Infrastructure for Tracking Users in Open Collaborative Applications: A Preliminary Report

Gal A. Kaminka and Danny Shimoni

The MAVERICK Group
Computer Science Department
Bar Ilan University, Israel
galk@cs.biu.ac.il, dannys@myrealbox.com

Abstract. Open collaborative applications which utilize user-modeling techniques pose several challenges to the basic capability of tracking the users' actions. In such environments, users may use multiple applications—which can differ between users. Unfortunately, existing approaches to tracking actions either rely on access to the applications' source code, or on their supporting scripting, or on their being written in specific languages. Unfortunately, all of these conditions frequently do not hold in open settings. To address this challenge, we are developing a new measurement tool for the user-modeling community, which relies on intercepting the API calls between the applications and the operating system to detect users' actions. Although in initial state of development, it is already capable of tracking collaborative scenarios which previous approaches could not. We report on the design of the tool and the challenges it raises.

1 Introduction

Collaborative applications increasingly utilize user modeling techniques, which infer the user's goals, preferences, and intent. This inference process relies in turn on being able to track the user's actions when interacting with the application. For instance, the inference process may track the user's selection of web sites, the user's choice of buttons and menu items in different applications, the user's reaction when she receives email from a colleague, or a subordinate in the organization, etc. [4]. Based on these tracked actions, a user modeling algorithm is able to predict future actions, infer the user's intent, provide assistance, or help maintain collaboration (e.g., by noting that a user has taken steps which support the team's task).

Open collaborative environments pose several challenges to such user-tracking mechanisms. In such environments, multiple users may be using multiple applications—where those applications are not necessarily the same for all users. For example, different users may use different preferred email clients or web browsers, etc.

A first challenge is to track the same actions in different applications. Since different applications have different action repertoires (even for the same intended effect), it becomes difficult to capture situations where different applications are doing the same thing. For instance, saving a file can be done in multiple ways in GNU Emacs (from the keyboard "ctrl-x ctrl-s" or using the GUI), and is done differently in Windows' Notepad ("ctrl-s" from the keyboard, standard file dialog in the GUI). Thus if we could somehow track actions on multiple applications, we would be faced with the challenged of deciphering them differently, depending on the application.

A second challenge is to track a single user's actions as it interacts with several applications as part of the user's individual task work within the collaborative process. For instance, a user may receive an instant message from a colleague (using ICQ), and then based on this message, open Internet Explorer to a specific URL. A user-modeling component would need to be able to recognize both actions, across the different applications, in order to correctly identify and respond to this scenario.

Traditional closed collaborative applications could rely access to their own internals in order to track the user's actions and feed it internally into the user-modeling process. In other words, the designer of the user-modeling capabilities could rely on begin able to modify the source code of the application itself in order to add tracking facilities (see Section 2 for details).

Unfortunately, one cannot expect in practice to be able to modify the source code of multiple applications—for reasons of proprietary information and modification costs[1]. Thus a different mechanism is required in order to facilitate tracking in open collaborative environments. Unfortunately, previous investigations have either provided solutions that require access to the GUI system source code [9], or were limited in scope to applications that had scripting capabilities [5, 6], or were written in specific languages [10, 2].

This paper describes our initial steps at tackling these challenges, by developing a tool that relies on two components to address the challenges of open environments. The first component manages operating system *hooking* to intercept events at the level of GUI and operating system API, without having to modify the source code at either level. This translates GUI and operating system actions into a stream of raw atomic events corresponding to API system calls. Any interaction of the user with an application causes events to be generated. For example, moving the mouse across text causes multiple events to be generated which indicate mouse movements, re-drawing of the text below the mouse, etc. Thus careful parsing of these events allows capturing a user's actions user[2].

The second component in this tool allows the user to define such parsing behavior, using multiple deterministic finite-state automatons (DFAs). States in these DFAs correspond to parsing progress, and transitions correspond to intercepted atomic events. This parsing component addresses the challenges of interpreting different action sequences that have the same intended affect. For instance, it is easy to set up multiple DFAs that interpret several different keyboard command sequences as a *save-file* event. Similarly, an DFA can be built which corresponds to recognizing two applications opening in a specific order. The DFA will have a transition from the starting place to an intermediate place, where the condition on the transition corresponds to the "*open application*" event (with a specific name). It will then have a second transition from the intermediate

[1] We remind the reader of the cost of "Year 2000 Bug" modifications.

[2] Naturally, this approach raises serious security and privacy concerns. However, these are beyond the scope of this paper.

state to an accepting state, where the condition on this second transition corresponds to intercepting a the second "*open application*" event.

This paper is organized as follows. The next section presents related work and motivates the creation of infrastructure for user modeling in open settings with a concrete example. Section 3 presents the design for the tool, and some of the challenges we are facing in building it. It also shows how the tool can be used to address the motivating example presented earlier. Finally, we conclude and present our plans for future developments.

2 Related Work

Previous work in user modeling can be classified into two categories. One class of previous investigations exemplifies the traditional approach and its limitations. Another class of previous work focuses on methods of overcoming these limitations. Our work is mostly related of course to this second class of investigations. However, to motivate our approach, we discuss here a good example of the traditional approach, presented in [7] and [4]. In this previous work, the authors changed the source code of an email application, in order to modify their original behavior. They used the MBR algorithm [11] to predict the next user's action. The advantage of this approach is that the agent is more specific for the problem domain. However, this requires access to the source code of the applications themselves. But the effort spent results in the ability to track a single application—and the effort requirements do not scale up with multiple applications.

There have been few previous investigations of non-obtrusive tracking, which does not rely on modifying the application. [9] developed the *WOSIT* (*Widget Observation Simulation and Inspection Tool)*, software which observes a user's actions on an X11 program's user interface, while requiring no modifications to a program's source code. The software produces output such as: *window(open, "PromptDialog", 106)*, meaning that a window with name *PromptDialog* was opened, after 106 seconds from the start of WOSIT. This software is designed only for UNIX platforms that use the X Intrinsics (Xt) and Motif (Xm) libraries environment. It works by compiling special versions of the Xt and Xm libraries, which the user can then use instead of the standard versions. While we are inspired by the work on WOSIT, our tool (when completed) will have several distinct features. First, WOSIT only tracks GUI events, and does not track communicated information, such as incoming messages, or outgoing web requests. Thus WOSIT cannot track information flowing in and out of the individual user's task space. In contrast, the tool we are developing is capable (even in its current development stage) of tracking the arrival and departure of instant messages, and web URLs, and is therefore capable of tracking user's responses to information coming in from other users. Second, WOSIT targets a specific GUI system (Motif) in Unix, which is unfortunately not very popular (in terms of market share in personal computers—WOSIT does not work for Linux). Thus WOSIT is unfortunately not usable in many situations where we would like to track users (as opposed to researchers). Our tool targets Microsoft Windows, since we want our tool to be used with most common users. Indeed, some of the experiments we intend to pursue involve tracking common collaborative applications that are in use in most offices and home users today (such as Outlook and Internet

Explorer), but are not tied to a single groupware or collaboration application. Finally, WOSIT has a rigid output format, which may not be appropriate for all modeling algorithms (since these vary in their input requirements—for instance some requiring knowledge of actions and state, e.g., [10], some only of actions, e.g., [1], etc.). Since our tool uses researcher-defined DFAs to interpret sequences of atomic events, it enables customization of the output to match the requirements of the user-modeling algorithm.

[2] suggest installing hooking functions in the Java Virtual Machine (JVM). This allows them to monitor specific events in applications built using Java. They suggest new algorithm, ONISI for the predicting of next user's action. The disadvantage of this technique is that it monitors only a single application at a time, therefore we need to develop agent per application. Moreover, most applications in current use today (certainly in the Windows environment) are not delivered in JVM byte code, and thus the applicability of this approach is again limited. Similarly, [10] suggest monitoring the user's actions in Smalltalk environment, by spying the interface tool (browsers, debuggers, inspectors and editors). They propose an algorithm called IBHYS for predicting next user's action. One disadvantage of this approach is that there are a lot of actions that the user does behind or without the interface (such as actions that occurred during the idle time or batch commands), which can be missed. In contrast, using our tool will produce more resolution about user's actions, since it monitors the core of the OS.

[5] [6] suggests to attach the interface agent functions to an applications by adding script code. There are no discussions on the user modeling algorithms. This technique assumes that applications have their own scripting language, which we believe is not a frequent case. And unless all applications support this functionality, this approach will again be limited to tracking only a limited applications, and thus won't be appropriate to open settings.

As a concrete example of settings in which it would be difficult to track users with the technology discussed above, consider the following scenario, in which two users collaborate spontaneously. Suppose Alice has noticed a web-site which she believes her teammates should know about (for instance, the publications web page of an interesting research group). Alice logs into one of the popular instant messaging clients, ICQ (also known as the America Online Instant Messenger), and sends the URL for the web page, in a short message, to her colleague, Bob. Upon receiving the message, Bob is intrigued, and starts up his favorite web browser, Internet Explorer. He clicks on the URL in the message, or uses copy-paste to put it into the browser, and then looks at the web page.

Obviously, we want to be able to track collaborative interactions such as these, in order to build or utilize user models—for instance to improve collaboration. However, none of the tools/techniques described above are able to do this, for the simple reason that none of the applications involved have been built to support such tracking, and we cannot access their source code in order to add such support. Moreover, even if somehow we could add such support to ICQ and Internet Explorer, we would still be faced with the task of then adding the support to any other application which our tracked team-members may use. For instance, it could be that Alice uses Netscape, and Charlie—a third teammate—uses Mosaic (a different browser client).

3 Tracking Users in Open Collaborative Applications

The aim of our system (called TMA, for *Tracking Multiple Applications*) is to collect data about all the activities between the user, the applications and the operating system. In this section, we describe the system architecture that we have chosen, and its different components. Figure 1 presents this architecture in detail. The system has two main components: (i) the Hook Server Application, a standalone application which manages the interception of events by inserting special hooked versions of the functions into the operating system; and (ii) the Dataset Builder Application, another standalone application which connects to the Hook Server (via sockets) to receive information about the events intercepted. This Dataset application manages the DFAs which transform sequences of atomic events into complex events. The Dataset Builder Application is itself a server process, to which client user-modeling applications can connect (again, using sockets) and request the complex events.

The design of the TMA application was motivated by several constraints and goals. First, the separation of the hooks from the interpretation of the events allows flexibility in how the information is used for different purposes. In principle, multiple Dataset Builder applications, each having its own set of DFAs, can connect to the same Hook Server to receive the same stream of events. Moreover, the Hook Server may serve information to a Dataset Builder application running on a different machine, thus facilitating tracking of multiple users remotely.

Second, we made an explicit choice to build the Dataset Builder as a server in itself, to separate it from the choice of a specific user-modeling technique. To date, most related investigations commonly focused on proving the worth of a specific algorithm, and the effort invested in tracking infrastructure was therefore not reusable by others. In contrast, the design of the Dataset Builder as a server of information allows multiple (and different) user-modeling algorithms to use the same information, and thus to be compared and contrasted easily.

3.1 Part 1 - Hook Server Application

The hook server application's goal is replacing the original API functions with new functions, using the hooking mechanism [8, 3]. As a result, we can intercept every API calls that occurs in the operating system, before it reaches the target application. This allows us to record any information sent with the function call, such as the originating application, the time of the call, and the details involved. For example, this allows us to record GUI widget definition and activation (such as menu, dialog box, etc.), communications services, I/O services and more.

The hook server targets the Windows operating system, because it has a market share of over 90of the significant differences between Windows and other operating systems, such as Linux, we do not expect to be able to reuse it with other operating systems. However, the separation of this component from the component interpreting the events (the dataset builder, described below), allows to at least localize this dependency on a specific operating system to the extent possible.

At this point, current capabilities of this component are fairly complete. However, the interception mechanism is specific to API calls, which means we must know in

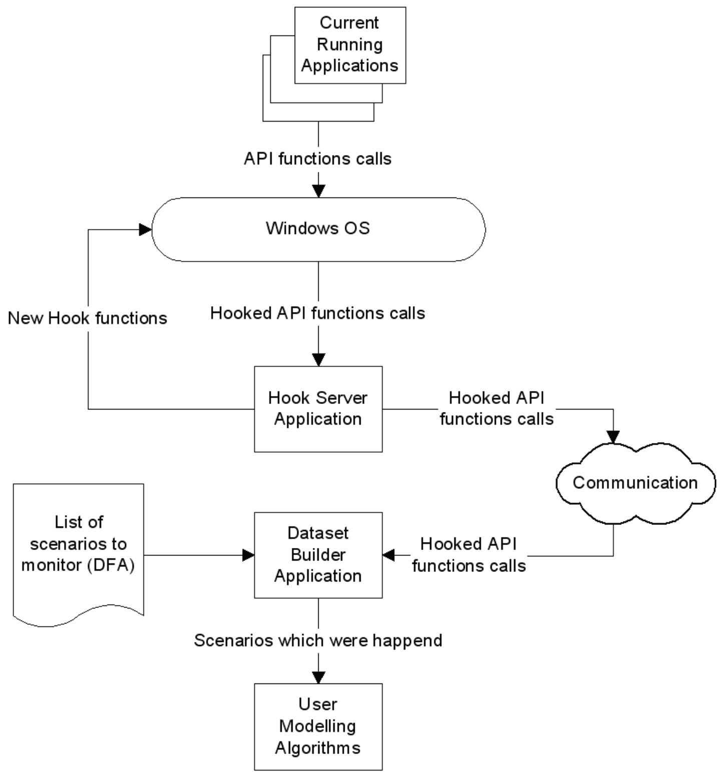

Fig. 1. Architecture of infrastructure for tracking actions across multiple applications.

advance which function calls we would like to intercept. We plan to supply the tool with a set of built-in interceptors which covers most common functions of applications. Certainly, some of the functions calls are very generic, and therefore capture important events in most applications. For instance, we report below on a scenario where our tool recognized that an incoming instant message (sent to the user using AOL Instant Messenger—ICQ) contained a web URL, which was then selected by the user and subsequently browsed in Internet Explorer. Most of the atomic events captured in this example were intercepted by hooking a single Windows function call—*ExtTextOut()*.

3.2 Part 2 - Dataset Builder Application

The dataset builder application is responsible for filtering, analyzing and transformation of crude data (which was sent by the hook server in the form of a stream of atomic events) to a stream of complex events, each signifying a recognition of some important action or state feature. To do this in a manner that allows flexibility to the user (i.e., the researcher or designer of the user-modeling algorithm which will use the information), the Dataset Builder accepts as input a file containing descriptions of multiple DFAs, where the accepting state of each DFA corresponds to an output complex event which should be sent to the user-modeling algorithm. At this point of the development, the capability for reading such a file is still missing, and its completion is indeed our current thrust.

Each DFA defines a scenario or action of interest, composed of multiple atomic events. The states of the DFA correspond to (intermediate) states of recognition, and the transitions are conditioned on intercepted atomic events. We have identified that in terms of expressibility, regular languages are insufficient in practice for describing sequences of interesting events. Thus the DFAs have to be augmented with memory so that parameters and variables can be saved and passed from state to state.

Thus for instance a simple DFA for capturing the scenario previously described above is presented in Figure 2. The starting state is $q0$. Upon tracking an atomic event signifying that a URL was received through ICQ, the automaton transitions into state $q1$, otherwise, all other events (of which one—marked *IE with URL*—is shown here) transition back into $q0$. From state $q1$ there is a transition into the accepting state $q2$, marking the tracking of an atomic event signifying that Internet Explorer was opened with the same URL The memory augmentation is used here on the transitions $qo \rightarrow q1$ and $q1 \rightarrow q2$. Using the DFA's memory, we can verify that the same URL sent via ICQ (the transition into $q1$) was indeed the one used in browsing through Internet Explorer (the transition into the accepting state $q2$).

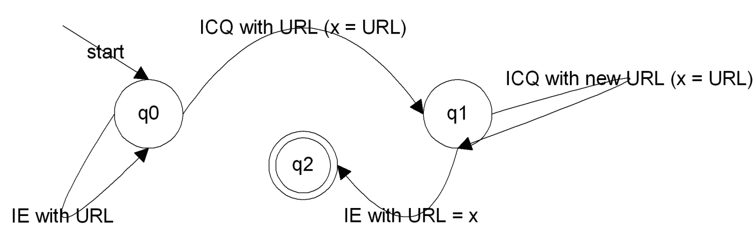

Fig. 2. Augmented DFA for recognizing the user receiving a URL in an ICQ message and then browsing it.

4 Discussion, Current State, and Future Work

The TMA is intended to be a new measurement tool in the science of user modeling. It is currently initial state of development. It can successfully recognize the scenario discussed above, and many others that we had experimented with. For instance, it can also be used successfully to cause the web browser to open with a URL, as soon as the URL is received in an ICQ message. We are currently building a generic DFA mechanism into the TMA, which will allow us to define DFAs externally, such that the TMA can be configured by researchers using simple text files.

For now, the TMA employs a simple GUI which allows us, the developers, to monitor its actions and the events it recognizes. A screen-shot of this GUI appears in Figure 3. Each row is associated with an atomic event that has been received. The highlighted row corresponds to the complex event that was generated when the sequence of events (ICQ received URL, user opened URL in Internet Explorer) was recognized by the corresponding DFA. The first column shows the name of the application which triggered the event. The *PID, Mem* and *Time* columns provide the internal process ID, the amount of memory taken by the application, and the time in which the event was generated. The *API Function()* column shows the name of the intercepted function, and the last column shows the relevant information captured by the event (e.g., the URL captured).

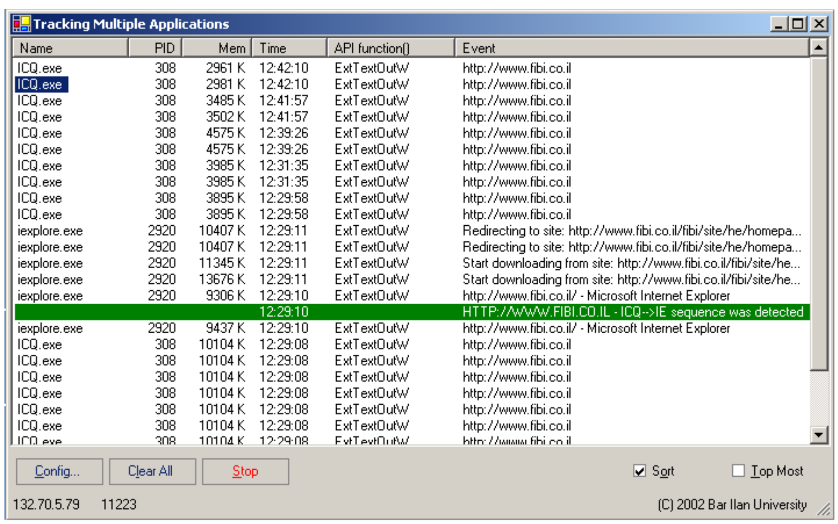

Fig. 3. A screen-shot of the TMA GUI.

We hope to be able to provide the TMA to researchers as soon as it reaches a stable state. However, several challenges (other than development) must still be addressed.

First, to our best knowledge, there exists no previous work which has examined mechanisms for tracking multiple applications simultaneously. We believe that the augmented DFAs will be powerful enough to deal with the multiple streams of atomic events that will arrive at the TMA. However, because of the multi-tasking nature of the systems being monitored, several DFAs may be applicable at once, using the same events. This is not unlike the problem of multiple matching rules in a rule-production system. We plan to tackle known strategies (such as “choose most general”, and “choose most specific”) to resolve such conflicts—but have very little to rely on in terms of the applicability of these strategies to the problem of multiple matching DFAs in the case of tracking users and applications.

Second, our examination of the relevant user-modeling literature reveals that different user modeling algorithms require different inputs. For instance, some (e.g., IPAM [1]) rely on a sequence of tracked actions, while others (e.g., ONISI [2]) require input in the form of state–action pairs. We wish the TMA to be configurable to support these different format, and are currently conducting a survey of the literature to finalize a set of conceptual inputs, which the TMA would then support. However, in any case, the DFAs would be useful in being able to interpret atomic events and generate complex events corresponding to either states or actions.

Third, there is in our mind a grand opportunity here for tracking multiple users across machines, by installing hook servers on multiple machines and then using one or more TMAs to collect information from them remotely. Thus for instance, it should be possible in principle to recognize situations where Alice is opening her a specific Word document just as Bob does the same (with the same document—a read/write conflict, potentially). DFAs are sequential in nature, and may therefore be appropriate for describing collaborative processes that are sequential in nature, perhaps with limited concurrency. We are therefore considering the use of more appropriate computational models (such as Petri Nets) for this purpose.

Indeed the opportunity to integrate information from multiple machines also touches on a key benefit of the TMA design. Since the hook server, which intercepts operating system events, is separated from the dataset builder, one can, in principle, run a hook-server built for a different operating system, and still collect information on the dataset builder portion of our TMA. At a second stage of this research, we plan to examine modifying the WOSIT tool [9] such that it supports hook-server functionality on Linux (since it currently only works for Motif on SunOS).

References

1. Brian D. Davison and Haym Hirsh. *Prediction the future: AI Approaches to Time Series Problems*, chapter Predicting Sequences of User Actions, pages 5–12. AAAI Press, 1998.
2. Peter Gorniak and David Poole. Predicting future user actions by observing unmodified applications. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, pages 217–2222, 2000.
3. Ivo Ivanov. API hooking revealed. http://www.codeproject.com/system/hooksys.asp, 2002.
4. Y. Lashkari, M. Metral, and Patti Maes. Collaborative interface agents. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, 1994.
5. Henry Lieberman. Attaching interface agent software to applications. Technical report, Media Laboratory, MIT, 1998.
6. Henry Lieberman. Integrating user interface agents with conventional applications. In *Proceedings of the ACM Conference on Intelligent User Interfaces*. ACM, 1998.

7. Max Metral. Design of a generic learning interface agent. B.Sc. Dissertation, MIT, 1993.
8. Microsoft Corporation, Microsoft Developer Network. http://msdn.microsoft.com/default.asp, 2002.
9. The MITRE corporation, WOSIT: Widget Observation Simulation Inspection Tool. http://www.mitre.org/technology/wosit/, 2001.
10. J. Ruvini and C. Fagot. IBHYS: a new approach to learn users habits. In *Proceedings of ICTAI-98*, pages 200–207. IEEE Computer Society Press, 1998.
11. Craig Stanfill and David Waltz. Toward memory-based reasoning. *Communications of the ACM*, 29(12):1213–1228, 1986.