TASK-ORIENTED REPRESENTATION OF ASYNCHRONOUS USER INTERFACES

Antonio C. Siochi
H. Rex Hartson

Department of Computer Science
Virginia Tech
Blacksburg, VA 24061

ABSTRACT
A simple, task-oriented notation for describing user actions in asynchronous user interfaces is introduced. This User Action Notation (UAN) allows the easy association of actions with feedback and system state changes as part of a set of asynchronous interface design techniques, by avoiding the verbosity and potential vagueness of prose. Use within an actual design and implementation project showed the UAN to be expressive, concise, and highly readable because of its simplicity. The task- and user-oriented techniques are naturally asynchronous and a good match for object-oriented implementation. Levels of abstraction are readily applied to allow definition of primitive tasks for sharing and reusability and to allow hiding of details for chunking. The UAN provides a critical articulation point, bridging the gap between the task viewpoint of the behavioral domain and the event-driven nature of the object-oriented implementational domain. The potential for UAN task description analysis may address some of the difficulties in developing asynchronous interfaces.

KEYWORDS: Notation, interface design representation, asynchronous user interfaces, task-orientation, user actions, task description analysis.

INTRODUCTION

The dynamic nature of user interfaces has always been difficult to represent. A good notation reduces the semantic gap between entities in the designer's mind and objects with which that designer must communicate the design. Such entities lie in two interaction metaphors postulated by Hutchins, Hollan and Norman [8] – the conversational and the model world. The first is sequential in nature (as in command line interfaces) while the second is asynchronous (as in direct manipulation interfaces).

There are many techniques for explicitly representing the control flow of sequential dialogue, including BNF and state transition diagrams. However, representational needs for asynchronous interaction are difficult to satisfy with such techniques [10]. For example, how are the user actions required in a direct manipulation [17] interface specified? How are multi-thread and concurrent dialogues represented?

This paper introduces a task-oriented representation technique for asynchronous interaction and focuses on a notation called the User Action Notation (UAN) for representing user action sequences involved in the execution of a task. *The UAN is part of a set of techniques that, taken together, is used to describe interface designs.* This set of techniques has been successfully used to design the interface of a UIMS, where a direct manipulation interface was required. Because the project involved designers, implementors, and evaluators, communication of designs was vitally important. Our initial designs were in prose, supplemented with illustrations. These were time-consuming and varied in expressiveness. The descriptions were often verbose, imprecise, and typically difficult to read, understand, and change.

Our need was for an expressive and concise notation specifying user actions in relation to screen objects, as well as techniques for associating feedback and state changes with those actions. Because the design process is driven by requirements and task analysis, descriptions had to be expressible primarily from the viewpoint of the user, not the computer. Moreover, the interface's asynchronous nature demanded a technique which avoided explicit specification of control flow among tasks. The purpose of the UAN is therefore to communicate with all developer roles the user actions required to perform a task in an asynchronous interface.

The next section summarizes work related to the problem. The rest of the paper presents the UAN by describing portions of a well-known interface, then discusses some UAN characteristics.

RELATED WORK

Models and meta-languages for interfaces exist which can describe the task structure of an interface. They are formal in nature since their creators intended to use them in an analytical fashion, e.g., to predict the quality of the interface given its formal description [15, 11, 16, 2, 12]. Our needs, however, were for a simple notation to describe user actions in a direct manipulation interface – a synthetic rather than analytic tool.

From a synthetic viewpoint, existing work has involved user interface specification based on state transition diagrams, e.g., [9, 18, 19], requiring explicit specification of the interface's control flow. Other work has focused on use of concurrent programming concepts to specify or implement the interface, e.g., [6, 3, 7, 4].

Myers [13] specifies interfaces by demonstration, producing only program code, with no other representation of the interface that conveys its design or that can be analyzed. Olsen [14] generates a standard interface given a set of application functions. Jacob [10] combines state diagrams and object orientation. A mutually asynchronous set of state diagrams represents the interface, avoiding the complexity of a single large diagram. Foley, et al., [5] build a knowledge base consisting of objects, attributes, actions, and pre- and post-conditions on actions which form a declarative description of an interface, from which interfaces are generated. These approaches describe the interface in terms of algorithms the computer must execute in order to interpret user actions. None describe actions from a task orientation.

A SIMPLE EXAMPLE

The description of the task of selecting the trash icon in the Macintosh™ Finder as written in [1] is

1. Position the pointer on the Trash icon...
2. Click the icon by pressing and immediately releasing the mouse button.

The actions required by this task are moving the cursor, locating a particular context on the screen, and operating the mouse button. In the UAN, cursor motion is represented by ~. A screen object's context is indicated by enclosing in square brackets a mnemonic descriptive of that object. Thus

```
~[trash-icon]
```

represents moving the cursor to the context of trash-icon (leaving that context would be written as [trash-icon]~). The context of trash-icon is the "handle" that allows the user to manipulate trash-icon. In this case it is trash-icon itself.

The mouse button is identified as a device, and its possible operations described by representing the mouse button as M and denoting the actions of depressing it by v, and releasing it by ^. Hence

```
Mv^
```

indicates the user action "click the mouse button." The complete specification for the "select trash-icon" actions is

```
~[trash-icon] Mv^
```

Again quoting [1], the description of actions required for the task of moving the trash icon is

1. Position the pointer on the Trash icon.
2. Press and hold the mouse button while you move the mouse.
   When you press the mouse button, you select the icon. As you move the mouse, the pointer moves and drags an outline of the icon and its name along with it...
3. Release the mouse button
   The icon snaps to its new place.

In the UAN, these actions are represented by

```
~[trash-icon] Mv
(~[x,y])*
M^
```

This example introduces several concepts. Reference to the context of a coordinate pair, [x,y], represents some screen location. Grouping of actions is indicated by parentheses. The Kleene star, *, denotes zero or more occurrences of the immediately preceding action. Thus (~[x,y])* means "move the cursor to an arbitrary number of screen locations." These and other concepts will be developed more in the next section.

THE USER ACTION NOTATION

The example above presented parts of the notation to describe user actions necessary for a task. A more complete specification for this portion of the user interface requires more data, e.g., feedback and system states. It is important to associate that information with the appropriate user actions. The following sections describe the UAN and techniques used with it to make such associations.

User Actions

The user actions for a task are represented by a sequence of symbols. Table 1 lists the current UAN symbols.

Feedback

In the example of the task for moving the trash icon, the prose version included descriptions of what happened on the screen in response to user actions. The user actions can be annotated to describe this feedback:

```
~[trash-icon] Mv   trash-icon !
(~[x,y])*          outline of trash-icon follows cursor
M^                 show trash-icon at (x,y)
```

where trash-icon ! means highlight trash-icon. Note that feedback for a user action sequence is written on the same line as that sequence, giving a simple method of indicating the relationship between user actions and feedback. Feedback is often described with prose in combination with notation, since the range of feedback effects is relatively unconstrained compared to the set of user

Macintosh™ is a trademark licensed to Apple Computer, Inc. MacDraw™ is a trademark of Apple Computer, Inc.

actions. In some cases supplemental figures may be required, or may themselves be annotated with user action sequences.

TABLE 1. The User Action Notation Symbols.

User Actions:

~	move the cursor
[sym]	context of sym; "handle" with which user manipulates sym
[x,y]	context of screen location x,y
Xv	depress key X
X^	release key X
()	grouping mechanism
*	Kleene star; indicates zero or more repetitions of previous action
&	concurrency symbol; used to indicate that actions it connects are performed together, but are order independent
;	task interrupt symbol; used to indicate that user may interrupt the current task at this point (the effect of this interrupt is specified as well, otherwise it is undefined, i.e. as though the user never performed the actions)

Feedback:

!	highlight an object
-!	dehighlight an object
!!	same as !, but use an alternate highlight

Conditions of Viability:

condition: action	if condition is true, then action may be performed, e.g., X -! : ~[X]Mv^ means "if X is not highlighted, user may click the mouse on it"

! is used abstractly in the UAN; its definition is kept separate since how to highlight an object is a detail that can interfere with the design process at this level, and is subject to change. The specific meaning of X! can be defined as part of X. For example, a "check box" can be highlighted by placing a check mark in it, a radio button with a bullet, or text by reverse video. If the methods for highlighting are inheritable within a hierarchical structure, consistent highlighting is easy to achieve over a class of interface objects.

System State

Communicating the interface design requires specifying connections to the task semantics. This tells implementors how to interpret user actions. The UAN can be further annotated to indicate semantic connections:

~[trash-icon] Mv	trash-icon !	currentObject = trash-icon
(~[x,y])*	outline of trash-icon follows cursor	
M^	show trash-icon at (x,y)	update location of trash-icon

This indicates that trash-icon becomes the currently selected object when the mouse button is depressed while the cursor is over it. When the mouse button is released, the location of trash-icon is set to the coordinates where the mouse button was released.

Conditions of Viability

The user and the developer need to know under what conditions a particular task is *viable* (i.e., can be performed) at a given point in the interaction. Hence conditions of viability must be included as part of a task description (see pre-conditions in [5]).

For example, a condition of viability for the task of duplicating a file in the Macintosh Finder is that a file must be selected. Assume that the task of selecting from a pull-down menu has been defined as menu(X), where X is the menu item to be selected. The duplication task would thus be described by

Condition of viability: some file is selected

menu(duplicate)	display duplicate-file duplicate-file !	duplicate the selected file

Conditions of viability may be used by designers to prevent errors by disallowing non-viable tasks, e.g., graying out a menu item. Conditions of viability also indicate to the user what has to be done to make a task viable.

Examples

Deleting Multiple Macintosh Files. As an example of using the set of techniques above, consider the task of deleting several files on the Macintosh. Two assumptions are made about the interface design:

1. An object is highlighted if and only if it is selected.
2. A file exists if and only if an icon for it is on the desktop.

The condition of viability is that there are files on the desktop. A UAN task description is shown in Figure 1.

TASK: delete multiple files

USER ACTIONS	FEEDBACK	SYSTEM STATE
('shift-key'v;		
(file-icon -!:		
~[file-icon];		
Mv	file-icon!	add file-icon to selected set
M^;)*;		
'shift-key'^)*;		
file-icon!:		
~[file-icon];		
Mv		
~[x,y]*	hilited icons follow cursor	
~[trash-icon];	trash-icon!	
M^	erase hilited icons, trash-icon!!	mark selected files for deletion

Figure 1. UAN Task Description for Deleting Multiple Files on the Macintosh

TASK: select-menu(x,choice')	RETURNS: choice'	
USER ACTIONS	FEEDBACK	SYSTEM STATE
~[x-menu-bar-choice] Mv	x-menu-bar-choice ! , show x-menu (See figure x.)	
select-pull-down-choice(x,choice')	hide x-menu	Return choice'

Figure 2. UAN Task Description for Selecting a Pull-Down Menu

The file-icon -!: (second line of "User Actions") indicates a condition of viability that applies to a specific step. For this task the file being selected cannot already be selected. This cannot be stated in a task-level condition of viability since during the iteration of selection actions (lines two and three of "User Actions"), some files are selected and some are not. After the mouse click, file-icon is highlighted. In the next pass through the iteration, file-icon refers to some other file since the condition of viability restricts selection to non-highlighted files.

This task can be thought of as two sub-tasks, select-files and delete-selection. These sub-tasks could be defined as separate tasks and referred to here by name.

Selection from a Macintosh Pull-Down Menu. This example shows the definition of two sub-tasks, one (see Figure 2.) for selecting a pull-down menu from a menu bar and another (see Figure 3.) for selecting a choice from the resulting pull-down menu. The second sub-task is referred to in the first task description.

TASK: select-pull-down-choice(x,choice') RETURNS: choice'		
USER ACTIONS	FEEDBACK	SYSTEM STATE
(~[x.choice]	x.choice!	
[x.choice]~)*;	x.choice -!	
~[x.choice']	x.choice'!	
M^	x.choice'!!	Return choice'

Figure 3. UAN Task Description for Selecting a Choice from a Pull-Down Menu

Select-menu(x,choice) can be used in another task description. For example, in the task to open a file, the description might contain select-menu(FILE,OPEN).

TASK ORIENTATION

Task descriptions, such as these examples, are written at a detailed level of abstraction, i.e, in terms of user actions. This level is the *articulation point* between two major activities within the development life cycle: task analysis and design. Because these task descriptions are at once the terminal nodes of the task analysis hierarchy and the beginnings of a user interface design, it is a case where task analysis quite naturally drives the design process.

Use of sub-task references in task descriptions provides a direct means for chunking user actions. The sequence

MvM^

is often thought of as a single mouse click:

Mv^

For many, this chunking is carried to higher levels. For example, the user can assimilate the sequence

~[X] Mv^

into a cognitively atomic action for selecting X. Since the sequence is automatic to the user, lower level actions do not add to the task's cognitive load. The sequence can be defined once and referred to as a sub-task, abstracting away undesirable detail. Many common actions in, for example, the Macintosh Finder soon become automatic for the user. Sub-task references in the UAN then become appropriate for tasks such as invoking a pull-down menu, making a menu choice, or moving an object. In addition, from a design and implementation viewpoint, sub-tasks support reusability and consistency.

ANALYSIS

User actions in a task description can be processed to analyze structure and detect ambiguity and inconsistency. Although the motivation for the UAN was to provide a practical notation for representing asynchronous interaction, it became apparent that there was a need for analyzing task descriptions, since an interface design consists of a set of disjoint task descriptions. The common prefix problem is presented as an illustration.

With our approach to interface design, it is possible to design two tasks such that they share an initial subsequence of actions. Although only one of two such tasks is intended by the user, the computer cannot distinguish between them until actions go beyond their common sequence. This situation, which could lead to ambiguity and conflict in the design, can be detected by task description analysis.

As an illustration, consider these two tasks:

```
--------------------------- [ TASK 1: select-X ]------------------------
~[X]; Mv          X!
M^
```

---------------------------- [TASK 2: move-x]------------------------

~[X]; Mv	X!	
~[x,y]	X follows cursor	
M^	Show X at new location	Update location of X

These tasks have a common prefix, ~[X]; Mv. One explanation for this prefix is that selection is a move of zero distance. Yet this is a case in which task description analysis can identify a potential user problem with the interface. An inadvertent hand movement between Mv and M^ in the select-X task will cause a switch to an unintended task, move-X. MacDraw™ users will recall having moved an object while attempting only to select it.

Another potential problem is unresolvable ambiguity. If these two tasks were designed at different times by different designers, two different highlighting styles might be specified. This presents an impossibility to the implementor because for the prefix

~[X]; Mv

it cannot be decided which task is active and, therefore, which highlighting to use.

Consider further the tasks of selecting and opening a document. The select task, ~[document]Mv^, is a prefix of the open task, ~[document]Mv^v^. Macintosh users may recall the distressing effects of accidentally opening a document while trying to select it.

Other types of analysis might involve user performance metrics, usage strategies, structural consistency, and implementation verification.

SCOPE

Because the UAN is textual, it is not suited to describing screen layouts. When the actual appearance of screen objects must be specified, figures are used to complement UAN descriptions. Because the UAN is user-oriented, it does not address the design of the system structure, nor of the interface objects. Because the UAN is task-oriented, it has a microscopic view of the interface: it does not describe the tasks from a global view, nor does it provide an overview of the system (what the system is).

These characteristics restrict the scope of the UAN to describing user actions in the context of interface objects. Within this scope, it is the simplicity of the notation (few symbols, simple syntax) that makes it useful – precise, concise, expressive, and easy to understand. In addition, its textual nature gives it the power of abstraction, its user-orientation encourages the designer to think from the user's perspective, and its task-orientation allows the designer to focus on specific tasks.

CONCLUSIONS AND FUTURE WORK

We have introduced a task-oriented User Action Notation (UAN) for describing user actions and their associated feedback and system state changes. Our notation is part of a set of asynchronous user interface design techniques that was successfully used to represent the design of the direct manipulation interface of a UIMS. The UAN was found to be concise, expressive and highly readable. It is an articulation point between task analysis and interface design, bridging the gap between the behavioral and constructional domains. Our notation also has potential for task description analysis, thus possibly easing the process of developing asynchronous user interfaces. Future work includes developing more formalism and precision in the notation, and exploring task description analysis.

ACKNOWLEDGEMENTS

The authors wish to thank Deborah Hix, Eric Smith, and the rest of the DMS Project at Virginia Tech for their contributions to the development of this notation. The financial support of the Software Productivity Consortium and the Virginia Center for Innovative Technology is also acknowledged.

REFERENCES

1. Apple Computer, Inc. *Macintosh II Owner's Guide.* Apple Computer, Inc., California, pp. 31 - 33. 1986.

2. Card, S. K., Moran, T. P., & Newell, A. *The Psychology of Human-Computer Interaction,* Lawrence Erlbaum, Associates, New Jersey. 1983.

3. Cardelli, L., & Pike, R. Squeak: a Language for Communicating with Mice. *Computer Graphics* 19,3 (1985), 199 - 204.

4. Flecchia, M. & Bergeron, R. D. Specifying Complex Dialogs in ALGAE. In *Proceedings of CHI+GI 1987* (Toronto, Apr. 5-9). ACM, New York, 1987, pp. 229 - 234.

5. Foley, J., Gibbs, C., Kim, W., & Kovacevic, S. A Knowledge-Based User Interface Management System. In *Proceedings of CHI 1988* (Washington, May 15-19). ACM, New York, 1988, pp. 67 - 72.

6. Green, M. The University of Alberta User Interface Management System. *Computer Graphics* 19,3 (1985), pp. 205 - 213.

7. Hill, R. Event-Response Systems – A Technique for Specifying Multi-Threaded Dialogues. In *Proceedings of CHI+GI 1987* (Toronto, Apr. 5-9). ACM, New York, 1987, pp. 241 - 248.

8. Hutchins, E. L., Hollan, J. D., & Norman, D. A. Direct Manipulation Interfaces. In *User Centered System Design,* D. A. Norman, & S. W. Draper, eds. Lawrence Erlbaum Associates, New Jersey, pp. 87 - 124. 1986.

9. Jacob, R. J. K. An Executable Specification Technique for Describing Human-Computer Interaction. In *Advances in Human-Computer Interaction,* H. R. Hartson, ed. Ablex, New Jersey, pp. 211 - 242. 1985.

10. Jacob, R. J. K. A Specification Language for Direct-Manipulation User Interfaces. *ACM Trans. on Graphics* 5, 4 (1986), 283 - 317.

11. Kieras, D & Polson, P. G. An Approach to the Formal Analysis of User Complexity. *Int. J. Man-Machine Studies,* 22 (1985), 365-394.

12. Moran, T. P. The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems. *Int. J. Man-Machine Studies,* 15 (1981), 3 - 51.

13. Myers, B. Creating Dynamic Interaction Techniques by Demonstration. In *Proceedings of CHI+GI 1987* (Toronto, Apr. 5-9). ACM, New York, 1987, pp. 271 - 278.

14. Olsen, D. R. MIKE: The Menu Interaction Kontrol Environment. *ACM Trans. on Graphics* 5, 4 (1986), 318 - 344.

15. Payne, S. J. & Green, T. R. G. Task-Action Grammars: A Model of the Mental Representation of Task Languages. In *Human-Computer Interaction,* vol. 2. Lawrence Erlbaum Associates, Inc. pp. 93-133. 1986.

16. Reisner, P. Formal Grammar and Human Factors Design of an Interactive Graphics System. *IEEE Trans. on Software Engineering* SE-7, 2 (1981), 229 - 240.

17. Shneiderman, B. Direct Manipulation: A Step Beyond Programming Languages. *IEEE Computer* 16, 8 (1983), 57 - 69.

18. Wasserman, A. & Shewmake, D. The Role of Prototypes in the User Software Engineering (USE) Methodology. In *Advances in Human-Computer Interaction,* H. R. Hartson, ed. Ablex, New Jersey, pp. 191 - 209. 1985.

19. Yunten, T., & Hartson, H. R. A SUPERvisory Methodology And Notation (SUPERMAN) for Human-Computer System Development. In *Advances in Human-Computer Interaction,* H. R. Hartson, ed. Ablex, New Jersey, pp. 243 - 281. 1985.