Interactive End-User Creation Of Workbench Hierarchies Within A Window System

Saul Greenberg and Ian H. Witten

Man-Machine Systems Laboratory
Department of Computer Science
The University of Calgary
2500 University Drive NW
Calgary, Canada T2N 1N4

Abstract — Conventional command-language interfaces to interactive computer systems do not support the problem-solving behaviour of users in a natural way. They supply a single view into a sequential stream of actions, whereas people normally juggle many activities concurrently, switching rapidly from one to another. They provide a wide and flat command structure which is fixed and insensitive to the context of the dialogue. They offer little opportunity for personalization, while people differ radically in what they do and how they prefer to do it.

This paper describes an experimental interface which supports parallel activity through user-defined extensions to a basic command interface. Windows provide multiple independent views into the system. Workbenches supplant the flat command structure. A specialized direct-manipulation editor allows easy creation and maintenance of workbenches by novice and expert alike. Users are encouraged to create their own informational support environments and to alter them as their activity dictates. The scheme complements the normal command interface and utilities can be invoked in whichever way seems most natural.

Introduction

Conventional command languages are widely used to communicate intentions to the interactive top level of a computer operating system. A powerful working environment can be constructed by combining linguistic expression — which provides a terse but simple dialogue — with the flexibility of allowing experienced users to combine commands and create their own. But one of the biggest drawbacks to such interfaces is their sequential nature. They are unsuited to parallel activity such as consulting an on-line manual, looking up references when preparing documents, checking calendars when composing mail, or responding to urgent higher-priority tasks. Sequential dialogue is a poor match to the highly parallel thought processes of people.

In contrast, modern interfaces are window-based and thereby permit parallel activity. Transient pop-up (or pull-down) menus focus the dialogue to the window's current context. Supporting metaphors supply the user with a model for making sense of the interface complexity. For example, the screen is a desktop in the Xerox Star (Smith *et al*, 1982) and Apple Macintosh (Williams, 1984). Applications take advantage of windows to give multiple views into information structures like documents. Similarly, interactive programming environments use an integrated toolbox paradigm for program development, where tools are highly tuned to the application. Smalltalk-80 (Goldberg, 1984) and Interlisp (Teitelman, 1979) are two good examples within the genre. But the structure of each context-dependent view within a window is normally imposed by the system designer. Changing this view is difficult or impossible for the end user, forcing him to conform to a standard which may not fit his current needs.

The present work explores the possibility of supporting parallel activity through user-defined extensions to a basic command interface. A main command window (or several of them) supplies the primary channel of communication with the computer operating system. Independent workbenches can be created — grouped hierarchically if necessary — to support parallel activity. These contain context-dependent pop-up menus to initiate supporting commands. Each workbench is associated with a window which displays any resulting output.

What is novel about this scheme is that context and workbench creation is totally user-defined. A specialized direct-manipulation editor allows easy creation and maintenance of the workbench infrastructure and the pop-up menu contents by novice and experienced users alike. The intent is to encourage users to create their own informational support environments and to alter them as their activity dictates, while retaining the convenience of separate windows with associated pop-up menus.

In order to set the scene before plunging into details of the workbench creation system, the paper begins by discussing aspects of parallelism in user activities, workbench paradigms, and personalization as a path to system flexibility. The subsequent section describes how workbenches can support a command interface. Illustrations and examples demonstrate the workbench in action. The thrust of the paper follows — personalization through end-user creation and editing of the workbench infrastructure. Finally, implementation issues and future directions are discussed.

Parallelism, workbenches and personalization

This section introduces three concepts fundamental to our work: supporting user activity through parallelism; organizing activities with workbenches; and matching user requirements with system offerings via personalization.

Parallelism

Traditional interactive interfaces are highly sequential. Most conventional command languages insist on completing the current activity before the next one can begin. Hierarchical menu structures compound this constraint — before a command can be invoked, the user must navigate from the current location in the hierarchy (the previous activity) to the desired target (the next activity).

Despite the sequential nature of traditional command interfaces, much parallelism arises naturally in interactive dialogues with computers. Documentation must be consulted, references checked, subsidiary programs run, and so on. Parallelism also occurs when the current activity is unexpectedly pre-empted by more urgent tasks, only to be resumed later. A user will generally internalize complex sequences of parallel activity while running tasks sequentially on the computer.

The model of computer-supported parallel activity provides a realistic framework for aiding the user's cognitive thought processes — specifically his short-term and working memory. As Shneiderman (1984) summarizes:

> Short-term memory is used in conjunction with working memory for processing information and problem solving. ... If many facts and decisions are necessary to solve a problem, then short-term and working memory may become overloaded. ... [In addition, these memories] are highly volatile; disruptions cause loss of memory, and delays can require that the memory be refreshed.
>
> Shneiderman, 1984

In practice, sequential dialogues provide little support for storing intermediate results of an action. For example, a programmer requiring documentation on a language construct must leave the editor, invoke the programmer's manual, find and read the required information, and finally recall the editor. The cognitive cost of using this information is high. He must recall his mental location within the program, remember the retrieved information, and apply it. Obviously, a high load is placed on short-term and working memory, resulting in inefficiency due to memory failure.

What is required is interactive support of parallel activity. Certain command languages (such as the Unix C shell; Joy, 1980) support primitive parallelism: tasks (called processes) can be suspended and resumed on demand. This practice provides only limited benefits, for only one viewport into these processes is allowed. Window-based interfaces address the viewport problem very nicely. They are specifically designed to permit parallel activity by providing a variable number of virtual views into a given structure, all mapped to a single physical screen. But windows also system complexity, for the user must now physically manipulate the windows and keep track of which one does what. Constrained systems — such as workbenches — help to minimize confusion by limiting actions in a window to the context of the subsystem running in it.

Workbenches

A second concept fundamental to this paper is that of the *workbench*, a metaphor for categorizing and accessing system utilities. We introduce this topic first by describing the difference between unconstrained and constrained dialogues and then by looking at organizational methods within the latter.

There are several different approaches available for interfacing with interactive systems, such as conventional command-driven interfaces, menus, forms, natural languages and icon-based metaphors (Witten & Greenberg, in press). Although quite different at the presentation level, they can usefully be compared according to the extent to which they constrain the user's actions. An unconstrained system (such as a command interface) allows the user access to all facilities at any time, while a constrained system (such as a menu hierarchy) limits actions to the context of the current subsystem.

General-purpose computers almost invariably eschew contextual dependency between subsystems by making all tools available at any time in the interaction. Because of the virtually infinite range of activities which may be required in unconstrained dialogues, linguistic expression in a command language is the conventional medium for the expert user to convey his intentions to a general-purpose system. Unfortunately, this power is also the system's failing — the user may not desire such a high degree of freedom within what he considers familiar contexts (Thimbleby, 1980).

In order to reduce the cognitive load imposed by a rich set of commands, the utilities which can be invoked by the user are sometimes arranged hierarchically to assist him with his work. Utilities are divided typically into minimally interacting subsystems. This kind of organization (although dealing with passive information rather than active utility programs) is exemplified by Videotex, a generic name for systems used by the general public for information retrieval, tele-shopping and tele-banking. Activities are normally accessed via a menu selection dialogue where each menu category represents progressive refinements of "fields of knowledge" (Tompa, 1982). Another good illustration is the Apple Macintosh (Williams, 1984), in which applications normally interact through an intermediary. For example, including a figure in a text document involves creating the figure in the "Macpaint" application, posting it in an intermediary scrapbook, leaving Macpaint and invoking the "Macwrite" application, and finally retrieving the figure from the scrapbook. Although division of the system into utilities is useful for routine tasks, it is necessary to leave the current context and enter a new subsystem in order to perform unusual actions. The result is a dialogue which is tedious to use unless it happens to be well-tailored to the user's needs while working within a given subsystem. A better paradigm for highly interacting sub-systems is found in the workbench/toolbox metaphor.

Nakatani and Rohrlich (1983) propose a method of integrating links between subsystems by analogy with tools in a workshop. The hierarchy used is a ***tool bin***, which is the entire set of tools, a ***workshop***, which collects similar tools, and a ***workbench*** on which the actual work is done. Toolboxes and workbenches differ from conventional hierarchies in that actions (verbs) are categorized instead of objects (nouns). Workbenches contain tools appropriate for acting on the object(s). Example objects may be the complete system environment, a collection of files, or a single file. Example workbenches are a viewing workbench for editing, displaying and listing files, a language workbench for compiling and debugging programs, and a manual workbench for accessing the programmer's manual.

A recent project at Bell Laboratories, called *Menunix*, shows how an extensive and flexible operating system interface can be implemented with menus (Perlman, 1984). One consequence of menu access to Unix programs is that the vast selection of utilities must be structured somehow into reasonably small subsets; otherwise the menu would become unmanageable. System programs are assigned to workbenches — arranged hierarchically — and a *program menu* displays brief (half-line) descriptions of the programs in the current workbench. When a program menu entry is selected, arguments are requested and the program is executed. In order to implement the hierarchy, an entry in a workbench may point to another workbench (in the same way that an entry in a directory may point to another directory in the file hierarchy). Selecting one of these entries will replace the current program menu accordingly. Menunix provides integration by using the workbench metaphor to unify the user's view of the system.

Menunix does not seem to have gained wide acceptance, perhaps because it uses sequential access to drive a paradigm which is best viewed in parallel. But most modern interfaces use windows to provide parallel views, and transient pop-up menus to access generic functions. In sharp contrast to linguistic commands, fixed menus constrain the user to a small set of operations in each context. By mapping workbenches to windows (the working area) and tools to the transient menus tied to the window, a parallel workbench system is created.

Another problem of the workbench paradigm is matching the collection of tools with the user's requirements. We explore this in the next section.

Personalization

The ability to create virtual views into a workbench structure through the use of windows addresses parallel activity very nicely. So does the use of pop-up menus which remind the user what can be done in the current context and which offer a simple way of invoking relevant procedures. The more clearly the user's activity can be delineated, the more probable it is that a pre-defined scheme will be able to support it unobtrusively. In the case of everyday interaction with a general-purpose computer system, activity is unpredictable and varies enormously from one user to another. In this case a fixed scheme is unlikely to provide a good match to the user's requirements.

The arrangement and selection of workbenches is normally imposed by the system designer, and so users are forced to adopt a standardized view of the system (or suffer). For example, a typical mail workbench may contain tools to send, receive, filter and edit mail. This is insufficient for the user who receives encrypted mail and requires access to a decryption tool. High overhead is involved if this tool belongs to a different workbench. In contrast, while some specialist systems (such as the Lisp machines and Xerox Dolphin) supply pop-up menus for context-dependent commands, they do permit the user to create his own windows and choose which operations to associate with their pop-up menus. However, only the highly skilled are likely to attempt this, and then infrequently and not as part of daily activity in coping with non-routine demands.

While the range of tools available may be very large, each individual user generally employs only a small subset of them (Hanson *et al*, 1984; Greenberg, 1984). Techniques of *modeling* the user promise to reduce his cognitive load by personalizing the environment to make available (or at least give preference to) only those commands which he is likely to invoke, and structuring them in a way which corresponds to the way he perceives the system. At its most basic, user modeling takes the form of a system designer interacting with the target population to adjust the system to fit the typical user's current needs (Eason & Damodaran, 1979). Unfortunately, users cannot accurately be viewed as "typical" in the normal case of a highly heterogeneous community (Rich, 1983). Edmonds (1982) suggests that our knowledge of human behavior is inadequate to portray correctly the typical user — especially one whose need will change over time. An alternative personalized outlook shifts the design focus towards a collection of models of individual users.

Explicit personalization, where the user explicitly alters his working environment to suit himself, offers the greatest potential for personalization — at least in the short to medium term (Greenberg, 1984). The ability to tailor one's working environment seems to be an important attribute of what are termed "user-friendly" systems. Examples are the use of a profile statement which is interpreted automatically on login, control over system parameters (such as the prompt), making appropriate entries in abbreviation files, and placing utility programs so that they will automatically supersede the standard system utilities when invoked (as supplied in the Unix C-shell; see Joy, 1982). Unfortunately the possibility of explicit personalization raises its own problems. In order for the user to construct the model, he usually needs fairly advanced knowledge of the system and its capabilities, possibly negating any benefits. Although inferential *automatic* modeling may offer a solution to this, it is very difficult to do for the general systems we are taking about.

With general-purpose workbenches, there is an excellent case for extensive explicit personalization so that the user can set up his own options in pop-up menus and specify their effect in terms of invoking system commands, creating new windows, and so on. In order to reduce set-up overhead, and to encourage frequent re-modeling by novice and expert alike to reflect changing task requirements, it is essential that easy-to-use tools be provided for constructing and maintaining the models.

Workbenches in a conventional command-driven interface

The rest of this paper describes the *workbench creation system* (WCS), which supports command-driven interactions with the Unix operating system through a window interface, the Jade window manager (Unger *et al*, 1984). Print on paper is a poor medium for explaining highly interactive systems like WCS: we use simulated snapshots of the workstation screen to help convey the nature of the interface.

The user pursues his primary activity through a command window (several parallel command windows may be created if desired). He is supported by an infrastructure of specialized windows (workbenches) which can be used to provide relevant information (Figure 1). This is much more than a set of views into a passive text database, for arbitrary Unix commands, shell scripts or local programs can be associated with pop-up menu items. In fact, the user can define his own hierarchy of workbenches in the specialized windows, thereby creating an individualized classification of Unix commands and non-standard utility programs.

Each workbench comprises a window and associated pop-up menu. Entries on the pop-up menu may be of three types. The first invokes a Unix command which produces output only. Arguments may either be specified when the menu item is created or requested when it is picked. For example, a "Manual" workbench may provide two commands, entitled "title" and "keyword" (Figure 1). Both have one argument, which is solicited in the workbench window by the prompt "Title of manual entry:" for the first command and "Keyword contained in manual entry:" for the second. The first invokes the Unix *man <argument>* command, and the second invokes *man -k <argument>* (which is Unix *man*'s way of specifying a keyword search). In either case the output of the *man* command appears in the workbench window. One enhancement of this arrangement is the automatic insertion of the output into a simple cut and paste editor, allowing scrolling, altering, and saving of results. Additionally, as the window may be of arbitrary size, the workbench and its contents may be shrunk and recalled later.

The second type of pop-up menu item invokes an interactive Unix command. If no process is running in the workbench, the sub-system invoked through the menu will run in its window. Otherwise, the workbench is duplicated in a new window and the command executed there.

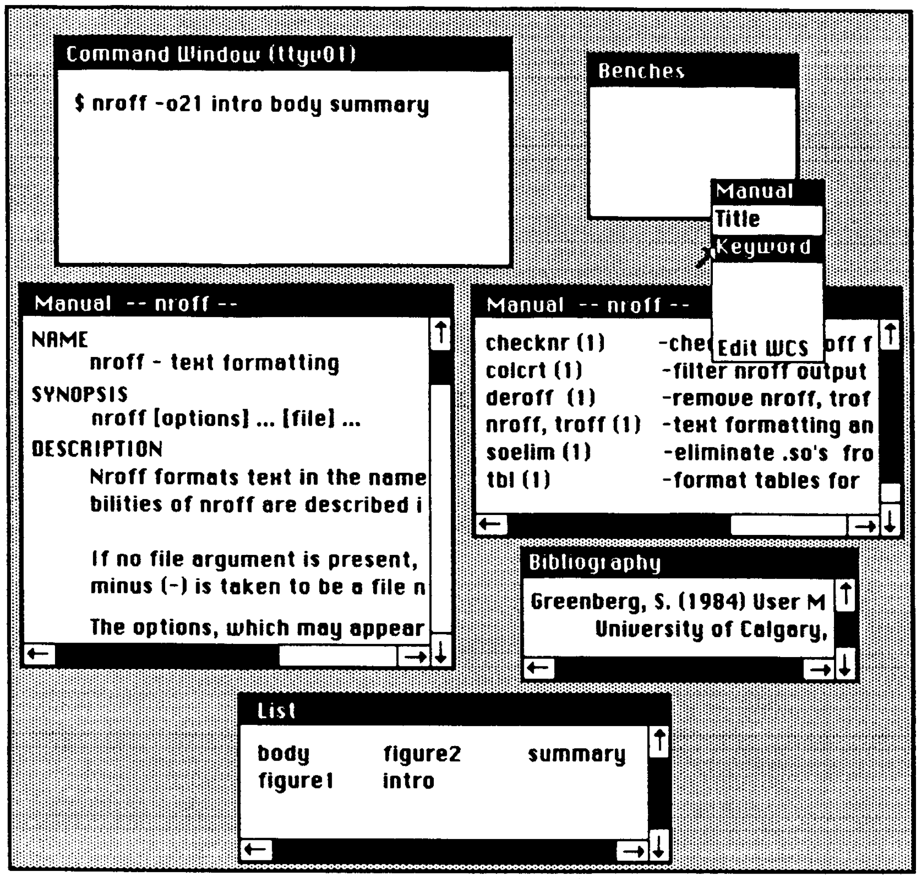
The final type of pop-up menu item enters another workbench. This creates a new window on the screen and associates the appropriate pop-up menu with it. The old workbench window remains on the screen and can still be accessed, along with its pop-up menu. As a consequence, the user may have entry points into many different parts of the workbench hierarchy.

The WCS operates in the environment of the command window that spawned it, allowing each workbench to share global notions such as the current working directory. For example, a workbench that lists files will list those of the current directory even after it has been changed in the command window.

The example in the Figure represents a scenario where the user has created workbenches to help him compose a command in the "command" window for putting text files through nroff, a text formatter. Of the two "manual" workbenches, the first invoked a keyword search for the title of a manual entry. The second was used to display the entry. A "list" workbench listed the files in the current directory, and a "bibliography" bench allowed perusal of references (perhaps used in the text files). In this illustration, the "benches" window is the root of the workbench hierarchy.

End-user creation of workbench hierarchies

Central to the WCS is the method of creating and altering the support system of workbenches. Without the ability to personalize it the interface would have limited novelty, being simply a way of allowing users to navigate through a predetermined hierarchy of utilities. Although superior to non-windowing workbench interfaces such as Menunix (Perlman, 1984) — in that, once accessed, a workbench remains immediately accessible as a window on the screen until it is explicitly deleted by the user — the workbench paradigm contains no fundamentally new ideas.



```
$ nroff -o21 intro body summary
```

```
NAME
       nroff - text formatting
SYNOPSIS
       nroff [options] ... [file] ...
DESCRIPTION
       Nroff formats text in the name
       bilities of nroff are described i

       If no file argument is present,
       minus (-) is taken to be a file n

       The options, which may appear
```

```
checknr (1)      -che
colcrt (1)       -filter nroff output
deroff (1)       -remove nroff, trof
nroff, troff (1) -text formatting an
soelim (1)       -eliminate .so's fro
tbl (1)          -format tables for
```

```
Greenberg, S. (1984) User M
        University of Calgary,
```

```
body       figure2      summary
figure1    intro
```

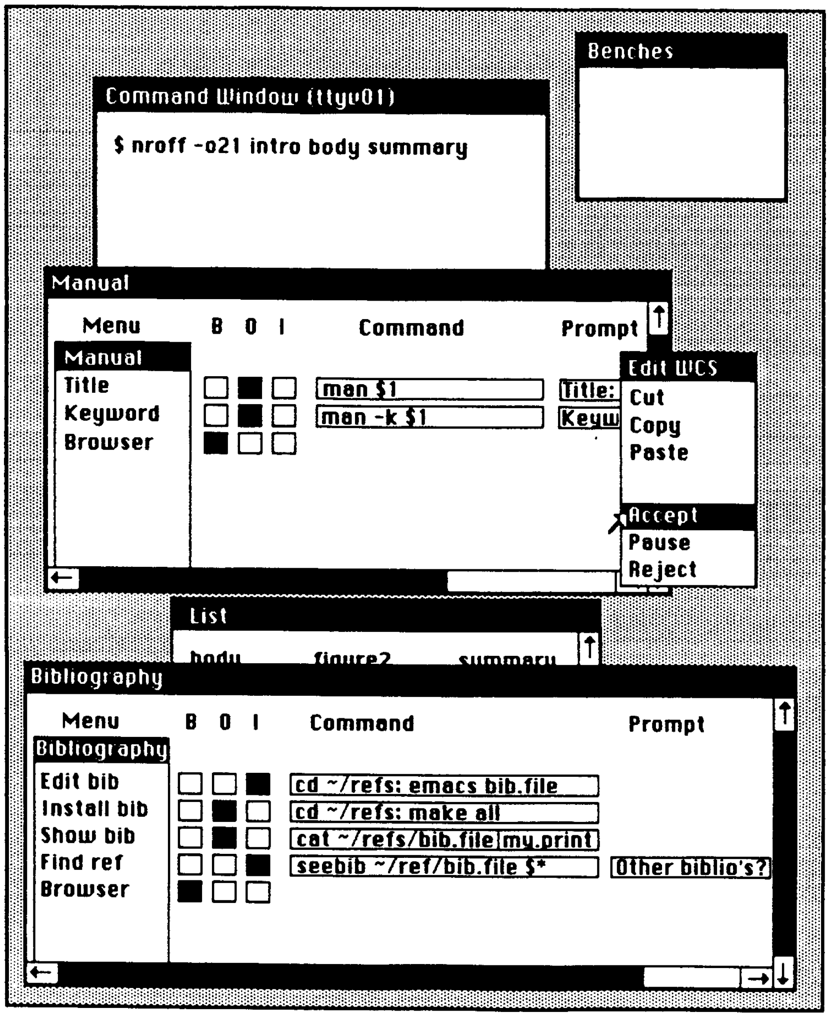
Figure 1: Workbenches as a support structure

But the inclusion of an end-user creation/maintenance system provides an interesting medium in which to explore explicit user personalization in a rather sophisticated interface. It is essential to success that modification be quick and easy, for if not, novice users will be denied access to a tool which should make work much easier for them, and expert users will not alter the support structure to reflect changing requirements. These drawbacks are evident in most existing explicit user personalization systems. A direct-manipulation interface† reduces the usually complex task of setting up windows and pop-up menu contents into a matter of form-filling (Figure 2).

The user defines workbenches in the first place by filling out and editing a simple form. This specifies the name of the workbench and the name and action associated with each pop-up menu item. Actions are of three types: the first executes a Unix utility or user-defined program which produces output only, the second executes interactive utilities, and the last enters a new workbench. In the first and second case, the Unix command is specified along with the name of the menu item. It may contain arguments (given in the usual Unix shell notation), in which case a prompt must be entered to solicit each one when the menu item is picked. In the last case, when the menu item specifies another workbench, that workbench must also be defined in the same way.

Along with each menu item is associated a HELP window which the user can see at any time by pressing a "HELP" button on the mouse. By default, the WCS displays the Unix command associated with the selection. However, the user is invited to specify the contents of each help window when creating the workbench. This encourages workbenches to be documented when they are created, removing one of the most severe drawbacks to

†A good overview of direct manipulation is found in Shneiderman, 1983



```
$ nroff -o21 intro body summary
```

Menu	B	O	I	Command	Prompt
Manual					
Title	□	■	□	man $1	Title:
Keyword	□	■	□	man -k $1	Keyw
Browser	■	□	□		

Menu	B	O	I	Command	Prompt
Bibliography					
Edit bib	□	□	■	cd ~/refs: emacs bib.file	
Install bib	□	■	□	cd ~/refs: make all	
Show bib	□	■	□	cat ~/refs/bib.file my.print	
Find ref	□	□	■	seebib ~/ref/bib.file $*	Other biblio's?
Browser	■	□	□		

Figure 2: The workbench editor

explicit personalization — that a user becomes confused and disoriented when faced with another's model (for example, when helping him on a terminal). Although we do not check that the user-supplied HELP information is accurate, the fact that it must be provided before a workbench can be completed should encourage sensible use.

Direct manipulation is used as far as possible in the WCS. A user edits or expands an existing workbench by traversing the workbench hierarchy in the normal way and then selecting *edit* from the pop-up menu. The window becomes a workbench editor, with a concrete view of its pop-up menu appearing inside (Figure 2). Attributes of each menu item are listed in fields next to the item. These attributes and all other workbench entries may be altered at will. At any point, the user may choose to accept, reject, or pause the editing session.

The direct manipulation paradigm is also used to edit the hierarchy (Figure 2). New workbenches are added by creating a new menu item in the parent, selecting that entry, and editing the empty workbench. Removing an entry point deletes the workbench. Attributes are duplicated between windows through cutting and pasting. For example, copying the name of a menu item from one workbench to another duplicates all attributes associated with that entry. If the item was an entry into a child workbench, the complete hierarchy is duplicated.

Figure 2 shows a user modifying a workbench infrastructure. He is editing two benches, "Manual" and "Bibliography". Within the editing view, selection boxes indicate the menu entry's type (B for benches, I for interactive and O for output). For example, the "Edit bib" entry is a program which would be run interactively in the Bibliography workbench. Attribute fields are displayed only if appropriate, i.e. the "Prompt" field is visible only if a variable argument was specified in the "Command" area. In the Figure, the user has copied the "Browser" item from the Bibliography menu to the Manual menu. As this item is actually an entry to a workbench sub-tree, the complete sub-tree is duplicated.

The workbench creation system sketched above is an effective user interface prototyping scheme for windowed interfaces with pop-up menus. With it, one can easily and interactively build a window interface for a command-based interactive program. For example, a Unix software tool with a plethora of switches to generate different variants of its behavior can in a matter of minutes be given a smooth, window-based, interface which is controlled by pop-up menus, each having pertinent context-dependent help.

Implementing the WCS

The workbench conceptual model

From the user's perspective, the WCS has two conceptually different components. The first is the *workbench access system*, which allows users to interact with a pre-defined *workbench model*. The second component is the *workbench editor*, which alters the model (Figure 3). The rest of this discussion will be concerned with the implementation details of those components which are kept hidden from the user.

The workbench model is defined by a recursive intermediate script language. In BNF form, this language is defined in Figure 4a. In the Figure, ItemName, Help, UnixCommand and Prompt are character strings. Of special interest is the recursive appearance of WindowType within the BenchItem definition, for it permits the user to construct a tree structure of workbenches. Figure 4b illustrates one realization of the script language using a

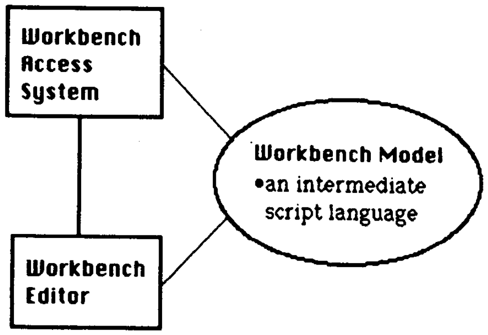


Figure 3: The conceptual model of the WCS

lisp-like syntax. Except for the help window contents, which are abbreviated to *<help>* for brevity, the example completely defines the workbench hierarchy of Figure 4c.

Each workbench carries with it its location in the model. For example, the "Manual" workbench of Figure 4c points to its corresponding definition in Figure 4b. Sub-trees may be include in this description as they are nested in the workbench definition. Editing a workbench is simply a matter of altering a localized view of the model.

The workbench implementation model

The workbench creation system is conceptually simple in design, consisting of five fundamental processes: a unique *environment process*, workbench *controllers* for each bench, *program servers* to execute programs in a window, *cut and paste editors* for filtering program output, and *workbench editors* for editing the workbenches.

Consider the model in Figure 5. After invocation from the user's command window, two processes are spawned. The first is the environment process, which maintains a common working environment between all benches and the original command window, and which creates program servers upon request. The second process is a workbench controller, which is responsible for activating the four types of user requests.

The first asks the workbench to create a new child bench. The controller for Workbench 1 in the Figure is shown spawning a child subtree. A new controller is recursively created, and is given a pointer into the workbench model which describes its activities.

Running interactive programs involves requesting a program server from the environment process (Workbench 2 in the Figure). Cooperation between server and the workbench controller allows program execution in the appropriate window. Non-interactive programs invoke similar actions, except output is filtered to a cut and paste editor spawned by the controller (Workbench 3).

The final task of the controller — workbench editing — invokes a special editor process whose effect is to alter the intermediate script language of the workbench (Workbench 4).

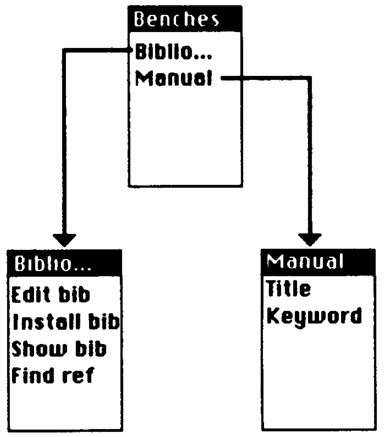
Figure 5 does not show all the details of process interactions. For example, the workbench editor does not edit the intermediate script language directly, for it must go through a master script editor which is responsible for maintaining a consistent workbench

```
<Script>            ::= <WindowType>
<WindowType>        ::= <WindowName>, <Help> {, <MenuType>}
<MenuType>          ::= <MenuName>, <Help>, <ItemType> {, <ItemType>}
<ItemType>          ::= <OutputItem> | <InteractiveItem> | <BenchItem> | <>
<OutputItem>        ::= <ItemName>, <Help>, O, <UnixCommand>, <Prompt>
<InteractiveItem>   ::= <ItemName>, <Help>, I, <UnixCommand>, <Prompt>
<BenchItem>         ::= <ItemName>, <Help>, B, <WindowType>
```

4a: Script language (BNF form)

```
("Benches", <help>,
|   ("Benches", <help>,
    |   ("Bibliography", <help>, B,
|       |   ("Bibliography", <help>,
    |           ("Bibliography", <help>,
|       |               ("Edit bib", <help>, I, "cd ~ /refs; emacs bib.file", "")
    |                   ("Install bib", <help>, O, "cd ~ /refs; make all", "")
|       |               ("Show bib", <help>, O, "cat ~ /refs/bib.file | my.print", "")
    |                   ("Find ref", <help>, I,"seebib ~ /refs/bib.file $*", "Other biblios?"))))
|    __ ("Manual", <help>, B,
    |       ("Manual", <help>,
|       |       ("Manual", <help>,
    |       |       ("Title", <help>, O, "man $1", "Title:")
|       |       |   ("Keyword", O, "man -k $1", "Keyword:"))))))
    |       |       |
|       |       |   menu items, 2nd level
    |       |   menu names, 2nd level
|       |   workbench names, 2nd level
    |   menu items, 1st level
|   menu names, 1st level
workbench names, 1st level
```

4b: An example script



4c: A simple workbench hierarchy

Figure 4: The intermediate script language and an example

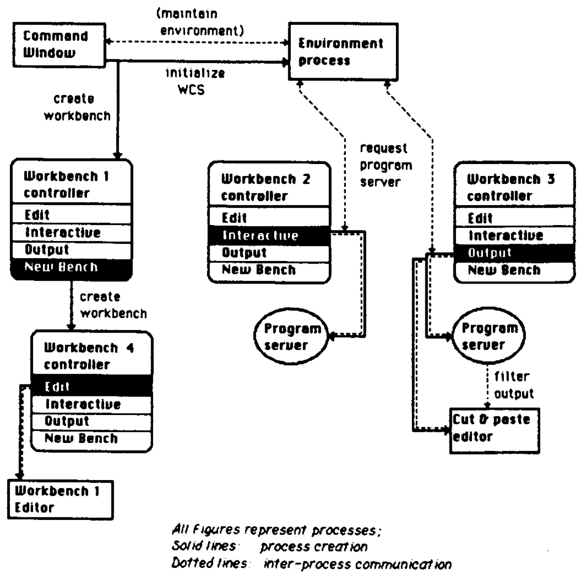


Figure 5: The implementation model of the WCS

model. Another hidden detail is that the environment process knows the status of all workbenches. Upon request, it may record the location and activities of all benches on the screen, allowing the user to "save" his workshop between sessions.

Difficulties in implementation

Although the workbench model described above is simple in design, implementation is difficult, partly as a result of the Unix operating system. The first problem stems from the number of processes created for a large number of active workbenches, for Unix restricts the number of processes a user is allowed. Our implementation multiplexes all workbench controllers into a single process. Although messy in practice, it is totally transparent to the user.

The second difficulty arises from the distributed nature of our work. Our model allows processes to run on (or between) workstations and a Unix host (a Vax 11/780). For example, the cut and paste editor and the workbench editor reside on a workstation. Program servers are actually two communicating processes running on the host and the user's workstation. The execution location of an individual program is determined by the program itself. Fortunately, the WCS is built upon the Jade inter-process communication protocol, which elegantly handles the distributed nature of our system (Unger *et al*, 1984).

But the major problem comes from the nature of the WCS, which attempts to amalgamate two fundamentally different paradigms — sequential command dialogues and cooperating workbenches. In practice, many extra processes must be created to support parallelism due to the sequential nature of Unix and its underlying assumptions. For example, the environment process of Figure 5 is necessary because maintaining a common working directory between workbenches is difficult.

The marriage of the WCS to a modern interactive programming environment would not only minimize these problems but would also enhance its capabilities. The strong notion of objects in Smalltalk (Goldberg, 1984) would allow binding of workbenches to individual objects, such as documents, files or graphical entities. The extensible nature of Lisp environments (Teitelman, 1979) would eliminate naturally many of the extra processes needed in our implementation.

The WCS has been partially implemented, and should be completed by the summer of 1985. A prototype of the workbench access system was recently released, and the workbench editor is now under development. Currently, users may edit the intermediate script language for personalization. That they are willing to edit an internal language shows high promise for the completed system.

Summary

The workbench creation system represents a synthesis of many ideas. General purpose command languages, parallelism, workbenches for constraining activities, and personalization are all combined in a natural interface to offer the user flexibility in performing tasks on a computer. We have shown how this paradigm can be implemented on top of a conventional interface.

The thrust of our work is to provide a parallel workbench structure which may be created and modified by the end user. It is this aspect of personalization that the workbench creation system exemplifies.

Most systems allowing personalization do so at great cost to the user in learning and modification time. Traditionally, the user minimizes this cost by avoiding personalization altogether or by copying and modifying the profile of the local "expert" (Greenberg, 1984). As the WCS is specifically designed to ease the user's burden, it should provide a good vehicle for studying how users can effectively personalize their environment.

The union of *explicit* and *automatic* personalization suggests exciting new developments. We foresee an *interface monitor* which keeps track of user activities and offers potentially useful workbench configurations on request. When combined with a knowledge base, the monitor may infer workbenches from minimal user actions, possibly through stereotyping with existing models (Rich, 1983). One consequence is rapid development of workbenches suitable for transient user actions.

We envision a complete system which integrates all aspects of modeling. Perhaps first time users will have a default workbench structure to begin with (created by the designer through discussion with users and through analysis of their needs). It is a simple learning progression to go from modifying individual workbenches, to adding new ones, and finally modifying or creating new support infrastructures. The exciting possibility of workbenches modifying themselves (possibly through consultation with the user) would go even further to facilitate effective use of the WCS.

Acknowledgement. We would like to give special thanks to the talented Dan Freedman, who is busily turning our ideas into reality. This research is supported by the Natural Sciences and Engineering Research Council of Canada.

Bibliography

Eason, K.D. and Damodaran, L (1979) "Design procedures for user involvement and user support" *Infotech - Man Computer Communications*, London.

Edmonds, E. (1982) "The man-computer interface: a note on concepts and design" *Int J Man-Machine Studies, 16* (3) 231-236, April.

Goldberg, A. (1984) "The influence of an object-oriented language on the programming environment" in *Interactive programming environments*, edited by Barstow, Shrobe and Sandewall, pp 141-174. McGraw-Hill Book Company, New York.

Greenberg, S. (1984) "User modeling in interactive computer systems" MSc Thesis, Department of Computer Science, University of Calgary.

Hanson, S.J., Kraut, R.E., and Farber, J.M. (1984) "Interface design and multivariate analysis of UNIX command use" *ACM Transactions on Office Information Systems, 2* (1), March.

Joy, W. (1980) "An introduction to the C shell" in *Unix Programmer's Manual, Seventh Edition, Volume 2c.* University of California, Berkeley, California, November.

Nakatani, L.H. and Rohrlich, J.A. (1983) "Soft machines: A philosophy of user-computer interface design" *Proceedings of human factors in computer systems*, Boston, Mass., December 12-15.

Perlman, G. (1984) "Natural artificial languages: low-level processes" *Int J Man-Machine Studies, 20* (4) 373-419, April.

Rich, E. (1983) "Users are individuals: individualizing user models" *Int J Man-Machine Studies, 18* (3) 199-214, March.

Shneiderman, B. (1983) "Direct manipulation: a step beyond programming languages" *IEEE Computer, 16* (8) 57-69, August.

Shneiderman, B. (1984) "Response time and display rate in human performance with computers" *Computing Surveys, 16* (3) 265-285, September.

Smith, D.C., Irby, C., Kimball, R., Verplank, B., and Harslem, E. (1982) "Designing the Star user interface" *Byte, 7* (4) 242-282, April.

Teitelman, W. (1979) "A display oriented programmer's assistant" *Int J Man-Machine Studies, 1* (2) 157-187, March.

Thimbleby, H. (1980) "Dialogue determination" *Int J Man-Machine Studies, 13* (3) 295-304, October.

Tompa, F.W. (1982) "Retrieving data through Telidon" *CIPS Session 82 conference*, Saskatoon, April.

Unger, B., Birtwistle, G., Cleary, J., Hill, D., Lomow, G., Neal, R., Peterson, M., Witten, I.H., and Wyvill, B. (1984) "Jade: a simulation and software prototyping environment" *Proc Conference on Simulation in Strongly Typed Languages*, San Diego, California, February.

Williams, G. (1984) "The Apple Macintosh computer" *Byte, 9* (2) 30-54, February.

Witten, I.H. and Greenberg, S. (in press) "User interfaces for office systems" in *Oxford Surveys in Information Technology*, edited by P. Zorkoczy. Oxford University Press, Also available as Research Report 84/161/19, Department of Computer Science, University of Calgary.