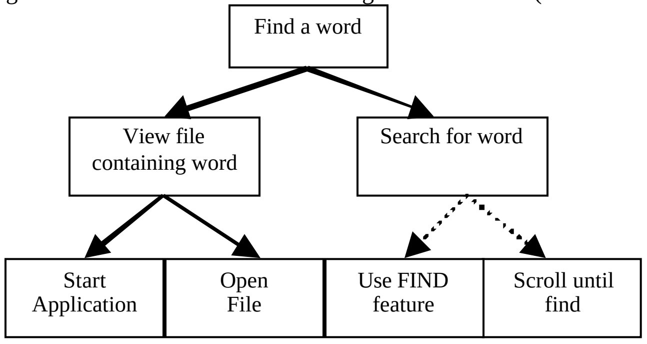


Figure 4). All subgoals are ordered linearly unless otherwise noted. Goal descriptions are identical to event descriptions, except goals may contain element variables that match any string.

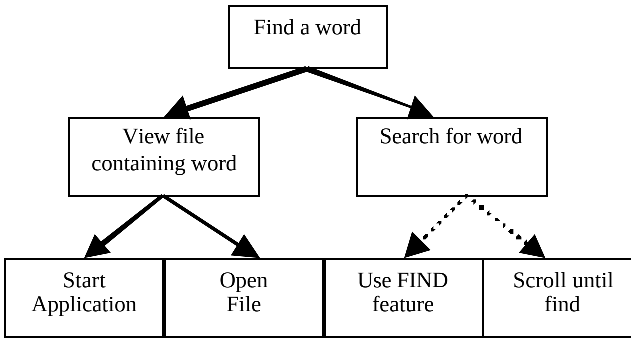


Figure 4: Task model

Application Model
The Application Model must be pre-populated for each target application. It enables the recognition rules to be completely application-independent. The application model for the text editor includes parent/child relationships between windows and cause/effect relationships between pressing a button (e.g., Search|Find) and bringing up a window (the Find window).

The Application
System administrators work with many applications, but this example uses only one: the Notepad text editor. The event types for the text editor consists of operations on applications (Start/Exit), windows (Open/Close), files (Save), and documents (searching, editing).

When running the capture module on the Notepad text editor, we found that window titles were insufficient to differentiate application windows, so our event descriptions include the user interface widget. In our previous work [4], we used unchanging portions of content for unique window identification. For some applications, it may be necessary to include the relative position of windows or other information.

Adjunct
Our Task Critic evaluates the user's inputs by starting with the most recent task. The system recursively matches the goals in the goal descriptions against the events in the interpretation, starting with the high-level goals. If a goal matches an event, then the goal is marked as *achieved*, the pairing is stored, and it's subgoals are matched against the event's supporting events. If the goal is a near miss (only one element of the two n-tuples does not match), the pairing between goal and event is stored. If the goal does not match any events, then it counts as *omitted*. A second pass is made over the interpretation, recursively querying each of the events to see if it has been paired with a goal. The events that are highest in the interpretation forest are returned as *unexpected*. The system passes the lists of achieved, omitted, near miss, and unexpected events through a simple dialogue generation procedure and uses template-based natural language generation to create the final output to the user.

DISCUSSION AND FUTURE WORK

Critiquing is an effective mechanism for improving human behavior, especially that of near-expert users. The critiquing method introduced here encourages reflexive problem solving and challenges the user to create self-explanations [1] for perceived expectation failures. A critiquing system can also reduce the “gulf of evaluation” [6] – the effort required to interpret the feedback provided by a system.

Expert critiquing systems have had a large impact where there is a good source of expert knowledge and where complex constraints must be fulfilled (e.g., spelling checkers) [9]. This is precisely the situation we find when the user is presented with a real-world task such as systems administration. The challenges of the real task create a useful role for an adjunct agent.

Critiquing can be difficult because users can perform any actions between the time they start the task and when they ask for an evaluation. Unlike many critic systems, which are only checking constraints on final solutions, our system examines the entire history (see [3]).

In the future, we would like to extend our framework to handle collaborative work in organizations. We need to include primitives for describing constraints on resources and users as well as timing.

CONCLUSIONS

We have introduced an architectural framework that enables user agents to interpret and respond to events originating from other applications. Our architecture separates the interpretation of events from how they are used. It is able to capture events from multiple applications on multiple computer systems. We have showed how we have built a task-oriented critiquing system that uses the hierarchical interpretation of events to provide assessment and assistance. By separating event interpretation from task goals, the system supports critiquing of the entire history of events, including unexpected and near miss events.

ACKNOWLEDGMENTS

We would like to thank Dan Oblinger for numerous discussions, Jacob Ukelson for his interest and support, and Dick Lam for helping with the GUI code review.

REFERENCES

1. Conati, C. and VanLehn. Teaching meta-cognitive skills: implementation and evaluation of a tutoring system to guide self-explanation while learning from examples. In *Proceedings of AIED’99: the 9th World Conference on Artificial Intelligence and Education*, Le Man, France, 1999.
2. Gibbons, A.S. and Fairweather, P.G. Computer-based instruction: design and development. Educational Technology Publications, Inc.: Englewood Cliffs, NJ.
3. Farrell, R. Capturing Interaction Histories on the Web. In *Proceedings of the 2nd Workshop on Adaptive Systems and User Modeling on the WWW*. Toronto, CA, 1999.
4. Farrell, R. and Lefkowitz, Lawrence S. Supporting development of task guidance for software system users: lessons from the WITS project. Bloom and Loftin Eds. *Facilitating the development and use of interactive learning environments*, pp. 127-162, 1998.
5. Fuerzeig, W. & Ritter, F. Understanding Reflective Problem Solving. Psotka, J., Massey, L.D., & Mutter, S.A. (Eds.), *Intelligent Tutoring Systems: Lessons Learned*. Lawrence Erlbaum Associates, Hillsdale, NJ. 1988.
6. Norman, D. and Draper, S.W. eds. User-Centered Design: New Perspectives on Human-Computer Interaction. Lawrence Erlbaum Associates: Hillsdale, NJ, 1986.
7. Lieberman, H. Modrian: A Teachable Graphical Editor. Cypher, A. et al (Eds.) *Watch What I Do: Programming by Demonstration*. The MIT Press: Cambridge, MA, 1993.
8. Priestly, M. Task-oriented or task-disoriented: designing a usable help web. In *Proceedings on the sixteenth annual international conference on computer documentation*, 1998, pp. 194-199.
9. Silverman, B. Survey of Expert critiquing systems: practical and theoretical frontiers. *CACM Vol 35*, pp. 106-127, 1992.