Automating Tasks for Groups of Users : A System-Wide "Epiphyte" Approach

Zeiliger R. *, Kosbie D. **

* CNRS-GATE, 93 chemin des Mouilles
69130 Ecully
FRANCE
zeiliger@gate.cnrs.fr

** Microsoft Corporation, One Microsoft Way
Redmond, WA 98052
USA
dkosbie@microsoft.com

ABSTRACT End-users have to use macro-recorders when they want to automate tedious tasks. Their tasks often include actions from multiple applications. While some application-specific macro facilities or PBD systems have proved efficient to automate single-application tasks, system-wide - and hence application-independent - systems are few. Most system-wide recorders operate on low-level events whereas most repetitive activities are repetitive at a somewhat higher level of abstraction. We present a so-called "epiphyte" approach allowing automation of some of the repetitive tasks only, but working with unmodified applications. It uses external and partial application models to allow recording of hierarchical event histories which in turn facilitate the detection, generalization, anticipation and completion of repetitive sequences of actions. In this approach, experts are in charge of building external application models. We present some features which facilitate and quicken their work particularly in the context of a group of users.

KEYWORDS Programming By Demonstration, Macro commands, End User Programming, Interface agents, Hierarchical Event Histories.

1. INTRODUCTION

People often use generic applications developed by distant, unknown programmers to handle tasks similar to theirs (Cypher, 1993). In the process of mapping their activities into the capabilities of these generic applications, end-users are inevitably led to perform tedious repetitive actions. Automation of these tasks would be possible if only end-users were also programmers. But programming is difficult : Potter

Human–Computer Interaction: INTERACT'97 S. Howard, J. Hammond & G. Lindgaard (editors)
Published by Chapman & Hall

(Potter, 1993) identified the obstacles users have to overcome to achieve "just-in-time" programming defined as the "implementing of algorithms during task-time" : inaccessible data, the effort of entering the algorithm, limited computational generality, effort of invoking the algorithm, and risk. Various techniques have been developed to allow programming by end-user : preferences, script languages, macro-recorders and programming-by-demonstration (PBD). In most modern commercial systems, end-users have to use macro-recorders when they want to automate tedious tasks. Macro-recorders are either application-specific (such as EXCEL macro recorder) or system-wide (such as Quickeys or Automator). While the former can be efficient, the latter would be more useful because user's tasks often include actions from multiple applications. Unfortunately system-wide macro-recorders are still too "literal" (Cypher,1993) : "they replay a sequence of keystrokes and mouse-clicks, whereas most repetitive activities are repetitive at a somewhat higher level of abstraction". PBD goes a step further in creating generalized programs from the recorded actions. But again most successful PBD programs today are application-dependent (the best example is EAGER which is dedicated to Apple's Hypercard (Cypher, 1991)). Architectures have been proposed to facilitate system-wide PBD (Piernot, Yvon, 1993, 1995) (Kosbie, Myers, 1994, 1995) but none of the proposed architectural enhancements have yet been incorporated into popular systems such as Microsoft Windows.

In the next section we are presenting some of the issues related to building a system-wide Microsoft-windows based PBD system.

2. SYSTEM ARCHITECTURE OVERVIEW AND RELATED ISSUES.

The proposed system has to record system-wide user actions, detect repetitive action sequences and generalize to produce a program automating the task, under user's control. It matches the definition of software agents as suggested by P. Maes : "a set of programs which have a specific user-understandable competence, which are capable of learning about their environment and to which the user can delegate tasks that she is unwilling to perform" (Maes, 1996). Our agent has to go through the following steps :

1. Capturing in the system the low-level events (LLE) such as mouse-click and keyboard strokes. This module should not contain any application-specific code. The successive events constitute a history.

2. Generating higher-level events (HLE). This is a crucial step. A good example of a high-level event is "save-file"; different low-level events map to this higher-level event : the "control-S" keyboard stroke, the selection with the mouse of the "Save" item from the "File" menu or a mouse-click on the "save-file button" in the tool bar. In LLE history - based on mouse-clicks and key strokes - those events will be different, while the events will appear identical if we record an HLE history, thus resulting in an increase in power of the inference engine. Two sequences of actions which have the same meaning for the user can differ considerably in their expression. Crow and Smith showed that incorporating minimal knowledge about the underlying system (particularly a "small amount of low-level domain syntax") greatly improves the performance of a PBD system (Crow, Smith, 1993).

3. Including context. Events must also include application context if they are to be adequately reasoned over. The canonical example here is deleting all mail messages from a certain person. The only way to infer this behavior is to be able to reason over "who sent" the messages. This information should be part of the **context** of a "delete-mail" event. The problem, of course, is how much context should be included : in the case of an email event, we may include the entire header, but probably not the body, so any inferences requiring the body cannot be made.

Another problem when working with existing, unmodified applications is how a software agent can access this context : user interaction with applications such as text editors or graphical editors is achieved through the use of controls (buttons, menus, dialog-boxes) which have a well-defined system-wide semantic : clicking on a check box for example results in setting an option to "on" or "off" and the check box caption usually indicates the option name. On the contrary, interaction through the edit-window of a graphical editor has a specific semantic determined by the editor

itself : a mouse-click in such a window may result for example in selecting or on the contrary de-selecting an object, depending on what was the previous state of this object. The information about the state of an editable object - which can be an important contextual information - is stored in the application itself and cannot be accessed by an external agent without modifying the application. This is an important drawback when working with unmodified applications.

4. Storing and accessing history. Two aspects of history have to be considered here (Cypher, Kosbie, Maulsby, 1993) : after a bit of user's interaction with the system, a good deal of event-history has accumulated. This is called the **static history**. From this static history, the system can infer templates of repetitive events, or patterns to search for. This process of inferring templates can be very expensive and thus may have to be done **off-line** (for example over night). Then the system has to deal with a part of history which is called the **dynamic history.** It encompasses the more recent user's activity which the system has to process in order to recognize that a patterned activity is beginning : that is, the system has to match the user's actions against part of one of the stored templates. This is obviously an on-line process and so must be very efficient.

5. Inferring templates of repeated activities in the static history. Various machine learning techniques can be used here but basically most PBD systems use symbolic techniques.

6. Matching dynamic history against the stored templates : specific algorithms have been developed to resolve this problem (Cypher, 1991)(Maulsby, 1993)(Frank, 1994). They rely on linear event histories.

7. Interacting with the user to control the completion of the task in a mixed-initiative fashion. There are many important issues here such as how the system informs the user that a possible match has been inferred? How the system completes the task (step-by-step or monolithically with an undo facility) ? How can the user help disambiguate when there are multiple reasonable generalizations? Can the user directly modify the stored procedure, effectively parameterizing it (thus the user demonstrates deleting every message from "Bill", but then can invoke this routine to delete all messages from anyone)? How can the user explicitly invoke a stored procedure ?

8. Completing the repetitive task, that is generating the events to be sent to the application in order to perform the task.

A simple prototype agent was first built and field tests were conducted in our lab over 3 month with 8 users. It was based on flat low-level event histories. Two main things were learned and informed the work which is partly reported here : i) very few repetitive sequences were found in the LLE histories we had recorded, while a human observer watching a user at work could quickly identify repetitive tasks, ii) repetitive tasks should not be confused with habits : when users perform repetitive tasks they are still conscious; they perform an intentional activity; they are still intentional users. If a software capable of automating their repetitive tasks was available, they want control over it.

We will now present the new prototype agent under development within the framework of the ACCEL project, focusing particularly on solutions we have developed to overcome the issues mentioned in points 2 and 3 above : the use of external application models to generate hierarchical event histories.

3. THE ACCEL PROJECT "EPIPHYTE" APPROACH

The research work within the framework of the ACCEL project is supported by French CNRS[1] and CNET[2]. It aims at building a Microsoft Windows-based agent capable of automating repetitive user tasks involving any unmodified applications. This system is qualified "epiphyte" : a term coming from the domain of biology and first coined by S. Giroux (Giroux, 1996) in the domain of information systems to refer to a system which is developed on the surface of an unmodified pre-existing system with the aim to

[1] Centre National de la Recherche Scientifique

[2] Centre National d' Etude des Telecommunications

empower it.

3.1. Capturing low-level events.

Low level events (LLE) - also called device-level-events- are captured through the standard Windows hook mechanism : mouse events, keyboard events and system messages are processed by the ACCEL agent **before** they reach the target application. On-line LLE processing at this stage encompasses **aggregation** and **enrichment** :

- aggregation : several "very low-level events" can be grouped and recorded as one basic user's operation : for example opening the open-file dialog box through the file-menu necessitates several mouse operations (main-menu-selection and then menu-item-selection) generating a set of events whose details are not important in our project : they are grouped and recorded as a "open-open-file-dialog-box-thru-menu" unique operation (a unique LLE). This aggregation is based on an external model of the target application described in the next chapter.
- enrichment : LLE are then enriched with contextual information including the current date and time, the target application name and the currently opened file if any. More contextual information can be added depending on the specifications attached to the current event in the target-application model : for example, if the LLE is a selection in a list-box, the text (name or value) of the selected item is incorporated to the context; if the LLE is a click on a check-box or a radio-button, the final value (ON or OFF) of the control is recorded. If the LLE concerns an untitled widget (a list-box for example) then the system searches for a title in the form of a text-label aligned either horizontally or vertically with the list-box in the same parent window. In the example shown in figure 1 a mouse-click in the combo-box will be recorded as the LLE : "Font.SetSize(10)" which is composed of : i) the basic low-level operation : SELECT 10, ii) enriched with the contextual information : SIZE (found in a text-label aligned with the combo-box and closer to it than any other label), iii) and enriched with the contextual information : FONT (found in the caption of the parent window).

This on-line enrichment mechanism could not apply automatically to every LLE : as we will see it is controlled through the specifications incorporated by a human expert in the target application model. Recording enriched LLEs facilitates the generation of structured HLEs which is the next operation performed by the agent.

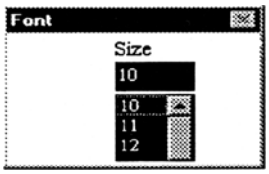

Figure 1 : set font size to 10 example

3.2. Generating high-level events.

Incoming LLEs are then aggregated again and structured to constitute mid-level events. This process is then repeated again to form even higher-level events. With this technique users' actions are recorded in the form of hierarchical event histories. This step is performed on the base of external models of the targeted applications which have to be built by expert users. This solution constitutes the key aspect of our approach.

Advantages of imposing a hierarchical structure on events' history have been exposed by D. Kosbie : hierarchical event histories are closer to user's task structure, they are more robust in the generalization process, they facilitate script matching (point 5 and 6 in previous chapter), they are more intuitive when read-controlled by end users, they are more efficient when played back. Detailed argumentation can be found in (Kosbie, Myers, 1993, 1994).

Unfortunately, the hierarchical structure of events cannot be fully inferred from the read-write patterns of the event handlers which do not provide enough information (Kosbie, Myers, 94). Due to heterogeneity in applications interface, the "enriched" LLE mechanism we have used cannot be fully automated and moreover it does not completely determine the event's hierarchical structure.

For hierarchical event histories to be feasible, there must be a way for applications to generate the event hierarchy. Different architectures have been proposed to allow application designers to incorporate the generation of hierarchical events in the new

applications they build : the hieractors model (Kosbie, Myers, 1994), hierarchical command objects (Myers, Kosbie, 1996), AIDE events (Piernot, Yvon, 1994) or the AppleEvents model.

In the ACCEL project, since we want to work with **unmodified** off-the-shelf applications running under Microsoft Windows, we need an **"external" model** for every involved application : i.e. a complementary knowledge source which is not originally built-in the application but which can be built separately by an expert user to allow the hierarchical structuring of the stream of low-level events. This approach is named "epiphyte" because of the analogy - in the domain of biology - with plants which grow at the surface of other plants without being parasitic. Making a complete model of an application would be a great deal of difficult work; our system is designed to enable an expert user to make a more limited model which allows rich inferencing over a subset of user actions only.

In the context of our project, six hierarchical levels have been considered :

- low-level (or device-level) events (LLE) correspond to the basic operations with the mouse and the keyboard.
- mid-level (or widget-level) events (MLE) correspond to user's interaction with the various controls : list and combo boxes, edit controls, buttons, etc.
- high-level-events (HLE) aggregate several MLEs such as those generated by the use of a modal dialog-box; "Open-file(foo.doc)" is a typical HLE description.
- application-level-events (ALE) aggregate MLEs and HLEs resulting from user's interaction with the same application : EXCEL events and then WORD events for example.
- modeled/specific level (MSLE) is introduced here in response to point 3 mentioned above; a MSLE aggregates a set of events which are generated through the user's use of modeled controls, as opposed to a set of events resulting from user's interaction with application specific unmodeled features. This level doesn't correspond to a structure in user's task but it is mandatory in our approach to differentiate between segments of modeled and un-modeled events.
- session-level events (SLE) aggregate lower events which are generated as long as a user works without resting. SLE are segmented on the basis of time and rhythm statistics.

While hierarchical levels 1 to 4 reflect the inherent structure of "wimb" interfaces, level 5 is project-specific and it is assumed that level 6 corresponds more or less to low level tasks; further field tests will inform on the appropriateness of incorporating new intermediate levels in this hierarchy.

In figure 2, we present an example of a hierarchical history structured on the basis of this model. Two applications (WORDPAD and TOOLBOOK) are involved in a repetitive task : text files (texte1.txt, texte2.txt ...) are successively opened in WORDPAD, text is selected, copied into the clipboard and pasted into TOOLBOOK. Some unmodeled features of WORDPAD have been used before the beginning of the repetitive sub-task. In figure 2, the history structure is partly expanded to show only the full hierarchy of the first modeled WORDPAD event : HLE :OpenFile(texte1.txt) which was started by MLE :OpenDialog(Open), which was itself triggered by LLE :"msg CmdId 7935...". File name (texte1.txt) was extracted from the enriched LLE part of MLE :SetFileName (not detailed in figure 2) and transmitted to its parent HLE.

3.3. Including context : every event descriptor in an application-model includes a section describing the contextual information which is required and a method to access it. When identifying the event, the ACCEL agent obtains the contextual information and incorporates it into the history.

3.4. Storing and accessing history : each time it captures a new LLE, the ACCEL agent selects the model (.HAM file) of the target-application (the application to which the event is destined) and it performs the 6-level hierarchical structuring. It is an on-line process whose result is the recording of a hierarchical event static history (.HEH file).

3.5. Inferring templates. In the ACCEL project, this is a heavy off-line process which is achieved by a

specialized agent : a "light" agent is installed on every end-user's computer; it permanently captures the LLE, includes the context, generates the hierarchical history and transmits the history data (through TCP/IP protocol) to a distant specialized agent which records it. The specialized agent searches for templates of repetitive activities. If some templates are detected, then the specialized agent informs the local agent which can then start to match the dynamic history against those templates. The specialized agent is supervised by an expert user.

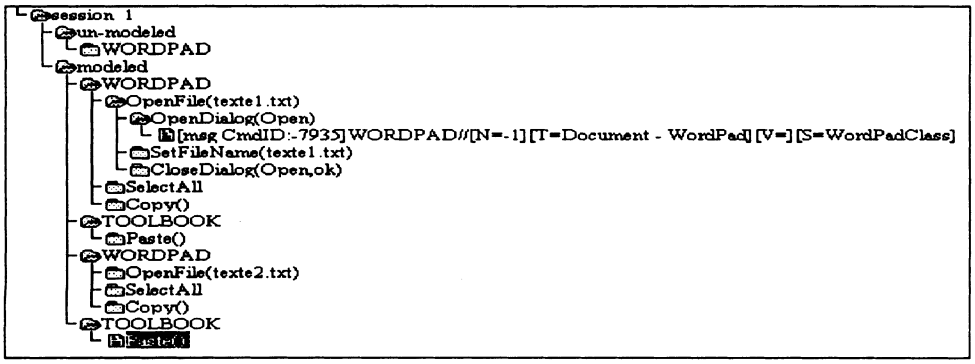

Figure 2 : example of hierarchical history including a repetitive task involving applications WORDPAD and TOOLBOOK (partly expanded).

Advantages of a hierarchical history when searching for templates can be easily seen in the example shown in figure 2 : at ALE level, the pattern is obvious : WORDPAD, TOOLBOOK and so on; it is still obvious at HLE level (thanks to the processing of the file name as a contextual information) and at MLE level (thanks to hiding equivalent LLEs).

3.6. Inferring matches against templates : existing algorithms (mentioned above) have been generalized to apply to hierarchical histories (instead of "flat" histories). These will be described in a forthcoming paper. The remaining steps (interacting with the user and performing the task) are still under development and cannot be presented here.

We will now focus on some aspects of the modeling task which is to be achieved by expert users. We present some features which facilitate and quicken their work particularly in the context of a group of users.

4. EXTERNAL APPLICATION MODELS

4.1. Application models overview

There is one model for each application. Model files are collections of event descriptors. Event description is inspired from the "hieractor model" (Kosbie, 1994) which was developed to express arbitrary high-level application behavior. This model was intended for application designers building new applications, however it is used in our approach to make an a-posteriori modeling of application behavior. The basic idea behind hieractors is that most application behaviors are naturally defined in terms of the events which **start**, **run**, **end** and **abort** them. Here is an example of an HLE event descriptor (from WORDPAD model) :

```
HLE : SetFont (<name>,<style>,<size>,<color>)
        start-when  : MLE  : OpenDialog(font)
        end-when    : MLE  : CloseDialog(font,ok)
        abort-when  : MLE  : CloseDialog(font,cancel)
        run-when    : MLE  : SetFont(<name>)
                  or MLE   : SetStyle (<style>)
                  or MLE   : SetSize(<size>)
                  or MLE   : SetColor(<color>)
                  or MLE   : Null()
```

Every event is described in terms of the events of level below which start, end, run and abort it. An HLE is described by MLEs and an MLE is described by LLEs. For example, we have the following description :

MLE : SetFont(<name>)
start/end-when :
LLE : [click]WORDPAD/<mfn> - WordPad / Font/ [T=&Font :][V=<name>]

The MLE named Nul() corresponds to user's actions (and thus LLEs) which are of no consequence on the behavior described by their parent HLE (for example : using a scrollbar in a list-box). Parameters are transmitted from an MLE to their parent HLE : MLE :SetSize(10) will have for parent : HLE :SetFont(-,-,10,-).

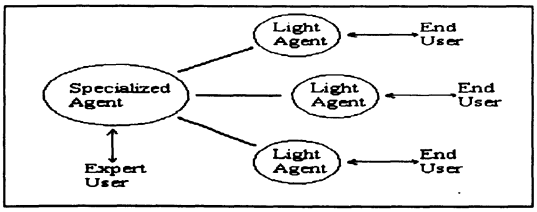

Figure 3 : ACCEL agents architecture.

The MLE OpenDialog(font) can be started by different LLEs depending whether a menu, shortcut or button is used : corresponding events are part of its "start-when LLE :" section.

In the LLE section of the MLE described above, <mfn> is an abstraction of the name of the file currently opened in WORDPAD : it is part of the context.

4.2. Building application models

Using external applications models is a key factor in our approach but the building of such models might also appear as a very time-consuming task. Rather than attempting to improve a fully automated agent based on sophisticated machine-learning techniques, we prefered to appeal to a human expert.

So the current situation involves a group of end-users aided by software agents under control of an expert-user. The expected benefit is an increase of the robustness of the agent's detection module; dependence on a human expert is a cost which can be i) minimized in developing dedicated tools aimed at facilitating and quickening the modeling task, ii) shared among a group of users having similar and time critical tasks. The entire system evolves toward a kind of groupware environment which facilitates the automation of end-users' tasks.

As we have said above, every end-user has a light-agent running permanently. Light agents (LAs) communicate with a specialized agent which runs on the expert-user computer (figure 3). The specialized agent (SA) provides also features to facilitate the modeling task :

- SA collects all LLEs sent by LAs; statistical analysis of all LLE histories gives the list of the most intensely used applications across the group of end-users. Using a simple pattern-matching technique (editing distance) it gives also the list of those functions in the applications which seem to be involved in repetitive activities. In short, statistics indicate those parts of applications which the expert-user will have to model first. The expert user does not work for individuals : when a part of an application get modeled, it benefits to every end-user.
- Enriched LLEs bring default contextual information and the system maintains a list of LLE templates which facilitate extracting of contextual parameters The enrichment mechanism - which was not robust enough in a fully automated approach - becomes very helpful when supervised by a human expert.
- Application models are fully editable while the agent and the application itself are running, so that the application model can be built in a nearly demonstrational mode.
- the specialized agent (SA) provides a debug-mode allowing the expert user to instantly "click-check " in a given application and control that the resulting events have been correctly modeled.

5. RESULTS, FUTURE WORK AND CONCLUSION

Eight ACCEL agents are currently installed in ARDEMI (http ://www.ardemi.fr/), a software company

developing multimedia courseware. Text, sound, graphic editors and authoring tools are to be used in conjunction to achieve multimedia authoring. Users often complain that their tasks involve tedious repetitive operations they cannot automate. Users' actions were permanently recorded there since December 1995, however no result was achieved primarily because the agents reasoned over low-level events. This was our principal motivation for appealing to hierarchical event histories. Generating such histories necessitates that the agents incorporate at least a minimal knowledge of the underlying system and applications.

We have recently undertaken the external (and partial) modeling of applications WORDPAD and TOOLBOOK in order to validate the mechanism which generates hierarchical event histories. First results show a clear increase in power of the inference engine, but the application modeling task is still too cumbersome; we are pursuing ways to simplify this task. More applications need to be modeled before a real field test can be made.

Research efforts are being pursued in two complementary directions, both of which are related to "Programming-by-Demonstration" : the development of the light-agents interface allowing end-users to control the automation of their tasks, and the development of a "modeling-by-demonstration" interface, i.e. an interface allowing expert-users to build application models in a more "demonstrational" fashion.

6. ACKNOWLEDGMENTS

Support for the work described in this paper was provided by CNRS and CNET (ACCEL project N°941B107). The idea of "Hierarchical event histories" was introduced by the second author who is completing a PhD thesis at Carnegie Mellon University.

7. REFERENCES

Crow, D., Smith, B. (1993), The Role of Built-In Knowledge in Adaptive Interface Systems, *proceedings of International Workshop on Intelligent User Interfaces*, ACM press, NewYork.

Cypher, A. (1991), EAGER : Programming Repetitive Tasks by Example, *in proceedings of CHI'91*, ACM press, NewYork.

Cypher, A. (1993), Watch What I do, Programming by Demonstration (ed. A. Cypher), MIT press, Cambridge, Ma., USA

Frank, M., Foley, J. (1994), A Pure Reasoning Engine for Programming by Demonstration, in *proceedings of UIST'94*, ACM press, New York.

Frank, M. R. (1996), Standardizing the Representation of User Tasks, in *Acquisition, Learning & Demonstration : Automating Tasks for Users*, AAAI Symposium, Technical Report SS-96-02, AAAI press, Menlo Park, Ca., USA

Giroux, S., Paquette, G., Pachet, F., Girard, J. (1996), EpiTalk, a Platform for Epiphyte Advisor Systems Dedicated to both Individual and Collaborative Learning, in *proceedings of ITS'96*, Springer-Verlag Lecture Notes in Computer Science, to appear, 1996 (http ://www.laforia.ibp.fr/~fdp/EpiTalk.html).

Kosbie, D., Myers, B. (1993) A System-Wide Macro Facility Based on Aggregate Events : A proposal, in *Watch What I Do* (ed. A. Cypher), MIT press, Cambridge, Ma., USA

Kosbie, D., Myers, B. (1994) Extending Programming By Demonstration With Hierarchical Event Histories, in *proceedings of EWHCI'94* (eds. Blumenthal, Gornostaev, Unger) ICSTI, Moscow.

Maulsby, D. (1994), PhD Thesis, University of Calgary, Canada.

Myers, B., Kosbie, D. (1996), Reusable Hierarchical Command Objects, *in proceedings of CHI'96*, ACM press, NewYork.

Piernot, P., Yvon, M. (1993) The AIDE Project : An Application Independant Demonstrational Environment, in *Watch What I Do* (ed. A. Cypher), MIT press, Cambridge, Ma.

Potter, R. (1993) Just-In-Time Programming, in *Watch What I Do* (ed. Cypher), MIT press, Cambridge, Ma.

Yvon, M., Piernot, P., Cot, N. (1996), Programming by Demonstration : Detect Repetitive Tasks in Telecom Services, in *proceedings of OZCHI'96* (eds. Hasan, Nicastri), Ergonomics Society of Australia, Canberra, Australia.

Maes, P., Wexelbat, A. (1996), Interface agents, CHI'96 tutorial notes, ACM press.