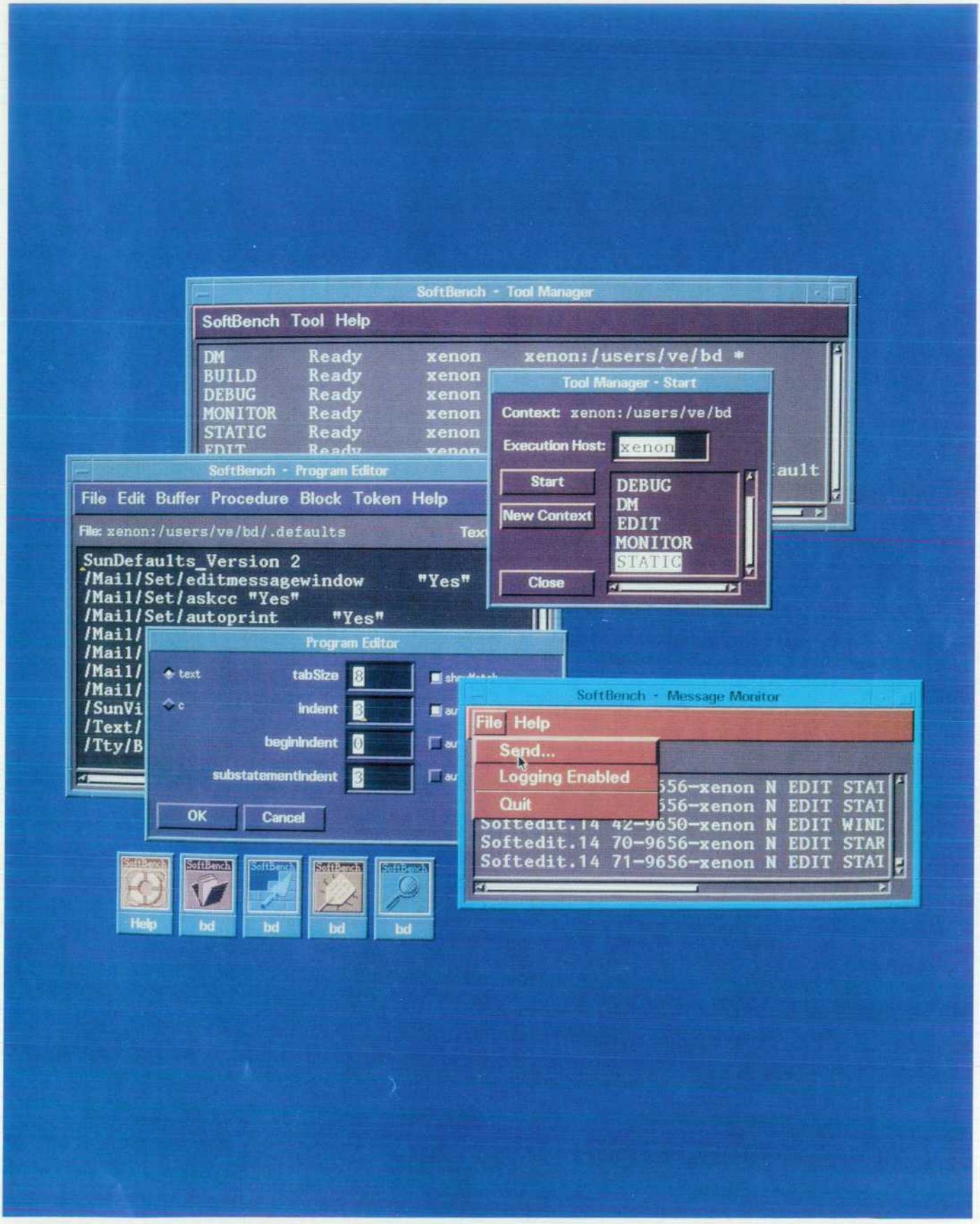
HEWLETT-PACKARD
JOURNAL

JUNE 1990



HEWLETT-PACKARD JOURNAL

June 1990 Volume 41 • Number 3

Articles

Research Report

Departments

The **Hewlett-Packard Journal** is published bimonthly by the Hewlett-Packard Company to recognize technical contributions made by Hewlett-Packard (HP) personnel. While the information found in this publication is believed to be accurate, the Hewlett-Packard Company makes no warranties, express or implied, as to the accuracy or reliability of such information. The Hewlett-Packard Company disclaims all warranties of merchantability and fitness for a particular purpose and all obligations and liabilities for damages, including but not limited to indirect, special, or consequential damages, attorney's and expert's fees, and court costs, arising out of or in connection with this publication.

Subscriptions: The Hewlett-Packard Journal is distributed free of charge to HP research, design, and manufacturing engineering personnel, as well as to qualified non-HP individuals, libraries, and educational institutions. Please address subscription or change of address requests on printed letterhead (or include a business card) to the HP address on the back cover that is closest to you. When submitting a change of address, please include your zip or postal code and a copy of your old label.

Submissions: Although articles in the Hewlett-Packard Journal are primarily authored by HP employees, articles from non-HP authors dealing with HP-related research or solutions to technical problems made possible by using HP equipment are also considered for publication. Please contact the Editor before submitting such articles. Also, the Hewlett-Packard Journal encourages technical discussions of the topics presented in recent articles and may publish letters expected to be of interest to readers. Letters should be brief, and are subject to editing by HP.

Please address inquiries, submissions, and requests to: Editor, Hewlett-Packard Journal, 3200 Hillview Avenue, Palo Alto, CA 94304, U.S.A.

In this Issue

We didn't plan it that way, but two groups of articles in this issue deal with the design of software to make user interaction with computers simpler, more consistent, more intuitive, more standard, more foolproof. One group of articles describes a standard graphical user interface and the other describes an environment that provides a consistent user interface for software development tools. Since we didn't do anything special to get these two packages into the same issue, their simultaneous appearance—close on the heels of the HP NewWave Office—is simply further evidence of the attention that user friendliness is receiving in the R&D community.

The graphical user interface is called OSF/Motif. It's the first product of the Open Software Foundation, an international organization created by leading computer companies to promote open software standards—standards that make it easier for users to mix and match applications and computers from different suppliers. Based on technology from Hewlett-Packard and Digital Equipment Corporation, OSF/Motif provides consistent behavior between personal computers and engineering workstations and an enhanced 3D appearance that makes buttons look as if they've been pressed when the user selects them. HP's implementation of the OSF/Motif graphical user interface is described in the three articles on pages 6 to 35. The first article discusses HP OSF/Motif concepts and external behavior. The other two articles discuss the two main HP OSF/Motif components: the HP OSF/Motif window manager and the HP OSF/Motif widgets. The widget library is a programmer's toolkit that makes it easy to develop applications that have the OSF/Motif graphical user interface.

The software development environment is called the HP SoftBench environment. It provides software developers with a unified, consistent interface to the computer-aided software engineering (CASE) tools they most often need. Tools included in the HP SoftBench product are a program editor, a static analyzer, a program debugger, a program builder, and electronic mail. Using an HP SoftBench component called the HP Encapsulator, other tools can be added to the environment and HP SoftBench tools can be replaced with other tools. Provided that they meet certain minimum requirements, encapsulated tools don't have to be modified at all. The HP SoftBench environment is designed to support development teams in distributed computing environments. It can be customized to conform to local organizational, team, and personal processes, and any tool can execute on any computer in the user's network. The HP SoftBench user interface follows the OSF/Motif appearance and behavior. (Because of the small size of the screen images shown in the articles, the 3D appearance isn't apparent there, but you can see it on the cover.) The HP SoftBench tool integration architecture is described in the article on page 36. The HP SoftBench CASE tools are explained in the article on page 48, and the HP Encapsulator is the subject of the article on page 59.

"Hyphenated techniques" is a name chemists use to refer to certain combinations of analytical techniques. One of these is liquid chromatography/mass spectrometry, or LC/MS. The constituents of an unknown sample mixture are separated by a liquid chromatograph, and a mass spectrometer is used to identify and measure the concentration of each constituent. It's not entirely straightforward. An interface is needed between the two instruments to control the flow rate and remove the solvent that carries the unknown through the chromatograph. While several interface techniques have been tried, none has been completely satisfactory. However, the relatively new particle beam interface looks good. It is applicable to a wide range of compounds and produces spectra that have high information content. The article on page 69 introduces us to particle beam LC/MS, describes the design of HP's particle beam interface, and presents performance data for the HP system.

Equipment for testing integrated circuits at the wafer stage—before the individual chips are separated—typically consists of an automatic test system, a prober, and a probe card. For testing high-pin-count or high-speed devices, conventional probe card designs just don't work reliably in factory conditions. The paper on page 77 presents the results of research aimed at developing an alternative. HP's proprietary membrane probe technology replaces the conventional probe card and its needle probes with a thin, flexible dielectric film supporting a set of microstrip transmission lines that have microcontacts at their ends. Complex, high-density contact patterns are easily formed photolithographically. Contact resistance was found to remain low and stable for up to a million touchdowns with only a simple cleaning every 20,000 cycles. The paper presents performance results from alpha-site tests.

R.P. Dolan
Editor

Cover

An HP SoftBench window environment, showing the OSF/Motif 3D appearance.

What's Ahead

The August issue will contain about one third hardware design and two thirds software design. The hardware consists of the HP 8130A 300-MHz, variable-transition-time pulse generator and the HP 8131A 500-MHz pulse generator. The software is HP's implementation of the Manufacturing Automation Protocol, MAP 3.0.

Making Computer Behavior Consistent: The OSF/Motif Graphical User Interface

Window-oriented user interfaces provide knowledge workers with powerful tools to control their computer environments and increase productivity. The OSF/Motif graphical user interface provides standards and tools to ensure consistency in the appearance and behavior of applications running in the X Window System.

by Axel O. Deininger and Charles V. Fernandez

IMAGINE THE PROBLEMS IT WOULD CAUSE the driving public if there were no standards for the location of the brake and gas pedals on an automobile. Fortunately, the auto industry has standards for the location of certain items that are critical for the operation of an automobile. In the computer industry, standardization and consistent behavior of the user interface for computer applications is not yet a reality. User interfaces defining how people and computer programs communicate with each other still differ from one application to another.

Inconsistent user interfaces make it much more difficult for users to learn and operate different applications. This problem is accentuated in multitasking operating systems such as HP-UX, whose appeal includes the ability to run several programs at once. The cost of such inconsistency is more than just a little frustration for computer users. Inconsistency causes users to be hesitant or to avoid using or purchasing new computer applications, thereby causing lost revenues to application vendors, and possibly lost productivity because the new applications might enable tasks to be done more quickly and efficiently.

Hewlett-Packard's efforts in developing and promoting a cooperative computing environment are based on an interest in industry standards that support a consistent user interface. HP's adoption of the UNIX* operating system as the basis for the HP-UX operating system and early support for industry standards such as the X Consortium and the Open Software Foundation (OSF), are examples of HP's interest in this area. The X Window System™ from the Massachusetts Institute of Technology has been available on HP-UX systems since 1988, and the OSF/Motif graphical user interface, completed for OSF in 1989, is now available on HP-UX 7.0. The OSF/Motif user environment is based on HP's graphical user interface CXI (common X interface). See the box on page 8 for more about OSF.

This article describes some of the concepts and external features provided by the OSF/Motif graphical user interface. The articles on pages 12 and 26 describe the two main components of the OSF/Motif user interface: the OSF/Motif widgets and the OSF/Motif window manager.

Concepts for Consistent Behavior

The following concepts are essential for designing a consistent user interface:

- An object-action design model that is universally applied and simple to understand
- Direct manipulation of objects with immediate and consistent visual cues for feedback
- Tools that are consistent enough to ease the learning burden of novice users, yet flexible enough to allow experienced users to take shortcuts.

The object-action selection model means that the user first selects an object and then selects an action to perform on that object. Standard controls such as menus and push-

The X Window System is a trademark of the Massachusetts Institute of Technology.
UNIX is a registered trademark of AT&T in the U.S.A. and in other countries.

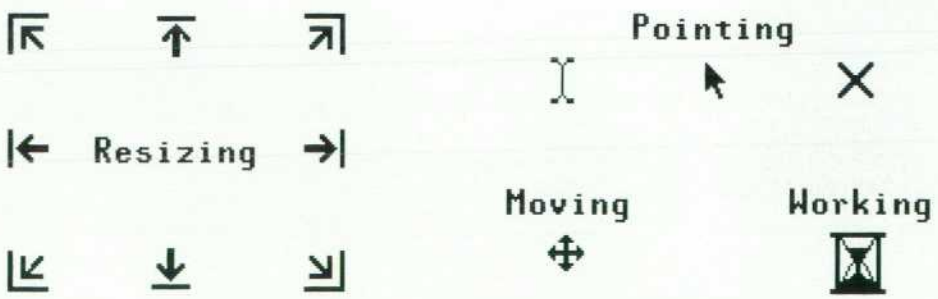


Fig. 1. *Different pointer shapes that provide visual cues to the type of activity.*

buttons represent the selections. The objects typically represent a real-life metaphor that the user is familiar with. For instance, in the HP NewWave Office,[1] the objects include file cabinets, folders, and documents. Consistent behavior implies that the set of controls and objects will always operate in the same way.

Direct manipulation with visual feedback means that the user is provided with a response that somehow represents the action taken, and it is done in real time. For example, when a button on the display is selected, the visual feedback might be that the button appears to be pressed in. Real-time feedback implies that the manipulation of objects on the display is synchronized with the motions of the device (mouse and buttons) being used to perform the manipulation. For example, the events on the display should not lag behind the motion of the mouse.

Consistent behavior does not eliminate individuality, nor does it imply rigid conformity. Much flexibility exists within consistent behavior for application developers to present their applications in the best possible light. Novice users typically make a menu selection by displaying the menu, reading the selections, and then clicking the mouse over the item they want. Experienced users make selections using a quicker method, such as entering a one-letter mnemonic or bypassing some menu levels. The specific controls such as pushbuttons and scroll bars do not represent a finite set, but rather a basic, core set that is expected to evolve as technology changes and users gain more experience.

Tools for Knowledge Workers

To be productive using a computer, knowledge workers must have tools that enable them to communicate with and economize control over the programs running on the computer. The two most common tools for this purpose are the traditional typewriter-style keyboard and a pointing device—usually a mouse.

Standard Mouse Techniques. Traditionally, control over the computer has relied on the user's ability to type. This is being rapidly replaced by the use of pointing devices such as the mouse. A mouse enables the user to control most operations using three actions:

- Pointing. Positioning the mouse pointer over an object. This signals a possible interest in that object.
- Clicking. Pressing and releasing a mouse button selects the object. Double-clicking, or clicking a mouse button twice in rapid succession, selects an object and then performs the designated default action on the object.
- Dragging. Pressing the mouse button and moving the pointer enables a user to move objects, select a range of objects, or browse a menu (depending on the context of the situation).

The shape of the mouse pointer indicates the current operations taking place in the user interface environment.

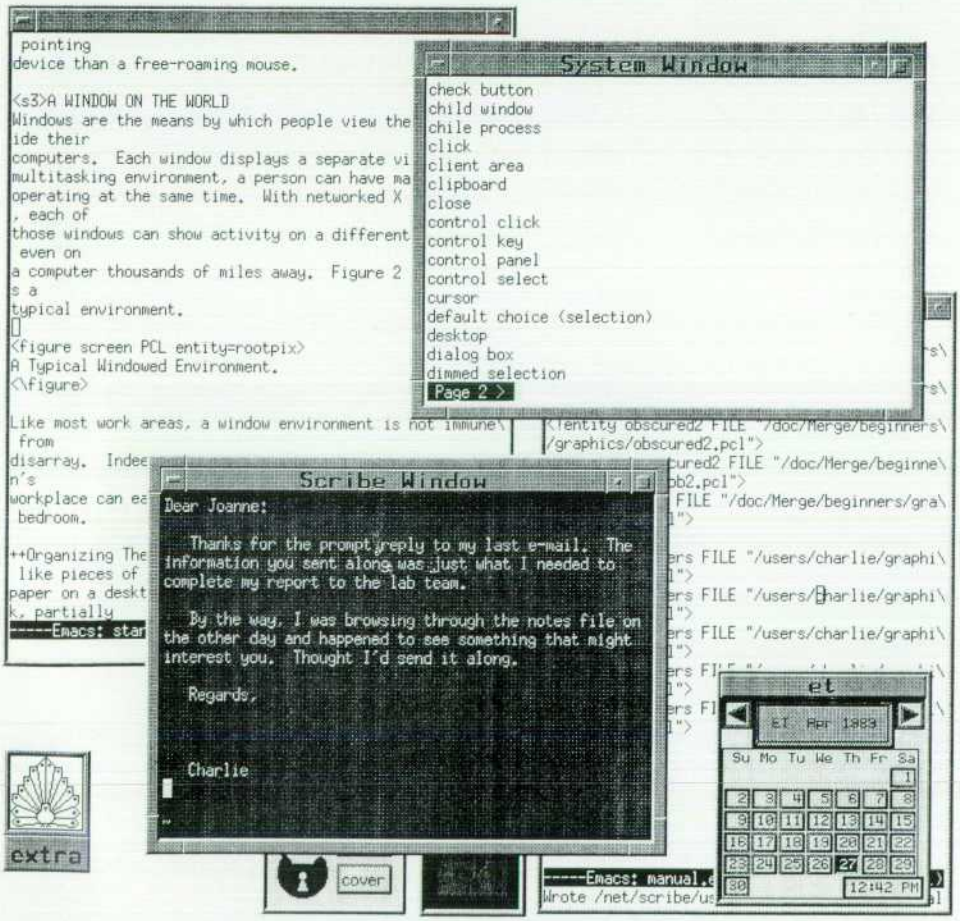


Fig. 2. *A typical window environment.*

Many pointer shapes are possible. Each shape is visually descriptive and provides an important visual cue about the operational state of the interface. Fig. 1 illustrates some common pointer shapes.

By using the modifier keys **Shift** and **CTRL** in combination with the mouse, the user can select a single choice, several choices, a contiguous range of choices, or a noncontiguous range of choices.

Keyboards. A typewriter-style keyboard may be the traditional tool for computer users, but graphical user interface environments like OSF/Motif do not require users to be keyboard experts or to learn the arcane syntax of traditional command-line interfaces.

Although the tools of a graphical user interface such as the mouse are easier to use, keyboards remain the most efficient tool in some cases, particularly for text entry. Also, a number of keyboard alternatives exist. Arrow keys can emulate mouse movement and can be just as fast as a mouse when only a few objects are on the screen, or when the user's hands are already on the keyboard. Single-letter mnemonics and keyboard accelerators for commonly used commands also show that the keyboard is still a useful

OSF/Motif

The Open Software Foundation (OSF) is a group of the leading companies in the computer industry organized to promote open software standards. The foundation is incorporated as a nonprofit, industry-supported research and development organization that has the responsibility to provide software that makes it easier for users to mix and match computers and applications from different suppliers by addressing the following needs:

- Portability. The ability to use application software on computers from multiple vendors.
- Interoperability. The ability to have computers from different vendors work together.
- Scalability. The ability to use the same software environment on many classes of computers, from personal computers to supercomputers.

In response to OSF's request for user interface technology, 39 companies including HP presented their visions of the future of computing. HP's vision of a common X interface (CXI) that united the behavior of Presentation Manager* in the personal computer world with the power of workstations in the UNIX-system world was chosen as the basis upon which to build an OSF user interface standard.

OSF awarded HP a contract to develop and document a CXI-based user interface. This became the OSF's first product, the OSF/Motif user interface. Like CXI, the OSF/Motif user interface is based on a three-dimensional appearance and the behavior of Presentation Manager, which is a standard graphical user interface of the personal computer world. The OSF/Motif product includes a style guide that defines a common user interface behavior consistent with Presentation Manager, a window manager to control graphical objects on the display screen, a software toolkit of widgets and intrinsics with which to build applications, and a user interface language to speed application prototyping. The article on page 12 describes the OSF/Motif window manager, and the article on page 26 describes the OSF/Motif widgets.

*Presentation Manager is a product of Microsoft Corporation.

The OSF/Motif user interface is the most visible piece of what will become a complete OSF/Motif user environment. It thus plays a major role in making the applications that run on UNIX-system-based systems more user friendly. The OSF/Motif environment enables users to operate their computers with graphical controls like pushbuttons, windows, and menus. Where once users had to memorize dozens of obscure commands and type flawlessly, now they need only point with a mouse and click a button.

Fig. 1 shows the interactions between the window manager and a client application. The X Window System is an accepted standard in the UNIX-system world and is the platform for the OSF/Motif widgets and intrinsics. The OSF/Motif window manager provides the Presentation Manager appearance and behavior characteristics for applications. Because OSF/Motif follows a technology standard, users need no longer ponder issues of hardware and software compatibility. Because OSF/Motif follows a behavior standard, users need not learn multiple command sets to control applications. Once they understand direct manipulation, they can control any program.

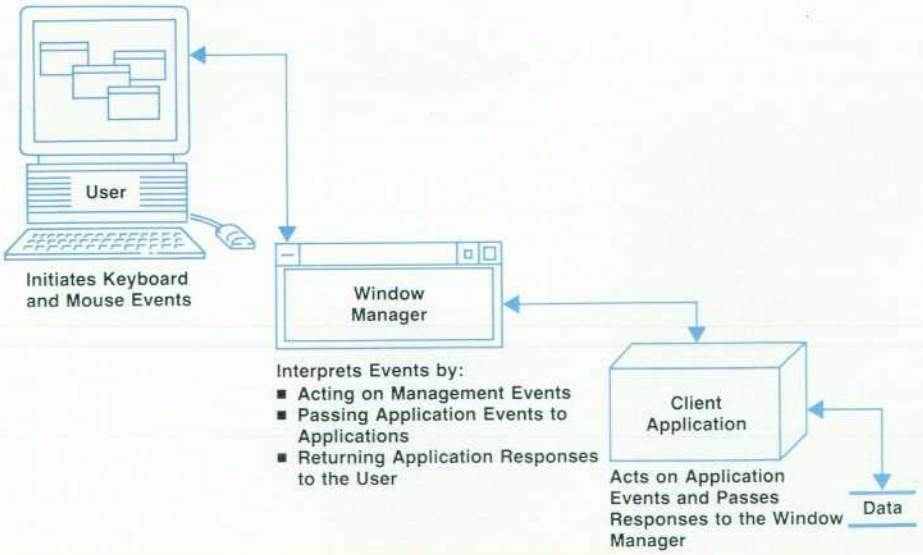


Fig. 1. *Interactions between some of the components in the OSF/Motif hierarchy.*

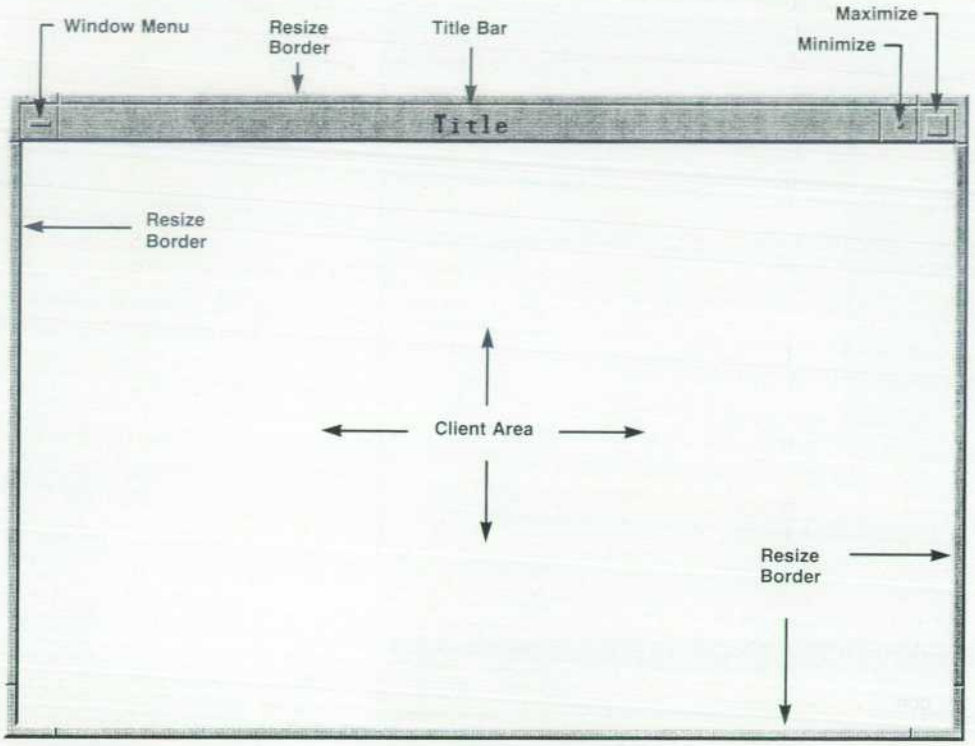


Fig. 3. *The frame of a window in the OSF/Motif environment.*

interface tool.

Special Tools. The keyboard and mouse are by no means the only tools available. Consistent behavior supports the use of many tools for just about all occasions. Hewlett-Packard's Human Interface Link (HP-HIL) provides many interface tools for computer users. Which tool is used depends on the application and the user. For example, a mouse might not be appropriate as a pointing device in all cases. If the application is a computer-aided design (CAD) application, perhaps a graphics tablet or light pen might be a better choice. If the situation is such that a minimum of desk space exists, perhaps a track ball would be a better choice as a pointing device than a free-roaming mouse.

Windows

Windows are the means by which users view the world inside their computers. Each window displays a separate view. In a multitasking environment, a user can have many windows operating at the same time. With networked X Window System technology, each window can show activity on a different computer, even a computer thousands of miles away. Fig. 2 illustrates a typical window environment.

Like most work areas, a window environment is not immune to disarray. Indeed, with remarkably little effort, the workplace (display) can easily become cluttered to the point of distraction. Windows typically overlap like pieces of paper on a desktop. New windows open on top of the stack, partially obscuring older windows lower in the stack.

There are a number of ways to organize the work area. Controls are present on the window frames for the convenience of mouse users. Fig. 3 shows the layout of a typical window in the OSF/Motif window manager environment. Windows can be moved out of the way by dragging the title bar. The window frame itself is not just a border; when grabbed by the mouse, the border stretches or shrinks to resize the window.

When moving or shrinking a window is not enough to get it out of the way, the window can be turned into a graphical icon by clicking on the minimize button in the window frame. The icon saves space on the screen without halting the application running in the window. This is analogous to a person putting a clock in a desk drawer—the clock still works, it's just out of the way.

To give a window undivided attention, the user can click on the maximize button in the window frame. This will enlarge a window to its maximum size and will often cause it to cover the entire screen. This is a useful feature for complex CAD design.

Menus

Consistent behavior provides a number of ways for users to control the windows in their work areas. The idea is that no one way will be correct for everyone, so by building flexibility into the environment, users can pick a way to manage windows that best fits the situation. To help provide this flexibility, every window has a window menu. Users can display a window menu either by clicking the left mouse button with the pointer positioned over the window menu icon for that window, or by pressing **Shift** and **ESC** simultaneously. If the window menu is hidden, it can be revealed with the click of a mouse button.

The window menu shows all of the window management commands available for a window. Fig. 4 shows the contents of the default menu for the OSF/Motif window manager. This menu duplicates the commands embedded in the window frame and may provide different commands as well. To initiate an action from the menu, the user positions the mouse pointer over the desired selection and clicks the left mouse button. For keyboard-oriented selec-

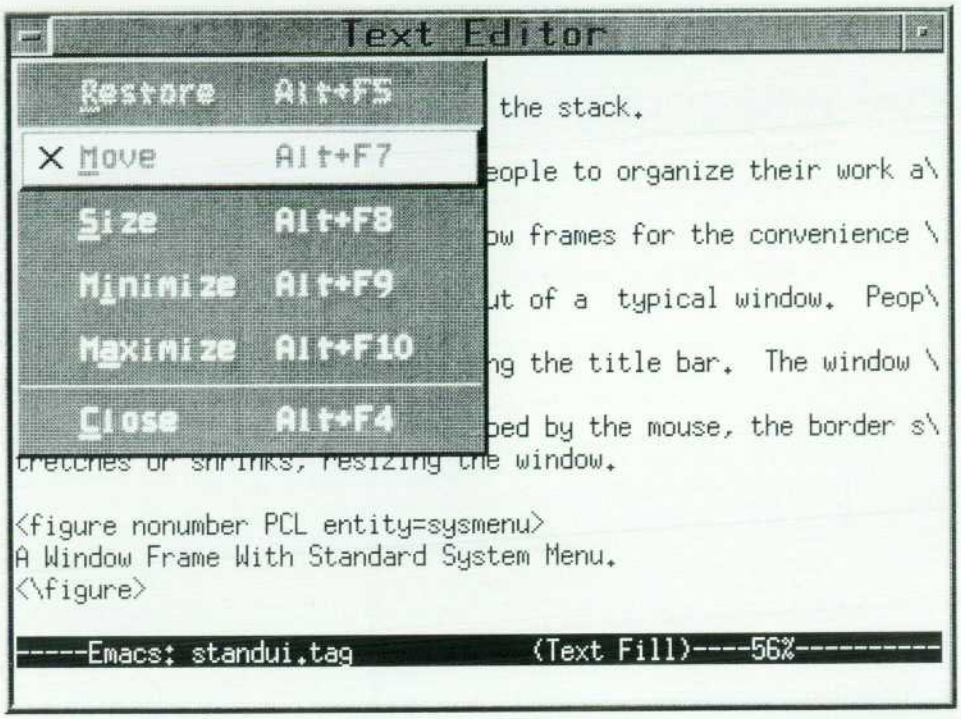


Fig. 4. *A window showing a window menu.*

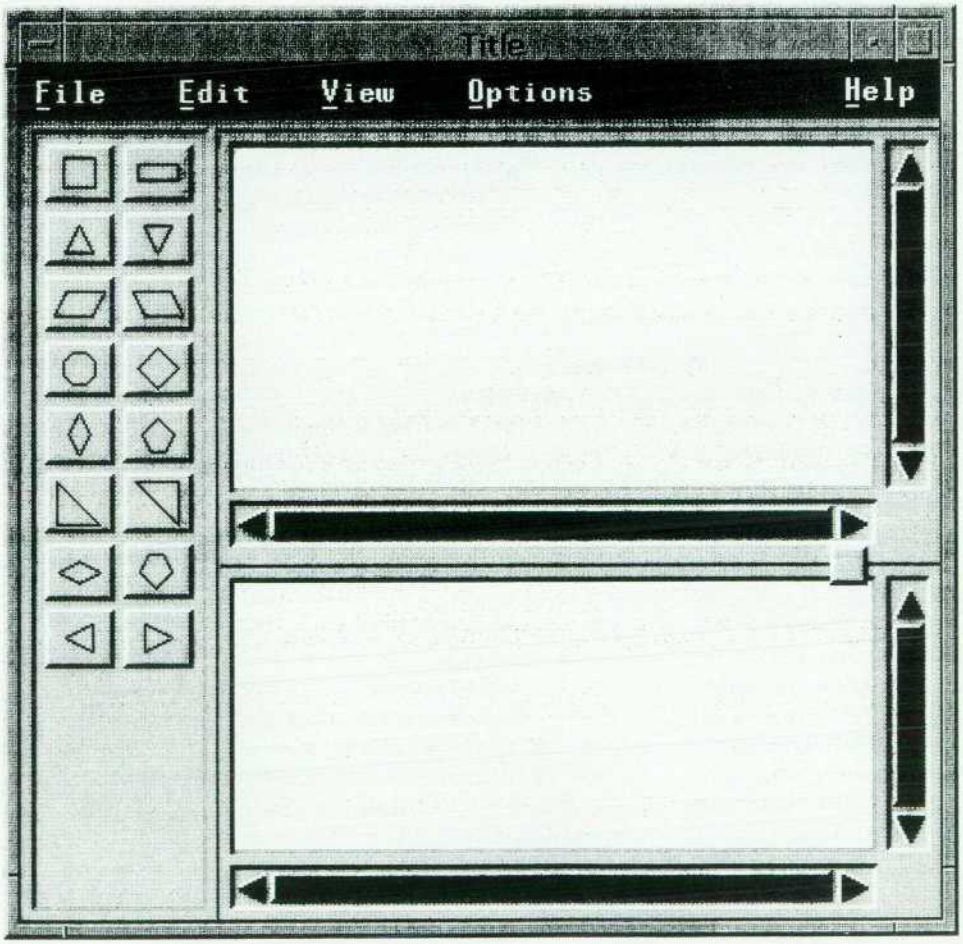


Fig. 5. *A typical application main window.*

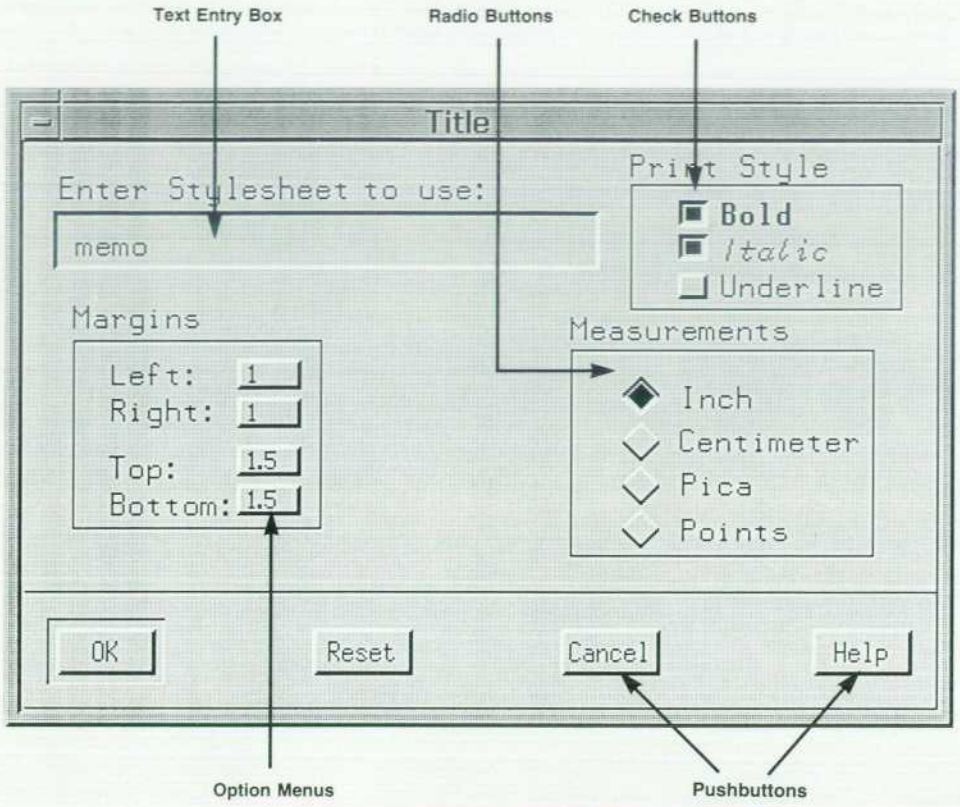


Fig. 6. *A sample dialog box.*

tion, the user can type a one-character mnemonic. Mnemonics are the underlined characters in a menu entry (see Fig. 4). Typing the keyboard accelerator (shown after the menu entry) will perform the command without displaying the menu first. Keyboard accelerators are the fastest way to invoke frequently used commands. For example, pressing the keys **Alt** and **f9** simultaneously will minimize a window. Users can customize keyboard accelerators to suit their personal needs.

Controlling Applications in the Window

Of greatest interest to users is not the window, but the application running in the window. Fig. 5 shows a typical application's main window. The bulk of the space in the window (known as the client area) is reserved for displaying the application. This can be text for a word processor or a schematic for a CAD package.

Commands used to control the application are tucked away in the menu bar at the top of the window. The menu bar lists the titles of available menus. To display a menu, the user positions the pointer over the menu title and clicks the mouse button, or uses one of the keyboard techniques. Selecting a command from a menu bar menu is the same process as selecting a menu item from the window menu described earlier. The menu bar menus can contain both commands, which are actions that occur immediately, and settings, which are states of being (such as double-spaced text) that are not actions themselves but that affect subsequent actions such as printing.

Standard Menus for Standard Actions. Standard menus are recommended for standard actions to ensure consistent behavior among applications. The titles of the standard menus for an application are listed in the menu bar. Three of the standard menus include:

- File. Contains file actions like opening, creating, saving, and printing a file.
- Edit. Contains edit actions like undoing, cutting, copying, pasting, and clearing sections of a file.
- Help. Contains helpful information like context sensitive instructions, information on the use of keys, index listings of help topics, and information on how to use the help function.

Pop-up Menus, Check Boxes, and Pushbuttons. The menu bar presents an effective compromise between providing an efficient storehouse for a large number of actions and presenting visual cues so users can readily see what choices are available.

Pop-up menus are a good choice for applications that want to place the most commonly used actions under the fingertips of mouse users. They are particularly effective in text and graphics editors. Users can select a range of text and press the second mouse button to pop up the menu. There is no need to travel with the mouse pointer to the menu bar. Pop-up menus are very fast when used with the mouse drag technique.

Applications that want to make certain action choices visible all the time can use pushbuttons to place them in control panels. Radio buttons and check boxes are used in the same way for settings. All of these controls are modeled after real-life objects. Pushbuttons are found on many electrical appliances. The radio buttons stem from a car stereo, hence their use for mutually exclusive settings (a radio can be tuned to only one station at a time). Check boxes appear on many paper forms such as job applications.

Dialog Boxes. Dialog boxes are so named because they enable users to carry on a dialog with an application. Fig. 6 shows an example of a dialog box associated with a hypothetical copy command. The sample dialog box contains a text entry box for entering the name of a style sheet, a set of radio buttons for indicating mutually exclusive units of measure, check buttons indicating settings for type style, option menus provding a limited choice of margin sizes, and a row of pushbuttons indicating what action should be taken.

Conclusion

Window-oriented graphical user interfaces offer an opportunity to make the computer as pervasive an appliance as the automobile. But, if they are truly going to do so there must be standards for consistent behavior. A behavior standard has advantages for both computer users and computer vendors. Users are finding programs easier to learn and use. The market for standards-conformant applications is growing. Vendors are finding they can produce more applications while concentrating their product efforts on developing performance and features rather then developing user interfaces.

References

1. B. Lam, et al, "The NewWave Office," *Hewlett-Packard Journal*, Vol. 40, no. 4, August 1989, pp. 23-31.

The HP OSF/Motif Window Manager

The HP OSF/Motif window manager, which is built on top of the X Window System, is a window management interface that provides a 3D enhanced Presentation Manager appearance and behavior using HP OSF/Motif widgets.

by Brock C. Krizan and Keith M. Taylor

THE X WINDOW SYSTEM, Version 11 (also known as X or X11)[1,2] was developed as a platform on which a variety of user interfaces can be implemented. The particulars of a user interface are determined by the X clients that run on the system. X clients are programs that use X to display information and receive input. The HP OSF/Motif Window Manager (mwm) is one such client.

Fig. 1 shows the relationship between the X Window System and clients. The OSF/Motif window manager mwm implements an interface that allows user and client manipulation of windows. Mwm dictates through its window management interface a particular user interface behavior. The principal objects that are manipulated using the window manager are the client windows placed directly on the background, or root, window of the screen. Windows within these top-level client windows are managed by clients and are not directly manipulated by the window manager. Users are provided with ways to move and resize windows, to direct all keyboard input to a particular window, and to install color maps[3] for a window.

X, as it comes from the Massachusetts Institute of Technology (MIT), provides mechanisms for supporting clients that implement a variety of window management user interfaces. A sample window manager, uwm, is distributed by MIT. Several window managers have been implemented at companies and universities to meet the needs of a particular application environment, to emulate some non-X Window System user interface, to provide the latest new and improved window management interface, or to provide personal customizations of uwm. Window managers are one of the most common types of X clients.

With so many window managers available, implementing another window manager would seem to be a waste of time. However, the window manager is an essential and highly visible part of any window system user interface, and the usability of a system can be significantly affected by the window manager. Prior to the availability of mwm's predecessor, the HP window manager, or hpwm, HP customers who had access to X used the sample window manager uwm or, less frequently, window managers available in the public domain. HP wanted to give users an interface that was visually refined, consistent, easy to learn, and based on industry standards.

Hpwm supports industry standards in appearance and behavior as well as X standards for client interoperability. The appearance and behavior of hpwm is based on Presentation Manager, which also defines the window management appearance and behavior for HP's NewWave Office. Users already familiar with the Presentation Manager standard from the personal computer environment now find their skills useful on an HP-UX workstation. The three-dimensional visuals of hpwm represent a refinement, not a change, from Presentation Manager standard appearance.

In 1988, the Open Software Foundation (OSF) accepted HP's proposal that hpwm be adopted as the basis for the OSF/Motif window manager. The commitment to Presentation Manager as an industry-standard user interface was key in OSF's decision. OSF/Motif encompasses several

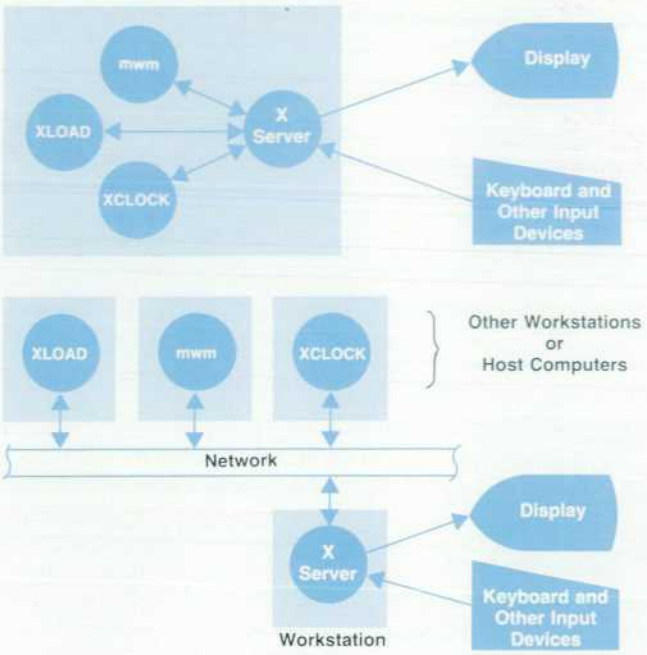


Fig. 1. *The X client-server model. In this model the X server is near the user and controls the display and manages the input devices. The clients in this model are the applications that talk to the server using the X protocol, such as* mwm, XLOAD, *and* XCLOCK. *The X protocol allows the clients and server to run either on the same machine or on different machines connected by a network. (a) X client-server architecture on stand-alone workstation. (b) X client-server relationships in a distributed environment.*

technologies built on top of the X Window System, and the new OSF/Motif window manager is only one piece of the OSF/Motif environment.

Window Manager Characteristics

The basic set of functions that a window manager provides is relatively constant in any window system. On the other hand, the appearance and behavior vary greatly from one window manager to another. Many of the characteristics of mwm were leveraged from hpwm. This allowed us to meet an aggressive schedule and still satisfy the functionality and quality goals for mwm.

Common Appearance and Behavior

Like hpwm, the appearance and behavior of mwm are heavily influenced by Presentation Manager. Indeed, the default behavior of mwm, as well as that of the OSF/Motif widgets, is as close to Presentation Manager as is practical. A key benefit of this is that users can easily move between systems running MS/DOS® or OS/2 and systems running the HP-UX operating system and X Windows. Nevertheless, some differences were admitted into the design of mwm to satisfy the variety of HP-UX users and to use the power of engineering workstations. This has led to a window manager with a high degree of configurability and an enhanced appearance over Presentation Manager.

Key behavioral aspects of Presentation Manager and the OSF/Motif environment include the direct manipulation of objects and an object-action paradigm for user interaction. Direct manipulation involves using the keyboard and/

MS-DOS is a U.S. registered trademark of Microsoft Corporation.

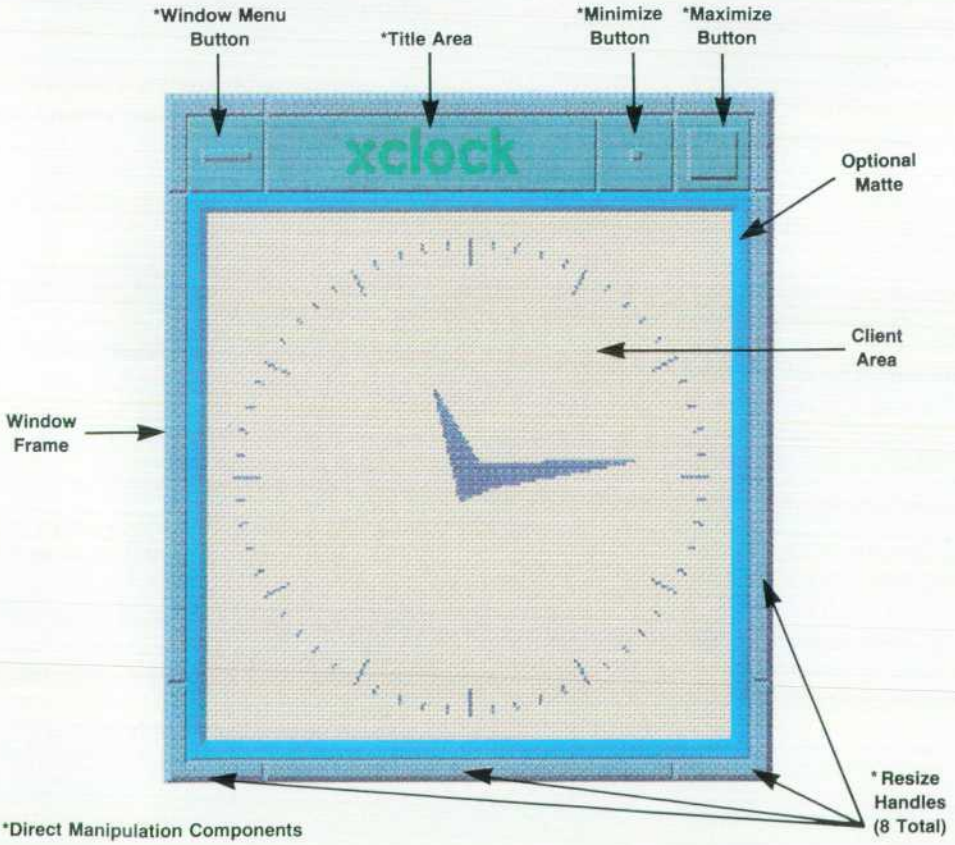


Fig. 2. *A client window and the various window manager components.*

or mouse to do window management functions directly, such as moving and resizing a window. A user does not enter a command such as move -w mywindow x=10 y=100, but rather drags the window using the mouse to the new position. With the object-action paradigm, the user selects an object and then performs some action on the object.

3D Appearance

One deviation from strict adherence to the Presentation Manager standard is in the appearance of the user interface components. The three-dimensional visual style developed for earlier HP products was accepted by OSF as part of the OSF/Motif standard. 3D components appear in both the window manager and the OSF/Motif widgets. Use of 3D components strengthens the direct manipulation paradigm by providing visual objects that react naturally to user actions (e.g., buttons appear to go in when pressed).

Mwm uses the OSF/Motif widgets to provide visual and operational compatibility with other clients that use the OSF/Motif widgets. All parts of mwm are displayed with the 3D visual style. This includes the window manager frame, icons, and menus. A key factor that influenced mwm's use of the 3D visual style was the prevalence of window manager components on the screen. The challenge was to provide a 3D appearance but not to distract from or limit the client user interface. Mwm is designed to be frugal with its use of screen space, subtle in its use of 3D indications, and restrained in its use of color. Fig. 2 shows a client window and the various window manager components.

Configurable Appearance and Behavior

Although mwm implements the Presentation Manager behavior with a 3D visual style, configurability was considered a desirable departure from a strict Presentation Manager model. In some cases configurability applies to aspects of the user interface that are not constrained by the standard appearance and behavior. The colors of components and the fonts that are used fall into this area. Configurability can also alter the standard appearance and behavior in fundamental ways. Since it is almost impossible to provide a single, fixed user interface acceptable for all users, configurability is highly desirable.

Configurability of mwm is provided in a way that does not burden users who are satisfied with the window manager's standard appearance and behavior. Mwm provides the standard appearance and behavior as a default and allows for user customization. Configuration is only necessary if there are specialized requirements. In addition, mwm provides a function that resets all customized mwm settings to default values to give the user a known starting place from which to work.

It is anticipated that only a small group of system administrators will want to customize mwm. To make their job easier, mwm uses the same resource names for specifying configuration values for colors and fonts as are used for OSF/Motif widgets. The result is that configuring mwm is similar to configuring any client built using OSF/Motif widgets.

ICCC Compliance

Compliance with the standard Inter-Client Communication Conventions (ICCC) developed by the X Consortium is a requirement for any X client. These conventions are intended to facilitate interoperability of X clients. Clients that follow the conventions can coexist on the same screen and not interfere with each other's behavior. This applies particularly to the communication between clients and window managers. The ICCC is the basis for the programmatic interface to X window managers (see the box on page 23).

Mwm implements the ICCC standard in a way that is compatible with the standard OSF/Motif behavior. This allows a user to run a client even though it was developed without specific knowledge of mwm.

Mouse and Keyboard Interfaces

Window managers are often implemented with a reliance on the mouse for user interaction and the keyboard is ignored. The OSF/Motif behavior specifies a functional equivalence between mouse and keyboard interaction.

Mwm is fully functional when it is run on systems that do not have a mouse input device. Not only does the standard OSF/Motif behavior have keyboard support, but mwm supports features beyond the OSF/Motif standard. For example, keyboard and mouse interaction can be mixed together, even while doing a particular action such as moving a window.

OSF/Motif Window Manager Operation

Mwm has two basic phases of operation: start-up and event processing. At start-up, mwm asserts itself as the window manager for a particular screen, processes configuration information, and takes care of currently displayed client windows (see Fig. 3). Event processing is the steady-state phase of operation. Like most X clients, mwm is event driven—that is, it waits for some type of X event, processes the event, and then waits again. In the event-processing phase, all mwm actions are the direct result of some event.

Start-up

When mwm first starts up it must indicate to the X server that it wants to be the window manager. The X server has no notion of a special window manager client, but there are some X facilities that are necessary for window management that cannot be accessed by more than one X client. By asserting control of these facilities, mwm effectively locks out other window manager clients (conversely, mwm is locked out if another window manager is already running).

The primary facility over which mwm gains control is the facility for redirecting several types of X requests from other clients (see Fig. 4). Usually a client makes a request to the X server to do a function and that function is done immediately by the server. With a redirected request, the function is not handled by the X server, but is passed to the redirecting client (i.e., the window manager). The window manager decides how to handle the redirected request and then makes the request, sometimes changing the request to be compatible with its window management policies.

The types of X requests that are redirected by mwm in-

clude:

- Window configuration (moving and resizing)
- Window stacking (who's on top of whom)
- Window mapping (display of a window on the screen).

These requests are redirected only when they apply to top-level client windows, which are windows displayed directly on the background or root window of the display. Using its ability to redirect X requests, mwm can control when, where, and how client windows are displayed.

Once mwm has asserted itself as the window manager, it can then configure itself and prepare to do event processing. In general mwm has its configuration specified through resource files like other X clients (see Fig. 5). These resource files contain user-specific configurations, client-specific configurations, and screen-specific configurations.

Resources that are specific to fonts, colors, and bit maps are defined and referenced in general-purpose resource files. However, not all configuration resources can be conveniently specified in a general-purpose file. The mwm resource description file (usually called .mwmrc) contains descriptions of resources that are difficult to specify in the general-purpose resource files. Mwm menus, mouse semantics, and keyboard semantics are described in the mwm resource description file and referenced in other resource files.

The last thing that mwm does during its start-up phase is adopt client windows that are currently being displayed. Mwm assumes control over the placement of client windows on the screen. In the usual case where mwm is the first client to be started there will be no clients to adopt.

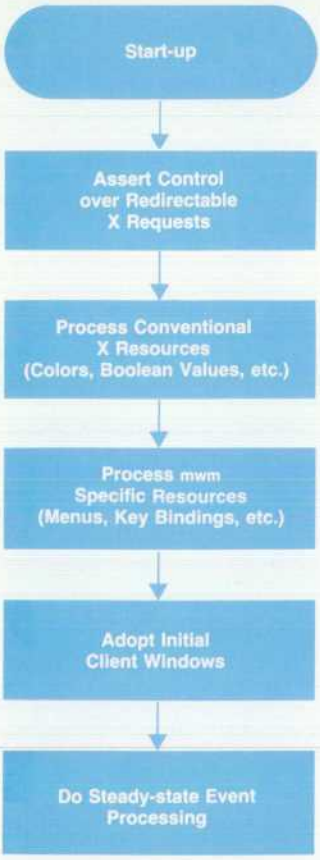


Fig. 3. *OSF/Motif Window Manager start-up process.*

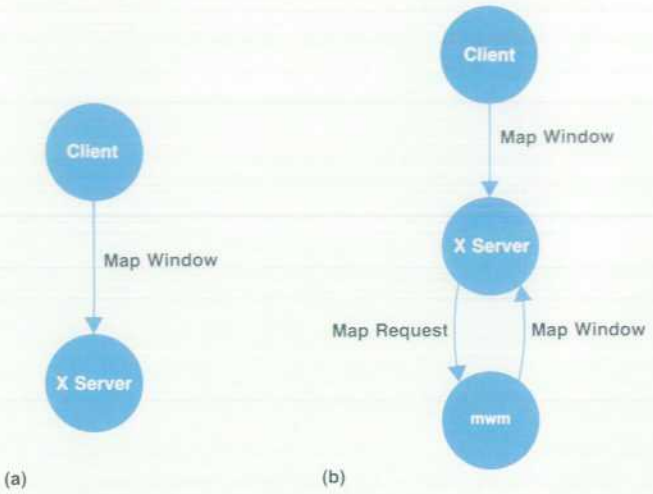


Fig. 4. *Event redirection. (a) If no window manager is running (no redirection), the client's window mapping is done immediately. (b) When* mwm *is running, the server redirects the client's map window request to* mwm. Mwm *adds its window border before asking the server to complete the window mapping.*

Processing

After mwm completes start-up it goes into a loop waiting for and processing events. Events are messages from the X server that are generated as the result of some user or client action.

When a top-level client window is to be displayed on the screen, the window manager receives a map request event. In processing the request, the window manager retrieves client-specified and user-specified configuration information to place the client window on the screen. The client window is reparented to a window manager frame window. In effect, the client window is placed inside a window manager frame window. This is the mechanism that allows mwm to give all clients a common top-level window border. In the frame window, around the outside of the client window, are placed the window manager direct manipulation components shown in Fig. 2. Once the client window is dressed up in its window frame, it is placed on the screen.

User interaction with the window manager results in mouse (button and motion) events and keyboard (key)

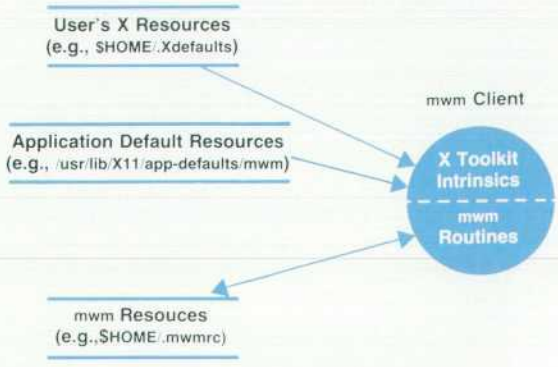


Fig. 5. *Resource files used by* mwm.

events. When a user interacts with a direct manipulation window manager component, a stream of events is generated. Mwm associates the events with a particular user interface component and invokes the associated function. Immediate visual feedback of the user's interaction maintains the appearance and behavior of the direct manipulation interface.

Users can configure window manager actions to be invoked by particular key or button events. This interface to the window manager is in addition to the standard interface which is based on direct manipulation of window manager components. Mwm arranges with the X server to grab button and key events that invoke window manager functions. This grab mechanism allows the window manager to get the events even while another X client window is receiving keyboard input.

Termination of mwm is triggered when a window manager function invoked by a user or by an event indicates that the X server has been shut down. When mwm is terminated, the window frames that belong to mwm are destroyed. Normally, all the child windows of a window that is being destroyed are also destroyed. However, since mwm reparents client windows to their window frames at start-up, the desirable behavior is for the client windows to be reparented back to the background (root) window so that the clients can continue to run. To accomplish this, mwm uses the X11 save set mechanism to cause client windows to be reparented back to the root window when mwm terminates. By placing all client windows that have been reparented to window frames into its save set, the windows are automatically reparented back to the root window by the X server when mwm terminates.

Restart

The restart function is invoked when a user wants to reconfigure mwm. Restart is necessary because some resources are only read by mwm in its start-up phase. Any aspect of the mwm configuration can be changed at any time using the restart function. The window manager restart function effectively terminates the current instantiation of mwm and starts a new one. This function is special in that it causes mwm to make a complete pass through both of its operational phases. The event that invokes the restart function is processed in the steady-state event processing phase. Restart execution begins with the termination of mwm and completes when mwm starts up again.

OSF/Motif Window Manager Implementation

Like the features and characteristics of mwm, most of the code and design for mwm were leveraged from the HP window manager. The period when hpwm was designed and implemented was one of rapid change for X and for HP's use of X. This had to be taken into account in formulating an implementation strategy for mwm. For example:

- Hpwm was implemented at the same time that there were new developments in user interface technologies and components. However, to minimize risk, stable technologies were used in favor of the newer ones.
- The user interface components that hpwm used were often first-generation products. Therefore, visual and performance tuning of these components could not be relied upon.
- Prototype versions of hpwm were required to refine the 3D visual style, to support usability testing, and to support prototype application environments.
- Standards that hpwm used were under development in parallel with the implementation of hpwm.

The implementation strategy used for hpwm involved substantial prototyping and design, followed by bottom-up reimplementation. Prototyping and design accounted for more than half of the engineering and calendar time spent on implementing hpwm. Development of a prototype delayed dependencies on user interface components and facilities. The prototype was used to identify visual and performance problem areas requiring design refinements. Design decisions were substantiated or changed based on experience with the prototype.

After the prototype and hpwm, mwm can be viewed as the third pass on the window manager. The experiences gained from the earlier efforts were used during the definition and implementation of mwm. Also, the use of the hpwm engineering team for the development of mwm allowed for rapid and effective progress once the functionality was defined.

Widgets and Windows

There are two principal levels in which a programmer can write a user interface for an X client: the high level using a widget library like the OSF/Motif widgets and the low level using the X library. Widgets provide high-level objects (like menus and buttons) that embody the semantics of specific user interactions, and the X library provides only basic window functionality. Since the HP window manager user interface was implemented using a mixture of widgets and the X library, mwm was implemented using a similar mixture of libraries.

Mwm uses the OSF/Motif widgets to implement its menus. This provides appearance and behavior consistent with applications that also use the OSF/Motif widgets. It also leverages the engineering effort that went into the design

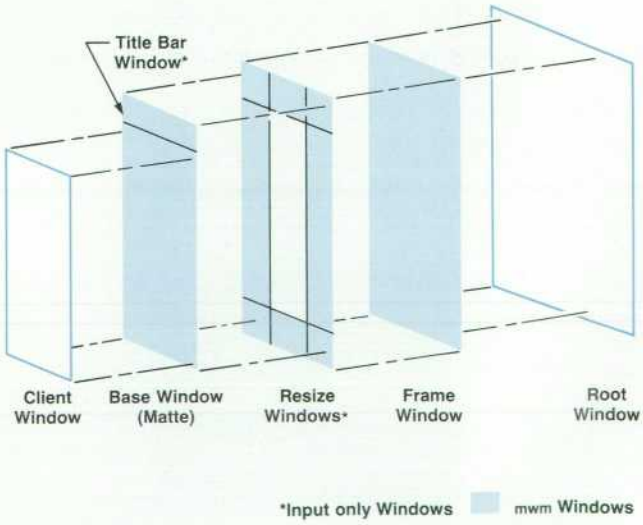


Fig. 6. *Exploded view of an* mwm *window frame.*

and development of the menu widgets.

Mwm does not use any widgets for the window frame components (title bar, resize handles, and border). To understand why, it is necessary to examine the decision made for hpwm. First, at that time, the available widgets did not offer enough control over the thickness of the 3D beveling (the top shadow and bottom shadow highlights) to give the desired 3D effect. The window frame has oddly shaped pieces and complicated joints that require explicit drawing by the window manager. Also, the visual design requires single-pixel beveling between components of the window frame.

Second, although using multiple widgets as buttons for the frame decoration simplified some aspects of event handling, it complicated changing the color of the entire window frame. Some window managers change only the title bar appearance to indicate the active window. However, this can be difficult or impossible to spot depending on the size of the window and the degree to which it is obscured. Mwm and hpwm change the color of the entire frame to indicate keyboard focus. Thus, the functional and performance needs of hpwm required a solution other than using the widgets available at the time.

It is important to note that with the latest version of OSF/Motif widgets, most of the objections that caused the initial decision not to use widgets for the window frame have gone away. For example, OSF/Motif provides widgets called "windowless gadgets" that provide better performance than the widgets with windows that we used. However, there are still some mwm user interface requirements, such as the resize cursors, that require either widgets with windows or special processing.

An mwm window frame consists of ten windows for drawing, cursor presentation, and event handling (see Fig. 6). The main frame window has the root window as its immediate parent. It is an input/output window and is the window to which frame drawing is done. Above the frame window are eight input-only windows for the resize handles. Each of these windows has its own cursor to indicate the type of resize that can be started in that area. The next layer up includes an input-only title window which is used to display a different cursor for the title area and partially

```
# This is a fragment of an .Xdefaults file containing some
# representative settings for the OSF/Motif window manager.


#  General Appearance and Behavior Resources
#
# Set private mwm button and key bindings (see .mwmrc).

Mwm*buttonBindings:             MyButtonBindings
Mwm*keyBindings:                MyKeyBindings

# Remove active label from icon decoration and tighten
#     icon placement.

Mwm*iconDecoration:             image label
Mwm*iconPlacement:              left bottom tight


# Component Appearance Resources
#
# Use these colors on the "active" window
# (the window that gets keyboard input).

mwm*activeBackground:           turquoise
mwm*activeForeground:           white

# Use this color scheme on "inactive" windows.

mwm*background:                 cadet blue

# font to use for Mwm (different fonts for titles, menus and icons)

Mwm*fontList:                   helvR18
Mwm*menu*title*fontList:        ncenR24
Mwm*icon*fontList:              helvB14


# Client-Specific Resources
#
#    + HPterm gets a special icon image
#    + Reduce frame decoration for xload and xclock.

Mwm*HPterm*iconImage:           /users/keith/Bitmaps/terminal.xbm

Mwm*XClock*clientDecoration:    border

Mwm*XLoad*clientDecoration:     menu title minimze
```

Fig. 7. *A sample resource file showing some sample configurations for the OSF/Motif window manager.*

obscures the upper resize windows. This layer also includes a base window on which the client window sits. The base window partially obscures the lower resize windows and is used for drawing the client matte if one is specified. The client matte is a feature of mwm that allows the user to create an extra level of distinguishability for a window by specifying a color for the area below the title bar window shown in Fig. 6. An example of this feature is illustrated by the strip labeled optional matte in Fig. 2.

The primary reason there are so many windows is to get the desired cursor behavior. As the pointer moves into each resize area, the cursor changes to indicate the type of resize that can be started in that area. This is accomplished in mwm by creating input-only windows that overlay the graphics in the frame window. Each window is created with a different cursor attribute. A benefit of this, from mwm's point of view, is that the X server takes care of changing the cursor shape when the pointer enters or leaves these windows. Careful overlapping of the title bar window and the base window clips the corner resize areas to their characteristic nonrectangular shapes.

Configuration

The mwm approach to configuration can be characterized in terms of consistency, flexibility, performance, and usability. These attributes were achieved using the following techniques.

- The mwm configuration is based on the values of resources set in the resource files. Mwm resource names are consistent with the standard OSF/Motif widget names. The names are defined such that a single entry in a resource file can be used to specify values for related resources. For example, the background color used for all window manager components can be specified with one resource.
- Most configuration overhead occurs at start-up and is avoided during user interaction, when quick feedback is required.
- All mwm resources have default values that are consistent with the standard Presentation Manager behavior and 3D appearance.

Three types of resources are processed by mwm: general-behavior resources, component-specific appearance re-

```
# This is an annotated fragment of an .mwmrc file
#
# Workspace menu description
#   This menu is posted by a button binding (see MyButtonBindings below)
#   It offers the options of
#       + starting an hpterm terminal emulator (80 columns by 42 lines).
#       + starting an hpterm that is logged into a remote system (bill).
#       + starting an hpterm that is logged into a remote system (dave).
#       + refreshing the entire display
#       + restarting the window manager
#
Menu Workspace
{
    "Workspace Menu"    f.title
    hpterm      f.exec  "hpterm =80x42&"
    bill        f.exec  "hpterm =80x42 -T bill -n bill -e rlogin bill"
    dave        f.exec  "hpterm =80x42 -T dave -n dave -e rlogin dave"
    no-label    f.separator
    Refresh     f.refresh
    Restart     f.restart
}


#
# key binding descriptions
#   This key binding replaces the default Shift-Esc binding
#   that posts the window menu.
#

keys MyKeyBindings
{
    Alt<Key>Escape      icon|window     f.post_wmenu
}


#
# button binding descriptions
#   These button bindings
#     + post a workspace menu over the root window (screen background)
#     + provide an accelerated move function for icons and windows
#     + provide an accelerated resize function for windows
#

Buttons MyButtonBindings
{
    <Btn1Up>            root            f.menu WorkMenu
    Alt<Btn1Down>       icon|window     f.move
    Alt<Btn2Down>       window          f.resize
}
```

Fig. 8. *A portion of a file defining* mwm *general behavior resources.*

sources, and client-specific appearance and behavior resources. Fig. 7 shows a portion of a file with some sample resource settings.

General-Behavior Resources. General-behavior resources are used to define window manager policies such as directing keyboard input to a particular client window and specifying when to install a client window's color map. Button and key associations* to window manager functions are also specified. For example, pressing the left mouse button with the pointer over the root window can be configured to post a menu. The general-behavior resources are completely processed when mwm is started.

Fig. 8 shows a portion of the mwm resource file used to define the button and key associations declared in the sample .Xdefaults file shown in Fig. 7. The first part of the resource file, labeled Menu Workspace, defines the appearance and the functions associated with the menu shown in Fig. 9. For example, for the menu item hpterm, the function f.exec is executed when hpterm is selected, and the field "hpterm =80x42&" defines the HP-UX command that is executed by f.exec to start a new hpterm terminal emulator that is 80 columns by 42 lines in size. The key and button bindings define the event (key or button selection), the context (where the event occurred), and the action associated with key and button selections. From the key binding description in Fig. 8, the key sequence **Alt ESC** entered while in an icon or window context would cause the Window menu to be displayed.

Component-Specific Appearance Resources. Mwm high-level components include the window frames, icons (small representations of client windows), and window manager menus. These components use the same set of appearance configuration resources. The resources specify the colors and textures to use for 3D appearance and the font to use for displaying text. Defining the 3D appearance of a component can involve specifying the texture and color for the foreground, the background, the top shadow, and the bottom shadow of the component. Default component-specific appearance resources can be used to avoid specifying any resources for a monochrome system, and only the background color on a color system. On a color system the top shadow, bottom shadow, and foreground colors are generated algorithmically. The algorithm generates an effective 3D visual appearance based on a background color. New colors are generated by shifting the RGB values of the background color. The values are shifted to make the top shadow color lighter, the bottom shadow darker, and the foreground color much darker than the background color.

The window frame and icon components have a set of appearance resources for both active and inactive states. A component in the active state can receive keyboard input, and a component in the inactive state cannot. In the case of a window frame, the client window receives the keyboard input. For human factors and performance reasons there is a single 3D color scheme for active components and a single color scheme for inactive components. Multiple, client-specific color schemes for active and inactive states led to problems with identifying the client window that was supposed to receive keyboard input. Also, interactive performance is maintained by allocating all component colors and graphics contexts (graphics state information used in X drawing requests) at start-up time.

Client-Specific Appearance and Behavior Resources. Resources used by the window manager to customize components for particular client windows are client-specific resources. The image in the icon representation of a client window can be specified. Client-specific colors can also be specified to color the client icon image and the 3D matte that fits within the window frame. Client-specific resources are retrieved based on the resource name or class of a client window. The resulting X resources and window manager components are cached to avoid resource processing overhead when several clients of a particular name or class are run. This enhances performance because client windows are placed on the display frequently during user interaction.

*Also called key bindings.

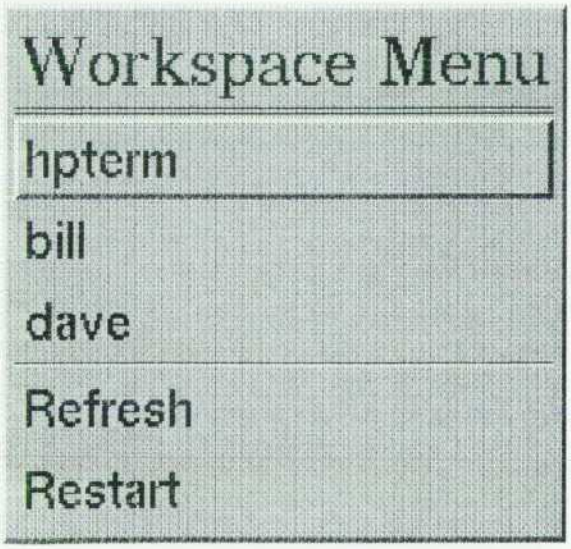


Fig. 9. *The* Workspace *menu described in the* .mwmrc *file in Fig. 8.*

Event Processing

Mwm event processing is designed to handle different types of events and event contexts. The events that are processed include button presses, pointer motion, window destruction, and many more. Event contexts define the locations where the events occurred. These locations include the root window, widgets, nonwidget window manager components, the window frame, an icon, and client windows. The window frame has subcontexts such as the system menu button, the resize border handle, the title, and the minimize button.

Table I lists some events that are processed, the contexts they occur in, and the actions taken when the event occurs.

Events with a root window context generally involve newly displayed windows, destroyed client windows, or the invocation of a window manager function that is not client-specific (e.g., repaint the screen). Events for mwm menus have a widget context. Events with a nonwidget context are generally on the window frame and are often related to user interaction with the direct manipulation components such as the resize handles. Events with a client window context are typically notifications about the actual or desired state of a client window.

The event-processing loop for mwm has the following

Table I
Events, Contexts, and Actions

Event	Root Context	Widget Context	Non-widget Context	Client Window Context
Map Window	Decorate the window with a new frame and place it on the display.			
Window Destroyed	Remove the frame from the display and recover resources.			
Button Press		Post (show) menu.	Activate frame component button or resize handle.	
Pointer Motion		Move selection cursor.	Move or resize frame outline.	
Button Release		Unpost (hide) menu.	Commit action.	
Change Color Map				Install color map for window.

flow of control.

- Use the Xt Intrinsic function XtNextEvent to retrieve the next event sent by the X server.
- Identify the event context. Events are always reported relative to some window. The X context manager, which is accessible through X library functions, is used to associate mwm contexts and data with the window identifiers provided in events.
- Dispatch nonwidget events to the appropriate event handler and dispatch widget events using the Xt Intrinsic function XtDispatchEvent.
- Go back to the start of the event loop to get the next event.

Mouse Event Processing. Much of the behavior of the window manager interface is based on how mouse events are processed. Mwm divides mouse event processing into two categories: mutable behavior event processing and immutable behavior event processing.

Immutable behavior is built into mwm and is associated with the direct manipulation features (title bar, resize handles, etc.) of window frames and icons. Each direct manipulation feature has its behavior encapsulated in mwm event processing. Button press-and-release events and mouse motion events that occur with a context corresponding to a direct manipulation feature are processed by the event handler for that feature.

Mutable behavior event processing is based on user specification of mouse event associations with window manager functions. For example, button three of the mouse can be associated with the minimize function such that whenever button three is pressed with the mouse pointer over any part of the client window or window frame, the window will be minimized.

Mwm maintains a table that associates mouse events with window manager functions, and it uses this table for deciding which window manager function to invoke.

Keyboard Input Focus Event Processing. The window with the keyboard input focus is known as the active window. What this means is that when a key is pressed, the input is applied to the window with the keyboard input focus. Moving the keyboard input focus between windows is an important window manager function.

Two behaviors are supported by mwm for setting the keyboard input focus: explicit selection and pointer-relative selection.* Explicit selection means that a specific window is designated to be the keyboard input focus window. Explicit selection of the input focus is Presentation Manager behavior. For pointer-relative selection, the window under the mouse pointer automatically becomes the keyboard input focus window. This behavior is favored by many technical users.

Very different event processing is needed to handle the two different keyboard input focus behaviors. Setting the keyboard input focus in pointer-relative mode is done using enter and leave window events. When the pointer enters a window frame, mwm receives an enter window event. Mwm responds by making a request to the X server to cause delivery of keyboard input to the client window. As long as the pointer remains over the window frame (or the client window), keyboard input will be delivered to the client window. This maintains the illusion that the window frame is just another part of the client window. When the pointer leaves the window frame, a leave window event is received. This is usually followed by an enter window event as the pointer enters the root window or another window frame. Mwm responds by resetting the keyboard input focus appropriately.

Event processing for explicit selection of the keyboard input focus primarily involves button press and key press events as opposed to enter and leave window events. When a button press event is received by mwm and the context is a client window that does not have the keyboard input focus, mwm calls the X server to cause the delivery of keyboard input to the client window.

Mwm has to take care when it is processing button press events. Usually button events go to the window that is under the mouse pointer at the time the button is pressed or released. This means that if the pointer is over a client window and the button is pressed, the client window would normally get the button press event and mwm would not see an event. Mwm handles this by establishing a passive grab of the button event when it is generated in the client window context. A passive grab of the button causes the event to be delivered to mwm and not to the client window (see Fig. 10a). Mwm has effectively stolen a button event that would normally belong to the client window.

*Also known as tracked listener and real-estate driven.

This is not very friendly because the stolen event is often a mouse button 1 press event which, according to Presentation Manager, is also used to do selections of user interface components in the client window. Mwm redeems itself by making the button event available to the client. After mwm sets the keyboard input focus, it replays the button press, causing the event to be delivered to the client window (see Fig. 10b). Mouse event processing by the server is then allowed to continue, and mouse events that occur after the button press are delivered to the client window (if the client window is interested in the events). While a client window has the keyboard input focus, mwm turns off its passive grab request for a button press.

Interactive Pointer Tracking. A direct manipulation interface has to work hard to provide good feedback to the user. An example of this occurs during interactive moving or sizing of windows in mwm. Mwm draws a frame outline that tracks the new position or size of the window as the user moves the mouse around. Making this operate smoothly and efficiently requires some interesting event processing.

All X window managers provide a feedback mechanism like the one described above. Many do so by polling the position of the pointer (mouse cursor) and drawing a new outline (erasing the old) when the position changes. This has the advantage of keeping the window manager and the server synchronized, providing smooth behavior. The disadvantage is that the polling continues when the pointer is not moving, using up network bandwidth if the window manager is running remotely.

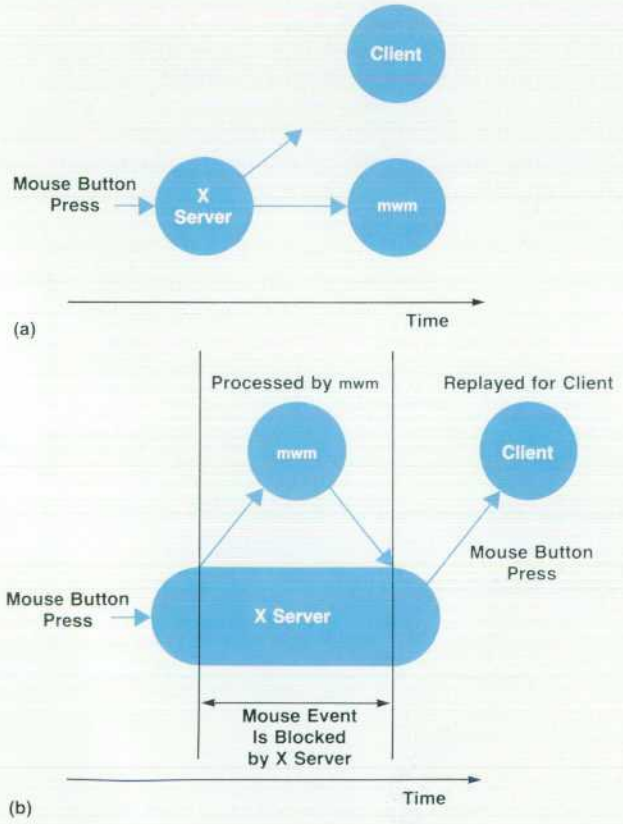


Fig. 10. *(a) Mouse input stolen by mwm using a passive grab. The short arrow indicates that the button event never makes it to the client. (b) Mouse input intercepted by mwm and replayed.*

The first implementation of hpwm, forerunner of mwm, departed from polling by requesting the server to report pointer motion events only when the pointer moved. Thus the drawback of polling was avoided. However, when the pointer moves, a large number of events must be processed. This was not a problem on medium-to-high-performance workstations that could keep up with the flood of events, but a problem did occur on low-performance machines, particularly X terminals. The time to process each motion event was longer than the time to generate a new one, causing the user to observe a window outline that would fall behind the motion of the pointer.

The solution to the problem, implemented in mwm (and a later hpwm), is to request the X server to send pointer motion hints, which are a special type of pointer motion event. In this mode of operation, the X server only sends pointer motion hints in conjunction with certain other events, such as window exit and entry. The X server also sends a pointer motion hint when the pointer moves from the last position queried by mwm. Each time a pointer motion hint is received, mwm acknowledges it by querying the position of the pointer. It then moves the pointer outline based on the values returned by the query. Tracking the pointer position with pointer motion hints is more expensive than polling when the pointer is moving, but it avoids the polling burden when the pointer is not moving.

Adopting a Client Window

Adopting a window refers to the process that mwm goes through when it initially encounters a window that it does not yet manage. This happens with the set of client windows that are on the display before mwm is started, as well as with clients that are started after mwm is already running. For each window that it adopts, mwm collects information from the client and the resource data base that affects the appearance of the window border, the placement of the window on the screen, and the window's behavior in response to user actions.

Communication between an X client and an X window manager occurs through events and properties (special information associated with a window). Among the events that are processed by mwm are those that begin or terminate management of X clients. The properties allow the client to indicate placement, decoration, and behavior information.

Mwm becomes aware of a new client when it receives the client's redirected request to display (or map) its top-level window. Mwm responds to this event by:

- Examining several client window properties
- Constructing a window frame and icon for the window
- Reparenting the client window to the mwm window frame
- Placing the client window on the display.

Several properties are used in this client-window manager communication. Some are listed in Table II.

The initial position and size of a window can be set either programmatically or interactively by the user. This information is passed to mwm in the WM_NORMAL_HINTS property of the client window. The value of this property is what determines how mwm places the window. Mwm will let the user place the window interactively if mwm's interac-

Table II
Window Properties

Property	Use
_MOTIF_WM_HINTS	Frame decoration and function preferences
_MOTIF_WM_MENU	Modify window menu
WM_CLASS	Client class for fetching resources
WM_HINTS	Icon image
WM_ICON_NAME	Icon name
WM_ICON_SIZE	Icon sizes preferred by window manager
WM_NAME	Client window name
WM_NORMAL_HINTS	Window position and size
WM_PROTOCOLS	Client-window manager communication
WM_STATE	Window manager state for client
WM_TRANSIENT_FOR	Secondary window indicator

tive placement is enabled and if the initial position has been set programmatically. However, if the initial position has been set by the user (e.g., via a command-line option), interactive placement will not be done even if it is enabled.

Mwm manages windows, not clients. If a client uses several top-level windows, mwm will treat them all equally even though they may have different purposes. However, a client may indicate a secondary top-level window, such as a dialog box, by placing the WM_TRANSIENT_FOR property on it. Mwm will decorate a window with this property differently, using a separate decoration resource for secondary windows. Mwm will not place a secondary window interactively.

In addition to reading properties when the window is adopted, mwm tracks changes to some of the properties while the client is running. The client may change the name displayed in the title bar by changing the WM_NAME property. Similarly, the client may change the name displayed in the icon by changing the WM_ICON_NAME property. Window geometry (i.e., size, position, and resize increment) changes are also tracked in WM_NORMAL_HINTS to make sure that resize units are properly reported. For example, a terminal emulator may resize its window to display function keys, but the number of text rows reported as the window size should not change.

Menu Handling

Mwm supports both client-specific and general-application menus. The contents of client-specific window menus and general-application menus can be specified by the user. The user can also specify the button or key event that causes a menu to be posted and the context for the event (e.g., post a utility menu when mouse button 1 is pressed with the pointer in the title area of the window frame). Everything that can be done with menus using a mouse can also be done using a keyboard.

Presentation Manager behavior includes a client-specific window menu that is posted using the window menu button on the window frame. The window menu is like a pull-down menu. It appears below the window menu button when the pointer is moved over the window menu button and the selection button (on the mouse) is pressed. A selection is made by dragging the pointer to a menu item and releasing the selection button. A client-specific window menu can also be posted by a button or key event in the client icon context.

To the user it may seem that mwm supports a large number of menus. This is because each client window has a menu that is posted from the window menu button, and each client icon has a menu that can be posted with a key press (typically **Shift ESC**). There are also menus that are commonly used to start clients and to perform various window management functions (e.g., change the stacking order of client windows). The heavy use of menus, combined with the relatively high performance cost of making menus, led to the design of a menu cache for mwm. A menu cache is possible because many menus have the same menu items. Also, the flexibility of the OSF/Motif menu widget allowed mwm to use a pop-up menu type for all the menus. Mwm uses the OSF/Motif pop-up menu type to implement window menus and simulates pull-down menu behavior when a menu is posted using the window menu button.

Mwm keeps a list of menu specifications. When mwm makes a menu it starts with a particular menu pane specification. The workspace menu entries given in Fig. 8 illustrate a menu specification. Other menus can be specified to cascade from the starting menu (see Fig. 11). When a menu is made, an association is made between the menu and the initial menu specification. Subsequent calls to make the same menu will return the menu that is already built. The key to making this work is the capability of mwm to adjust the characteristics of the menu dynamically so that the menu is set up correctly for the context in which it is posted.

Mwm adjusts the following menu characteristics:

- The active and inactive appearance and behavior of menu items are matched to the context in which the menu is posted. Menu items that are not applicable in a particular context are grayed out and are not selectable. For example, a menu item that minimizes a client win-

(continued on page 24)

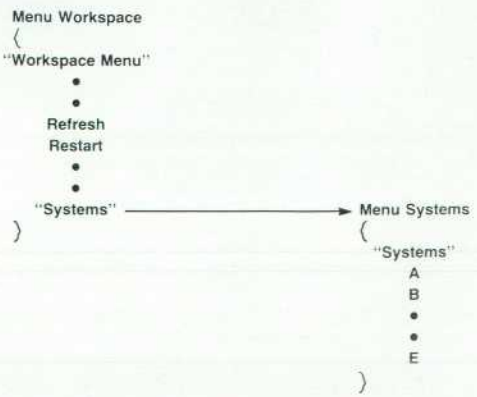


Fig. 11. *The* Systems *menu is cascaded from the* Workspace *menu because of the entry in* Workspace *that calls the* Systems *menu.*

Interclient Communication Conventions

The X Window System Version 11 (X) was designed to be a platform on which windowed application environments could be built. It provides a basic set of mechanisms for building these environments and does not impose any particular user interface behavior. With a minimal set of constraints on behavior an X-based application (X client) may be usable in isolation but unable to coexist with other X clients. Coexistence entails civilized sharing of limited resources (e.g., the physical color map) and the use of standard mechanisms for exchanging information (e.g., cutting and pasting text). A window manager can enforce coexistence of X clients in areas such as the use of screen space and keyboard input, but even a window manager does not have absolute power to maintain order. An unfriendly X client could grab the X server and prevent other X clients from getting input or doing output.

Inter-Client Communications Conventions Manual

Early in the development of X, representatives from the different companies working on or with X started meeting to address the problem of X client coexistence. This group has been officially sanctioned by the X Consortium to develop interclient communication conventions. The conventions that have been developed are documented in the *Inter-Client Communication Conventions Manual* (ICCCM). ICCCM compliance has become a key design criterion for X clients. The development of the ICCCM is ongoing and the general goals that shape this development include:

- Improving client coexistence in areas of potential contention.
- Tracking the evolution of the X Window System and X clients and providing new conventions that are generally applicable.
- Adding X Window System support for new conventions.
- Ensuring that all ICCCM changes are backwards compatible. This means that all previously defined conventions are maintained, and old conventions are changed only when they clearly cause incorrect behavior.

Client-To-Window-Manager Communication

Many conventions are documented in the ICCCM. However, the conventions that have received the most attention by X client developers have been those dealing with client-to-window-manager communication. A key goal of the mwm design was ICCCM compliance. X clients that are ICCCM compliant can coexist in a predictable manner with mwm and with each other. Window properties are one of the X mechanisms for client-to-window-manager communication. A window property is a collection of information of a particular type that is associated with a window. Clients associate, by convention, several properties with their windows to communicate with the window manager. Noteworthy examples of properties that are used for client-to-window-manager communication are WM_NORMAL_HINTS and WM_PROTOCOLS. The WM_NORMAL_HINTS property deals with window size and positioning, and WM_PROTOCOLS deals with public or private window manager communication protocols.

Client Window Size And Position

The WM_ NORMAL_HINTS property is used by a client to give a hint to the window manager on how the client window should be positioned on the screen and what its size should be. The window manager enforces how a client is positioned and sized on the screen. Some window managers may enforce a policy where all client windows are tiled on the screen (displayed without overlapping), or where windows are not allowed to be displayed with part of the window off the edge of the screen. The WM_NORMAL_HINTS property provides the window manager with a starting point from which it then applies the screen layout policies. An ICCCM compliant window manager can ignore some or all of the information contained in the WM_NORMAL_HINTS property. An X client should be designed to be robust enough to work in environments where this is the case. This demand on X clients is based on an ICCCM principle that the user is in control of the user interface, not the X clients.

The WM_NORMAL_HINTS property contains the following pieces of information:

- Minimum and maximum window sizes. These are reasonable minimum and maximum sizes for the window. Mwm uses the maximum size when a window is maximized.
- Base and increment window sizes. The overall window size is the base size plus some number of increments. Mwm adjusts a window size to meet this constraint when the window is initially placed on the screen or following resizing by the user. This is especially useful when the window is associated with a terminal emulator X client. The base window size usually includes the height of the softkeys. The increments are set to be equivalent to the height and width of one of the characters displayed in the terminal emulator window (terminal emulator X clients use fixed-size fonts in which all characters are the same size).
- Minimum and maximum window aspect ratios. The aspect ratios indicate allowable values for the ratio of the window width to the window height. For example, an X client can indicate that it would always like to be displayed in a square window (the aspect ratio is 1:1).
- Anchor point for window placement. The anchor point for placing a window allows an X client to specify how the window position should be interpreted. This is useful in the case where a window manager adds a frame around the X client window and adjusts the position of the X client window on the screen. The X client can specify an anchor point such that a corner or side of the X client window, including the window manager frame, is placed at a particular absolute location on the screen.

In general, mwm uses the WM_NORMAL_HINTS information with little or no change to place an X client window. Adjustments are only made if the user requests some refinement of the mwm window placement policy (e.g., the user requests that windows be interactively placed when they are first displayed). In placing an X client window on the display, mwm first determines a desirable window size, which is usually the window size specified by the X client. Mwm then retrieves the WM_NORMAL_HINTS property.

The processing of the WM_NORMAL_HINTS property varies based on the version of the ICCCM that the associated client implements. Mwm uses the size of the property in figuring out which version of the ICCCM to use. This allows mwm to be backwards compatible in complying with the ICCCM.

Client and Window Manager Protocols

The WM_PROTOCOLS property is used by an X client to indicate interest in public or private window-manager-to-client communication protocols. In general, these protocols are used to inform an X client of some window manager action that has occurred or is about to occur (e.g., the window system is about to be terminated). Public protocols are registered by the X Consortium, specified in the ICCCM, and supported by most, if not all, ICCCM compliant window managers. Private protocols are specific to a

particular window manager. Private protocols that have high utility and widespread acceptance by X client developers usually become public protocols.

The WM_PROTOCOLS property is formatted as a list of protocol identifiers. Many window managers, including mwm, keep track of X client changes to the property. This allows an X client to participate only in those protocols that it requires at a particular time. The WM_PROTOCOLS list can accommodate any number and mix of public and private protocols.

The WM_DELETE_WINDOW protocol is a commonly used public protocol. This protocol is used to inform X clients that a request has been made (probably by the user) to get rid of an X client window. This protocol is used by window managers to implement a clear and consistent user interface for getting rid of windows. Typically, deleting a window also includes deleting the X client that is associated with the window. Mwm uses the WM_DELETE_WINDOW protocol to close a window. The mwm close function can be accessed from the standard window menu that is posted by pressing the window menu button in the client window frame. If the close function is invoked on a client that does not participate in the WM_DELETE_WINDOW protocol, mwm uses the X request XKillClient to get rid of the window and terminate the client. In this case the client finds out that it has been terminated but cannot prevent or delay the termination. This is not appropriate for clients that would like to interact with the user on termination, or clients that have multiple windows that can be independently terminated. If a client does participate in the WM_DELETE_WINDOW protocol, mwm sends a termination request message to the client indicating that the window is to be terminated. It is then up to the client to determine how to deal with the window, because mwm takes no further action. Well-behaved clients immediately remove the window from the screen or prompt the user for confirmation.

(continued from page 22)

dow is grayed out if the menu is posted in the icon context.

- A menu is placed in keyboard traversal mode to allow keyboard manipulation of the menu. However, if a menu is not posted using a key press, the menu is not placed in traversal mode.
- A menu is configured to have particular key and button events select a menu item and unpost the menu.
- A menu is posted at a particular screen position (e.g., below the window menu button in a window frame).
- Mwm keeps track of the currently configured characteristics of a menu and does the minimal amount of adjustment that is necessary before posting the menu.

Component Graphics

The window frame provided by mwm for decorating client windows consists of a number of components representing different window management functions. The functionality and layout of the components are the same as in Presentation Manager. However, mwm enhances the appearance of the frame by adding the 3D appearance.

It is important for mwm to be as fast as possible to implement a good direct manipulation interface. The two principal things that were done to speed up the graphics rendering were to minimize the number of X protocol requests to draw the frame, and to do all the drawing to one window.

A fully configured mwm window frame consists of a border and a title bar. The border is divided into eight resize handles. The title bar is divided into boxes (or gadgets) for the system menu, the title text, and the minimize and maximize functions. The height of the title bar and the drawings inside the gadgets are scaled to match the height of the font used for the text in the title bar.

A frame with the 3D look may have as many as four colors displayed at once. These are the background, the foreground (title text), and the top shadow and bottom shadow colors (see Fig. 12). The background color makes up the majority of the color visible in a frame. Mwm sets the background color of a frame by setting the background attribute of the frame window. The background of all the frame components is set in one X graphics call. Once this attribute is set, the X server takes care of painting the background of the window in response to exposure events.

Graphic contexts are used to store much of the information required by the X graphics routines. This includes items such as colors, line styles, and clip regions. Mwm creates several graphic contexts for use in drawing the frame. These graphic contexts may differ in foreground color and fill tile. They are created when the window manager starts up and are used for all the window frames. When mwm draws a differently colored part of the frame, it passes a different graphic context to the graphics drawing routine.

The title text is usually drawn in one XDrawString call using the graphic context containing the foreground color. If the text is too long for the available space, then the text is truncated by setting a clip rectangle into the graphic context before calling XDrawString.

The remainder of the frame is made of the top and bottom shadow colors. This includes the outer 3D shadowing, the separations between the resize handles, the edges of the title bar buttons, and the images inside the system minimize and maximize buttons. This drawing is done with only two calls to XFillRectangles.

XFillRectangles takes, among its arguments, a list of rectangles and a graphics context. Mwm generates two lists of rectangles for top and bottom shadows when a frame is built. This occurs whenever a frame is needed for a new window, or when a window has been resized. To make this task simpler to code, two helper routines were constructed to add data to an existing pair of lists. One routine adds the top and bottom shadows to construct rectangular features. The other routine adds the top and bottom shadows to construct the corner resize handles. The shadowing for the entire frame is constructed out of multiple calls to these two routines.

Mwm always redraws the entire frame in response to an exposure event. In the best case, this takes three X graphics calls for drawing the text and the top and bottom shadows. If the text is clipped, then two more X calls are required for setting and clearing the clip rectangles. If the background color of the frame changes, then two additional calls are needed to set the frame window background attribute and clear the window to the new background. The common case of setting or clearing the focus indication on a window frame takes five X graphic calls.

The performance of this frame redrawing algorithm has been adequate. A possible optimization would make the exposure event handling smarter by only drawing those

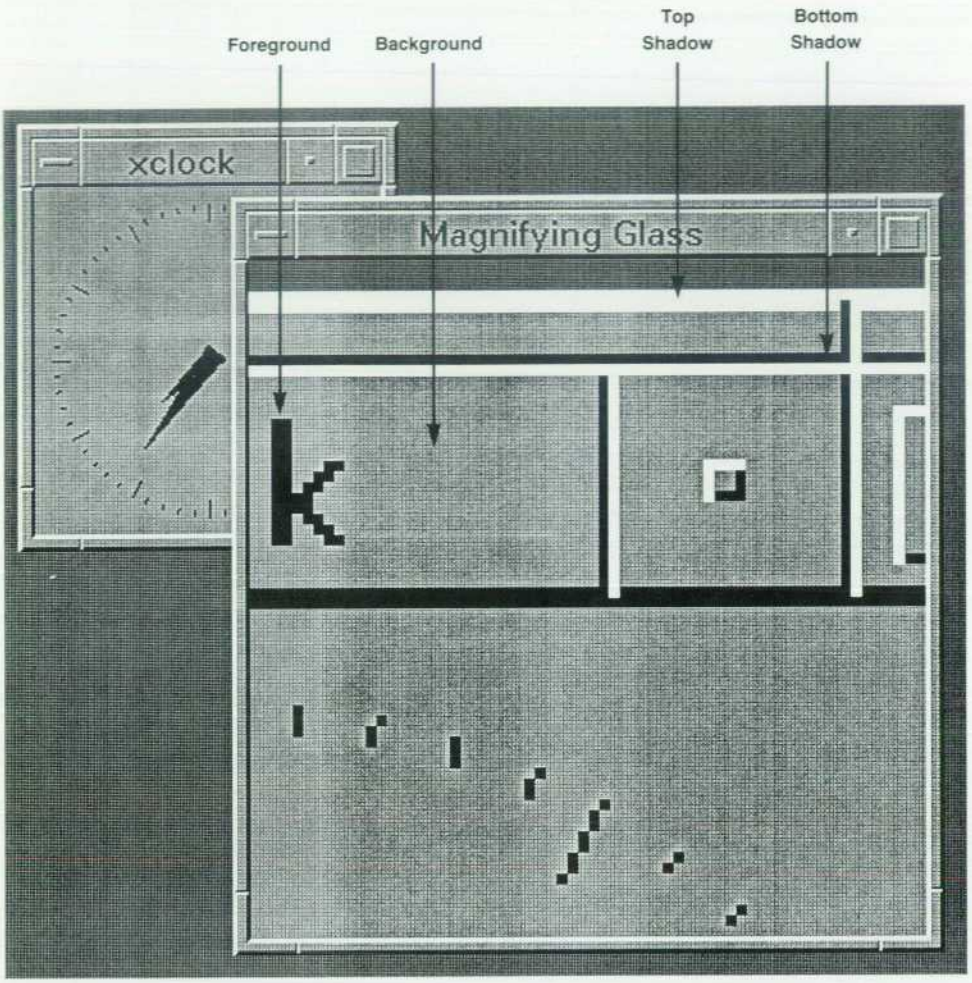


Fig. 12. *The four colors involved in achieving the 3D look for a frame.*

areas that need to be drawn. This would require either generating a new list of rectangles for the exposed region, or picking out the affected rectangles from the list of rectangles for the whole frame. Since X drawing calls map into X protocol requests (which can be computationally expensive), the optimization would have to avoid generating more X protocol requests than the approach taken above.

Testing a Window Manager

Mwm has a programmatic interface that is used by clients and an interactive interface for users. The testing of mwm needed to cover both of these interfaces. The approach to testing the programmatic interface involved writing a number of special-purpose clients that systematically generated all of the events that the mwm programmatic interface handles. These programs were run for each regression test as mwm progressed through its various development releases.

The testing of the interactive mwm interface required a much different approach. The interactive nature of the interface precluded the use of test programs. Testing could have been accomplished by developing test scripts that testers would follow for each regression cycle and each tested hardware configuration. However, this is an extremely tedious and expensive approach to testing.

Fortunately the Xtm (X test monitor) testing tool was developed for testing interactive X clients. Xtm is based on the record-replay software testing technique.[4,5,6] In this technique human interactions with the system are recorded in a file and replayed later for regression testing. Xtm records all mouse and keyboard interactions and saves them in an interactive test script file. The tester can at any time save snapshots of all or part of the screen. For regression testing the Xtm interactive test scripts can be replayed. Xtm compares the saved screen images with the replay screen images and flags any differences. A tester only has to spend time recording the interactive test script and checking the results of the automated regression tests. Use of Xtm also allowed repeatable testing. A user could not be expected to move a pointer in exactly the same way or remember what a screen looked like down to a single pixel each time a test script is followed.

Mwm testing also benefited from the wide distribution it received through OSF. Mwm was made available to a sizable number of people at OSF member companies including HP. These users had a variety of software and hardware environments as well as different patterns of use and expectations from a user interface. Their input provided a useful adjunct to the testing done using Xtm.

Acknowledgments

We would like to acknowledge all those that helped with

the development of mwm. First are the other members of the mwm (and hpwm) team: project manager Karen Helt, Fred Handloser, and Paul McClellan. Shizunori Kobara's help was instrumental in designing a good-looking window frame. Finally, we would like to acknowledge the Open Software Foundation for its vision in promoting industry standards and for picking hpwm as the basis for the OSF/Motif window manager.

References

1. F. E. Hall and J. B. Byers, "X: A Window System Standard for Distributed Computing Environments," *Hewlett-Packard Journal*, Vol. 39, no. 5, October 1988, pp. 46-50.
2. *Hewlett-Packard Journal*, Vol. 40, no. 6, December 1989, pp. 6-46.
3. *Ibid*, pp. 33-38.
4. C.D. Fuget and B.J. Scott, "Tools for Automating Software Test Package Execution," *Hewlett-Packard Journal*, Vol. 37, no. 3.
5. K.A. Olsson and M. Bergman, "A Virtual User Simulation Utility," *Hewlett-Packard Journal, Vol. 39, no. 2, April 1988, pp. 48-53.*
6. M.R. Tuttle and D. Low, "Videoscope: A Nonintrusive Test Tool for Personal Computers," *Hewlett-Packard Journal*, Vol. 40, no. 3, June 1989, pp. 58-64.

Programming with HP OSF/Motif Widgets

The HP OSF/Motif widget library makes it easy for a developer to create applications with a graphical user interface that has a consistent appearance and behavior.

by Donald L. McMinds and Benjamin J. Ellsworth

THE X WINDOW SYSTEM (usually referred to simply as X) is widely recognized as the industry standard window system for UNIX-system-based workstations X's greatest attribute is the fact that applications written for one vendor's platform will run on almost any other platform without modification. X provides a root window within which smaller windows can be displayed. A number of applications can be run simultaneously and each application can have any number of windows. A workstation screen with a typical assortment of windows is shown in Fig. 2 on page 7.

The X Window System is composed of a set of library functions known as Xlib. Xlib is the heart of X and it can be compared to an assembly language. Like assembly language programming, creating a user interface using only Xlib can be tedious and cumbersome (an example of Xlib programming is provided later in the article). To overcome this problem, the X designers created a second set of functions called the Xt Intrinsics or the X toolkit. The Xt Intrinsics use the Xlib functions to provide a higher-level set of functions that make user interface programming easier. The next library in the hierarchy, widgets, was designed to use both Xlib and the Xt Intrinsics to relieve the programmer of much of the extra work required to use these functions. The relationship between the two sets of X functions (Xlib and Xt Intrinsics) and widgets is much the same as the relationship between a computer's assembly language and a high-level language such as C.

The user communicates with X through the window manager. Depending on the request, the window manager communicates directly with one of the lower-level components in the hierarchy shown in Fig. 1 or with the X client. The window manager is really just another X client (although admittedly a very special one).

The box on page 27 provides more information about the evolution and development of widgets. This article describes some characteristics of the HP OSF/Motif widget library and shows how to write a program using this library.

Widgets

Widgets provide a base upon which the programmer can build an application user interface that has a consistent behavior and appearance. Widgets have a hierarchical class structure. Each widget has some resources of its own and

(continued on page 29)

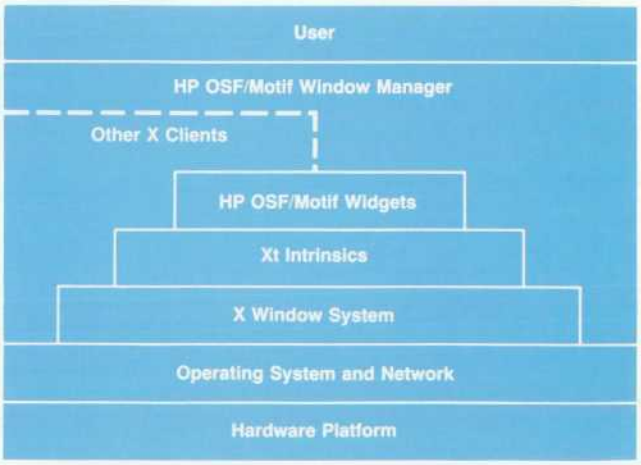


Fig. 1. *The X Window System and other components in the OSF/Motif environment. The dashed line indicates that other X clients can either be managed by the window manager or run independently of the window manager.*

The Evolution of Widgets

The development and acceptance of any new technology in the software industry as a standard is an evolutionary process that is driven by such things as competition, new technologies, and the desire for interoperability over a wide variety of hardware platforms. This is the case with HP's OSF/Motif widget toolkit. This toolkit and the Xt Intrinsics were developed to provide a standard set of tools for implementing user interfaces for UNIX-system-based systems running in the X Window System environment.

Fig. 1 shows the family tree for widgets in relation to the different versions of the Xt Intrinsics used to implement them. The Xt Intrinsics are the foundation upon which many user interface toolkits that run in the X Window System have been developed. The Xt Intrinsics began as the result of a collaboration between HP and Digital Equipment Corporation in late 1986 and early 1987. At the end of this period, the Xt Intrinsics were contributed to the X Consortium.* The X Consortium accepted the Xt Intrinsics as a nonexclusive standard for the creation of user interface toolkits for the X Window System environment.

The first freely available set of software objects (widgets) based on these early intrinsics was done by Project Athena at Massachusetts Institute of Technology. The Athena widgets, because they were the first widgets and their development was not particularly profit motivated, had a few bugs and did not offer much in terms of functionality. The Athena objects provided only buttons, scroll bars, editable and noneditable text, and boxes to contain them. Perhaps more important than the functionality, the Athena widgets presented a basic model of interaction supported by the Xt Intrinsics.

HP's First Widgets

HP's first X Window System user interface toolset was called x-ray. Xray was written in the C language and tailored to run in version 10 of the X Window System. Although this was a good toolset by everyone's estimation, it was realized that tools built on a standard base such as the Xt Intrinsics would have a better chance of long-term success. Therefore, x-ray was abandoned and work began on the HP X widgets, or as they were eventually called, CXI (common X interface) widgets.

HP's experience with x-ray helped to determine the feature set necessary for a successful user interface toolkit. Although we knew what we wanted to provide the customer, we were novices in using Xt Intrinsics. To accelerate our code production, the Athena widget code was used as the basis for the first widgets.

Simple widgets were created using a combination of the functionality inherited from methods in the core class widgets and the features provided in existing widgets. For this reason, many HP widgets started as a copy of an existing simple widget (the Athena label widget and the CXI button widget were the most commonly used). The core class methods were then modified until the desired change in functionality was achieved.

For more complicated widgets, two approaches were used. Either a simple widget was repeatedly modified until the complex functionality was achieved (the title bar widget is an example of this), or a closely parallel Athena widget was reworked and debugged as necessary (text and paned widgets are examples of this).

The New Generation

The key features that differentiated the CXI widgets from the

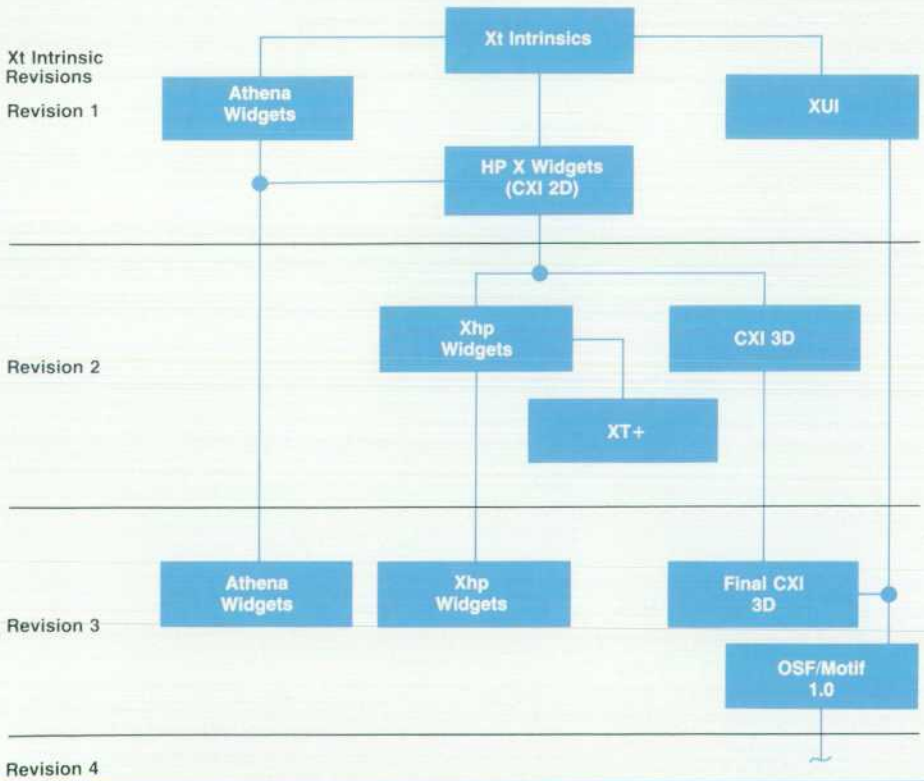


Fig. 1. *The widget family tree. Different improved versions of the Xt Intrinsics were used to implement different versions of the widget family.*

Athena widgets were keyboard traversal, a configurable menu system, and Presentation Manager behavior. Keyboard traversal is keyboard-only operation of the user interface without touching the mouse. As an example, consider a property sheet or a data entry form that contains numerous fill-in-the-blank fields. In the model presented by the Athena widgets, the user had to move the mouse every time there was a need to move to a different field. Touch typists complained that reaching for the mouse interrupted them to such a degree that they felt the interface was unusable. By offering keyboard traversal, the CXI widgets provided a way to navigate through the interface without ever having to leave the keyboard.

The most significant evolutionary feature of the CXI widgets, certainly in terms of product strategy, is the Presentation Manager behavior. Presentation Manager is itself a user interface that has evolved from Microsoft Windows and is characterized by a base set of graphical controls with consistent behavior. While the X Window System is primarily for the technical workstation market, much of the technical workstation market comes from users moving from personal computers to workstations. Often PC users are hesitant to move to technical workstations because the software environment appears foreign and therefore is presumed hostile. To make the move from personal computers to workstations easier, programs written using the CXI widgets present the user with an interface that behaves very much like Presentation Manager.

A New Dimension in Widgets

In 1988 we discovered that we needed to have a unique visual appearance for our widgets. This resulted in the development of widgets with a 3D appearance (see Fig. 2). This look was different enough to be considered proprietary. At this point we had two widget libraries: a 2D widgets library, which was contributed to the public domain, and a 3D widget library, which was proprietary. The 2D version of widgets, which became known as Xhp widgets, provided the basis for the XT+ toolkit from AT&T Bell Laboratories.

Soon after the release of the CXI 3D widget library, revision 3 of the Xt Intrinsics became available. Since there is always the urge to use the latest technology, a version of the CXI widgets was implemented using the latest Xt Intrinsics. One of the problems with the earlier version of CXI widgets was that it imposed the overhead of one window for every widget. Revision 3 of the Xt Intrinsics removed this problem by providing the ability to support windowless objects.

* The X Consortium is a group of companies that have joined together to promote standards and enhancements for the X Window System technology.

Open Standards

With the formation of the Open Software Foundation (see box on page 8), the role of CXI widgets took on a whole new meaning. In mid-1988 the newly formed Open Software Foundation requested members from the entire computer industry to submit proposals for a technology for creating an OSF user interface environment. After this industry-wide solicitation and review process, the OSF chose a hybrid of two proposals, both based on widget technology. HP was contracted to do the engineering work required to create this hybrid, the OSF/Motif widget set.

The OSF/Motif widget set is a combination of widget technology from HP's CXI and Digital Equipment Corporation's XUI. This hybrid provides the look and feel of CXI and the application programmer's interface of XUI. In late summer of 1989, version 1.0 of the OSF/Motif widget library was made available. All of the external features of the CXI widgets were improved upon and incorporated into the OSF/Motif widgets. The three-dimensional visual interaction clues were extended and made more consistent. Keyboard traversal was extended uniformly throughout the widget set and made almost entirely consistent with Presentation Manager. All other graphical controls such as menus and scroll bars were also made consistent with Presentation Manager behavior.

One of the most significant contributions XUI made to the OSF/Motif widget set was the addition of a rich dialog layer. XUI presents a large number of standard dialog boxes with a number of standard behaviors. These dialog boxes have been made visually consistent with CXI and behaviorally consistent with Presentation Manager.

A Process, Not a Result

OSF/Motif 1.0 is currently the top of the evolutionary chain for OSF/Motif widgets. However, the evolution of technology is much more a process than a result. Changes to the OSF/Motif widget set are already underway with the development of even better user interface components. More important, changes in the industry customer base and advances in such technologies as graphics hardware, object-oriented programming, cooperative work, and distributed networking will continue to change the environment of the software industry and to provide a fertile soil for widget evolution.

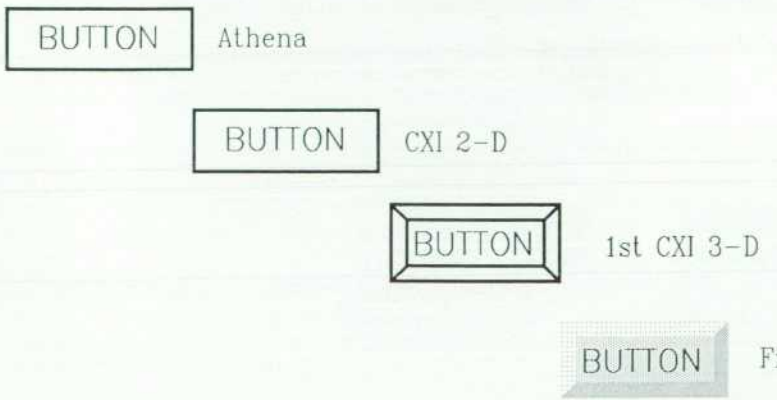


Fig. 2. *The evolution of the appearance of widgets from 2D to 3D with the beveled look.*

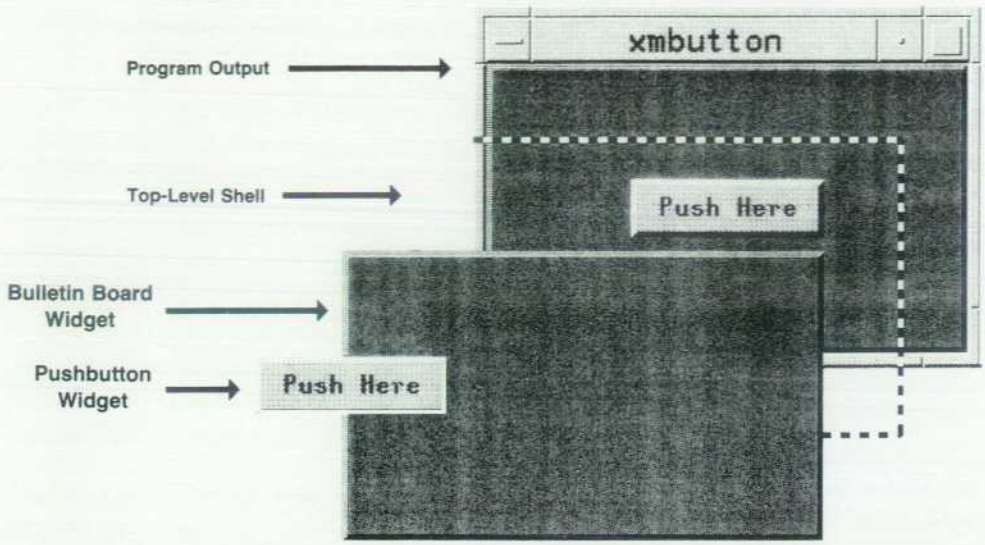


Fig. 2. *Exploded view of widgets in a program.*

can inherit resources from higher-class widgets. Resource simply means a data name or variable whose value affects some attribute of the widget. For example, there are resources that control the size, color, and behavior of widgets. Most widgets are visible in the form of a window. Examples of widgets include various types of buttons, scroll bars, menus, and dialog boxes through which information is exchanged. Some widgets cannot be made visible and are used as supporting superclasses. These widgets supply common resources for the other widgets.

Fig. 2 shows how widgets are combined to produce a window. The program output consists of a bulletin board widget and a pushbutton widget. The program that produces this output is described later. The top-level shell widget is an invisible widget that provides resources and communicates with the X server.[1] The frame around the visible output is provided by the OSF/Motif window manager and is not a part of the widget system.

A widget is composed of procedures and data structures that make use of the Xlib and Xt Intrinsics functions. The functionality provided by the few lines of code needed to create a widget on the screen can only be duplicated with many lines of code using Xlib and Xt Intrinsics functions. While widgets save coding time and make a program much easier to read and comprehend, the trade-off is that a widget program uses a lot more memory than an Xlib program. The program presented in this article uses nearly 680K bytes for the executable widget version and 140K bytes for the executable Xlib version.

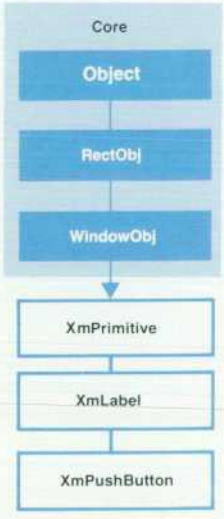


Fig. 3. *Basic widget hierarchy.*

Widget Hierarchy

The X toolkit defines widgets. To do so, it uses an object-based architecture that groups widgets into different classes. Each widget class has data structures and procedures (methods) that operate on the data. Widgets also define what data can be imported and exported to the application and what actions the widget supports. This set of data is referred to as the resources of the widget class. A widget is always an instance of some class. A pushbutton is a good example of a widget class that defines resources common to all pushbuttons. This class (XmPushButton) defines the methods for manipulating pushbuttons (e.g., resizing), and the set of data that can be imported and exported from any instance of the class. For example, the pushbutton class defines a resource called armColor. This resource controls the background color of the pushbutton when it is armed. This color can be modified in an instance of the pushbutton widget class by manipulating the state of the background resource of the affected widget instance.

The pushbutton widget class also defines the actions that pushbuttons support. Instances of the pushbutton class have three distinct states: armed, activated, and disarmed. By default a pushbutton is armed when the pointer is within the pushbutton area and mouse button 1 is pressed. A pushbutton is activated by first arming the pushbutton and then releasing the mouse button while the pointer is within the armed pushbutton area. After the mouse button is released, the pushbutton is disarmed. All of these behaviors are defined as part of the pushbutton widget class definition.

Any widget class can inherit some or all of the resources of another class. For example, the pushbutton class con-

tains resources belonging to the XmLabel, XmPrimitive, and Core classes, as well as its own resources. Fig. 3 shows the relationships among the basic widget classes. Label and pushbutton are primitive widgets. There are other primitive widgets besides label and pushbutton but they are omitted from Fig. 3 for clarity.

The basic widget class is the Core class. It contains resources that are inherited by all other classes. Each lower class can inherit some or all of the resources belonging to a higher class. The resources belonging to a given widget class can be determined by examining its man page in the *HP OSF/Motif Programmer's Reference Manual*.

A Widget Program

The following program illustrates the use of widgets in a program. The program consists of a pushbutton widget that is contained in a bulletin board widget. Selecting the pushbutton causes a message to be displayed on the terminal window and then the program terminates. The program is called xmbutton.c and the output from the program is shown in Fig. 4.

```
/**-------------------------------------------------------------------------------------
***
***    file:         xmbutton.c
***
***    project:      Motif Widgets example programs
***
***    description:  This program creates a PushButton widget.
***
***                  © Copyright 1989 by Open Software Foundation,
***                    Inc. All Rights Reserved.
***
***                  © Copyright 1989 by Hewlett-Packard Company.
***
***-------------------------------------------------------------------------------------*/

/* include header files */

#include <X11/Intrinsic.h>
#include <Xm/Xm.h>
#include <Xm/BulletinB.h>
#include <Xm/PushB.h>

/*    functions defined in this program    */

void main();
void activateCB();    /* Callback for the PushButton */

/*    global variables    */

char *btn_text;    /* button label pointer for compound string */

/*-------------------------------------------------------------------------------------
**    main - main logic for xmbutton.c program
*/
void main (argc,argv)
unsigned int argc;
char **argv;
{
        Widget          toplevel;       /*    Shell widget                */
        Widget          bboard;         /*BulletinBoard widget          */
        Widget          button;         /*PushButton widget             */
        Arg             args[10];       /*arg list                      */
        register        int n;          /*arg count                     */

/*    initialize the Xt Intrinsics    */
      toplevel = XtInitialize
          ("main", "XMbutton", NULL, NULL, &argc, argv);

/*    Create a bulletin board widget in which the pushbutton widget */
/*    can be placed */
          n = 0;
          bboard = XmCreateBulletinBoard (toplevel, "bboard",
          args, n);
/*    Manage the bulletin board widget */
          XtManageChild (bboard);
/* Create a compound string for the button text */
          btn_text = XmStringCreateLtoR
          ("Push Here", XmSTRING_DEFAULT_CHARSET);

/*    set up arglist */
          n = 0;
          XtSetArg (args[n], XmNlabelType, XmSTRING); n++;
          XtSetArg (args[n], XmNlabelString, btn_text); n++;
/*    create button    */
          button =XtCreateManagedWidget
          ("button", xmPushButtonWidgetClass,bboard, args, n);
/*    add callback    */
          XtAddCallback (button, XmNactivateCallback, activateCB,
              NULL);
/*    realize widgets */
      XtRealizeWidget (toplevel);
/*    process events */
      XtMainLoop ();
}
/*-------------------------------------------------------------------------------------
**        activateCB - callback for button
*/
void activateCB (w, client_data, call_data)
Widget w;               /*    widget id                 */
caddr_t client_data;    /* data from application        */
caddr_t call_data;      /* data from widget class       */
{
/*    print message, free compound string memory, and terminate
      program */
          printf ("PushButton selected.n");
          XtFree(btn_text);
          exit (0);
}
```

There are nine steps in writing widget programs. These steps are used regardless of the complexity of the program. The nine steps are:

1. Include the required header files.
2. Initialize the Xt intrinsics.
3. Add additional top-level windows, if needed.
4. Create argument lists for the widget.
5. Create the widget.
6. Add callback procedures.
7. Realize the widgets and loop.
8. Compile and link the program.
9. Create the defaults files.

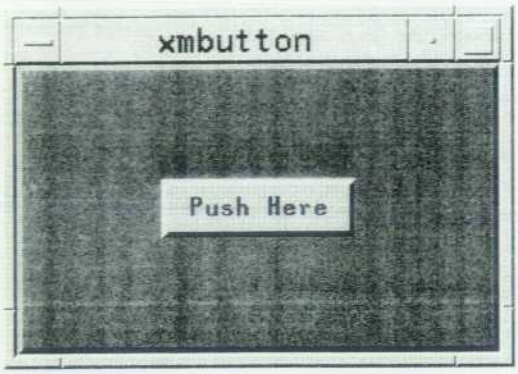


Fig. 4. *Output from the program* xmbutton.

Steps 4 through 6 are done for each widget included in the program. The next nine sections relate these steps to the program xmbutton.c.

Include Required Header Files

Some common variables and types of variables used by the widgets are defined in header files. The necessary header files are included at the beginning of the program. These header files are:

```
#include <stdio.h>
#include &<X11/Intrinsic.h>
#include &<Xm/Xm.h>
#include &<Xm/widget>
```

Replace widget with the name of the corresponding widget header file for each widget class used in the program. The include files for all widgets are found in the directory /usr/include/Xm. The header file name for each widget can be found in the synopsis section of each widget's man page. The order in which header files are placed is very important. This order must be:

- General header files, such <stdio.h>
- Xt Intrinsics header files, such as <X11/Intrinsic.h>
- Widget header files, beginning with <Xm.h> and including a header file for each widget class used in the program. The order within the widget header files is not critical.

For xmbutton.c, the include files are:

```
#include <X11/Intrinsic.h>
#include <Xm/Xm.h>
#include <Xm/BulletinB.h>
#include <Xm/PushB.h>
```

Note that there is an include file for the bulletin board widget and one for the pushbutton widget.

Initialize the Xt Intrinsics

The Xt Intrinsics must be initialized before any other calls are made to Xt Intrinsics functions. The most convenient method of accomplishing this is to use the function XtInitialize. This function establishes the connection to the X server, parses the command line that invoked the application, loads the resource data base, and creates a shell widget that will serve as the parent (or top level) widget for the application widgets. The call to XtInitialize in xmbutton.c is:

```
toplevel = Xtinitialize ("main", "XMbutton", NULL, NULL, &argc,argv);
```

The first two parameters, "main" and "XMbutton", are used to reference defaults files, which are ASCII files used by the system to set the values of widget resources. Defaults files are explained in more detail later in this article. The next two parameters are set to NULL since they are not used in this example. The last two parameters, &argc and argv, are the number of command-line parameters and the array in which they are stored.

The syntax for XtInitialize is:

```
XtInitialize (shell_name, app_class, options, num_options, argc, argv)
```

Type	Parameter
String	shell_name;
String	app-class;
XrmOptionDescRec	options[ ];
Cardinal	num_options;
Cardinal	*argc;
String	argv[ ];

where:

- shell_name specifies the name of the application shell widget instance.
- app_class specifies the class name of this application.
- options specifies how to parse the command line for any application-specific resources.
- num_options specifies the number of entries in the options list.
- argc specifies a pointer to the number of command line parameters.
- argv specifies the address of the command line parameters.

Adding Additional Top-Level Windows

XtInitialize can be executed only once in any program, so to create additional top-level widgets the functions XtCreateApplicationShell or XtAppCreateShell must be used. XtAppCreateShell allows the creation of a user-defined display while XtCreateApplicationShell uses the default display. XtCreateApplicationShell is from an earlier version of X, so it is probably better to use XtAppCreateShell. Xmbutton.c does not use a second top-level shell so neither of these functions appears in the program.

Creating Argument Lists and Widgets

The next step in the program is to create widgets. In most cases this involves setting widget resource name-value pairs into an argument list and then calling a create function for the widget. A name-value pair is a resource name and the value assigned to that resource. For example, the resource name labelString might have a string value "ABCD" assigned to it.

There are two methods to create widgets. The first method involves using convenience functions, and the second method involves using generic Xt Intrinsics. Conve-

nience functions, which are part of the widget library, are used to create a specific type of widget. For example, XmCreateBulletinBoard creates an instance of a bulletin board widget and XmCreatePushButton creates an instance of a pushbutton widget. Widgets created with Xt Intrinsics are automatically managed when they are created. However, widgets created with a convenience function must be managed with the function XtManageChild or XtManageChildren. Managing widgets this way provides some flexibility and saves time because a number of widgets can be created and managed all at once.

Using Convenience Functions. In the program xmbutton.c,the convenience function XmCreateBulletinBoard is used in the following lines of code to create a bulletin board widget.

```
/*   Create a bulletin board widget in which the pushbutton        */
/*   widget can be placed                                          */
        n = 0;
        bboard = XmCreateBulletinBoard(toplevel,
        "bboard", args,n);
/*   Manage the bulletin board widget                              */
        XtManageChild(bboard);
```

The variable n, which is used here to specify the number of name-value pairs in the argument list, is zero, indicating that there are no name-value pairs in the argument list. The function XtManageChild is used to manage the bulletin board widget bboard.

The syntax for XmCreateBulletinBoard is:

```
Widget = XmCreateBulletinBoard (parent, name, args, num_args)
```

Type	Parameters
Widget	parent;
String	name;
Arglist	args;
Cardinal	num_args;

where:

- parent specifies the parent widget of the newly created widget (toplevel in this example).
- name specifies the resource name for the created widget (bboard in this example). This name is used for retrieving resources and should not be the same as any other widget that is a child of the same parent.
- args specifies the argument list for resource values.
- num_args specifies the number of arguments in args.

The syntax for XtManageChild is:

```
XtManageChild (child);
```

Type	Parameter
widget	child

where the parameter child specifies the widget to be managed.

Using Xt Intrinsics. The pushbutton widget in xmbutton.c is created using the Xt Intrinsic XtCreateManagedWidget. The lines of code associated with creating the pushbutton widget are as follows.

```
/*   Create a compound string for the button text                  */
        btn_text = XmStringCreateLtoR
        ("Push Here", XmSTRING_DEFAULT_CHARSET);
/*   set up arglist                                                */
        n = 0;
        XSetArg(args[n], XmNlabelType, XmSTRING); n++;
        XSetArg(args[n], XmNlabelString, btn_text); n++;
/*   create button                                                 */
        button = XtCreateManagedWidget
        ("button", xmPushButtonWidgetClass, bboard, args,n);
```

The call to the widget library function XmStringCreate has nothing to do with creating a widget, but it does set up a compound string for the pushbutton label. A compound string is designed to allow any text to be displayed without having to resort to hard coding certain language dependent attributes. The variable btn_text is a pointer for the compound string "Push Here."

The Xt Intrinsic function XtSetArg is used to set up the argument list. It sets the the values for specified widget resources into an array that is subsequently accessed by the widget when it is created. The syntax for XtSetArg is:

```
XtSetArg (arg, name, value)
```

Type	Parameters
Arg	arg;
String	name;
XtArgVal	value;

where:

- arg specifies the name-value pair to be set.
- name specifies the name of the resource.
- value specifies whether the value of the resource will fit in a long integer (XtArgVal is defined to be of type long int in an Xt Intrinsics header file); otherwise, it specifies the address.

The intrinsic XtCreateManagedWidget creates a pushbutton widget that has the name button and the bulletin board widget bboard as its parent. Note that creating a widget merely creates the data structures associated with that widget. It does not make the widget visible on the screen.

The syntax for XtCreateManagedWidget is:

```
Widget = XtCreateManagedWidget (name, widget_class, parent, args,
                                num_args)
```

Type	Parameters
String	name;
WidgetClass	widget_class;
Widget	parent;
ArgList	args;
Cardinal	num_args;

The parameters name, parent, args, and num_args specify the same values as their counterparts in the convenience function XmCreateBulletinBoard. The parameter widget_class, which is of type WidgetClass, specifies the widget class pointer for the created widget.

Adding Callback Procedures

Callbacks are procedures that are executed when certain events occur within a widget. Events such as pressing a mouse button, pressing a certain key on the keyboard, or moving the cursor into or out of a window can trigger a callback procedure. Every widget has a callback list for each type of callback it supports. This list contains the callback procedures to be executed when a particular event occurs. For example, every widget supports an XtNdestroyCallback list. Each callback procedure in this list is executed before the widget is destroyed. Information on the callbacks supported by a given widget can be found in the man page for that widget and any supporting superclass widget that supplies resources to it. There are two steps involved in adding a callback procedure: writing the callback procedure and adding the callback procedure to the callback list.

Writing a Callback Procedure. A callback procedure returns no value and has three arguments:

- The widget for which the callback is applicable.
- Data passed to the callback procedure by the application.
- Data passed to the callback procedure by the widget.

In xmbutton.c, the callback procedure activateCB prints a message to the standard output (this is normally the terminal window from which the application was executed), frees the memory used by the compound string stored in the variable btn_text, and ends the program by executing the system exit procedure. The callback procedure is just like any routine or procedure except that it is called only when the event to which it is tied occurs. In xmbutton.c, the activateCB callback procedure is executed when mouse button 1 is pressed and released and the mouse pointer is located within the pushbutton window. It is the release of the button (an event known as Btn1Up) that causes the pushbutton to be activated and the callback procedure to be executed.

Adding the Callback Procedure to the Callback List. The callback procedure is added to a specific callback list that is owned by the widget. This is done by using the Xt Intrinsics function XtAddCallback. In xmbutton.c the callback procedure is added with the code segment:

```
/* add callback */
    XtAddCallback (button, XmNactivateCallback, activateCB, NULL);
```

The syntax for XtAddCallback is:

```
XtAddCallback (w, callback_list, callback, client_data)
```

Type	Parameters
widget	w;
String	callback_list;
XtCallbackProc	callback;
caddr_t	client_data;

where:

- w specifies the name of the widget to which the callback procedure is to be added.
- callback_list specifies the callback list within the widget to which the callback procedure is to be added.
- callback specifies the name of the callback procedure to be added.
- client_data specifies the client data to be passed to the callback when it is executed.

Note that the callback is added after the widget has been created with XtCreateManagedWidget. This is necessary because one of the parameters for XtAddCallback is the pushbutton widget known as button. An error would occur if a callback is added to a widget that does not exist.

Making the Widget Visible

Even though we have created a widget and added a callback procedure to one of its callback lists, if we were to compile and execute the program at this point, nothing would be visible on the screen. This is because we have not passed the essential information about our widgets to the Xt Intrinsics functions that actually display the widgets on the screen. This is accomplished by using the function XtRealizeWidget.

The final step in the program is the call to the Xt Intrinsics XtMainLoop. XtMainLoop is really an event loop. As events occur, this function dispatches them and then searches or waits for the next event to occur. In our simple example, only the one event Btn1Up has any meaning. The window shown in Fig. 4 will remain displayed indefinitely until the pushbutton is pressed by moving the mouse pointer into the button window and pressing and releasing mouse button 1.

In the program xmbutton.c, displaying the widget and looping are performed in the code segment:

```
/*  realize widgets */
    XtRealizeWidget (toplevel);
/*  process events */
    XtMainLoop ();
```

Notice that there is no exit in the main part of the program. Program termination is taken care of in the callback procedure activateCB.

Compiling and Linking

For compiling and linking the program xmbutton, one of the following command lines is used.

- For HP 9000 Series 300 computers use:

```
cc -O -Wc, -Nd4000 -Wc, -Ns4000 -Wc, -Nt5000 -DSYSV -o\
xmbutton xmbutton.c -lXm -lXt -lX11
```

- For HP 9000 Series 800 computers use:

```
cc -0 -DSYSV -o xmbutton xmbutton.c -lXm -lXt -lX11
```

The libraries Xm, Xt, and X11, which are linked into the program, must appear in the order shown. Xm is the widget library, Xt is the intrinsics library, and X11 is the Xlib functions.

Creating Defaults Files

In the example above, the values of certain widget resources have been set by means of argument lists within the program. Another method of setting resource values is by means of ASCII files called defaults files. These files are automatically read by the system before executing a program. There are two types of defaults files: an app-default file and a user-specific file called .Xdefaults.

App-Default File. This file is located in the directory /usr/lib/X11/app-defaults and supplies defaults for an entire class of applications. The class is specified in the call to XtInitialize.

For example, the call to XtInitialize in xmbutton.c specifies the application class XMbutton (by convention, application class names are the program name with the first two letters capitalized). The app-defaults file XMbutton contains resource values for certain widgets that are used in the program xmbutton.c. An example of such a file is shown below.

```
! XMbutton app-defaults file for Motif demo program xmbutton.c
! general appearance and behavior defaults
!
!
*fontList:                          hp8.8×16b
*shadowThickness                    3
!
!     BulletinBoard resources
!
*bboard.resizePolicy:               RESIZE_NONE
*bboard.height:                     150
*bboard.width:                      250
*bboard.background:                 sky blue
!
!     PushButton resources
!
*button.foreground:                 midnight blue
*button.background:                 goldenrod
*button.borderWidth:                0
*button.height:                     30
*button.width:                      100
*button.x:                          75
*button.y:                          60
!
```

XtInitialize uses the data contained in XMbutton to build a resource data base before the widget is actually created.

.Xdefaults File. This file can be created in each user's home directory to set resource values for any number of programs. Defaults found in this file override those in an app-default file and allow different users to specify different values for the same resources. For example, one user may prefer to have different background and foreground colors from the ones set in the app-default file. Note that the values in the .Xdefaults file only override the app-default values. They do not change them. Suppose you want to override the default background and foreground colors for both the bulletin board widget and the pushbutton widget in xmbutton.c. The .Xdefaults file shown below changes the background of the bulletin board to white, the background of the pushbutton to red, and the foreground of the pushbutton to white. Note that the colors are only changed for the program xmbutton.

```
xmbutton*button.foreground:    white
xmbutton*button.background:    red
xmbutton*bboard.background:    white
```

The colors can be changed so that the background and foreground colors are the same for every widget in xmbutton with this .Xdefaults file:

```
xmbutton.foreground:           white
xmbutton.background:           red
```

The order of precedence for setting values in widget resources is:

- The app-defaults files
- The .Xdefaults file
- The values that are set in the program.

This means that values set in an app-default file can be overridden by an .Xdefaults file and values set in an .Xdefaults file can be overridden by values set in the program.

An Equivalent Xlib Program

A program using Xlib and Xt Intrinsics functions rather than widgets to produce our pushbutton and bulletin board combination would take nearly four pages of code. For this reason, only a portion of the equivalent Xlib program is presented here.

The previous section discussed defaults files as a means of setting the value of a widget's resources. If widgets are not used then defaults files cannot be used. For example, in xmbutton.c we used a single line in an .Xdefaults file to set the background color of the pushbutton widget:

```
Button.background:   goldenrod
```

Similar entries were made in the defaults files to set values for other resources. Without widgets, setting up resource values requires several lines of code in the application program. For example, to set the background color in an Xlib program, the color must first be allocated and then used as one of the arguments to the function XCreateSimpleWindow. The following code segment shows what this involves:

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>
#include <stdio.h>

#define OFF 0
#define ON 1

#define FALSE 0
#define TRUE 1

Display *dpy;
Window parent, child, root;
GC shadow_gc, text_gc;
unsigned long sky_blue, lt_blue, goldenrod, wheat, dk_grey,
white, black;
int screen;
XFontStruct *fptr;
Colormap cmap;

main(argc, argv)
int argc;
char *argv[];
{
  XColor hard_def, exact_def;
  XSizeHints xsh;
  XEvent event;
  int state = OFF, tx, ty, status, pressed = FALSE;
/* Open the display */

  if ((dpy = XOpenDisplay(NULL)) == NULL) {
```

```
    fprintf(stderr, "%s: Cannot open DISPLAYn", argv[0]);
    exit(1);
  }
/* Set parameter values */
  screen = DefaultScreen(dpy);
  root = RootWindow(dpy, screen);
  cmap = DefaultColormap(dpy, screen);
  white = WhitePixel(dpy ,screen);
  black = BlackPixel(dpy ,screen);
/* Create graphics contexts */

  text_gc = XCreateGC(dpy, root, 0, 0);
  shadow_gc = XCreateGC(dpy, root, 0, 0);
/* Allocate colors */

  status = XAllocNamedColor(dpy, cmap, "SkyBlue",
      &hard_def, &exact_def); if (!status)  {
    fprintf(stderr, "%s: Cannot allocate the neccessary colorsn",
      argv[0]); exit(1);
   }
  else
      sky_blue = hard_def.pixel;
  status = XAllocNamedColor(dpy, cmap, "LightBlue",
      &hard_def, &exact_def); if ( status)  {
 fprintf(stderr, "%s: Cannot allocate the neccessary colorsn",
  argv[0]); exit(1);
  }
else
    lt_blue = hard_def.pixel;
/* Load font */
fptr = XLoadQueryFont(dpy, "hp8.8x16b");
if (fptr == NULL)  {
 fprintf(stderr, "%s: Cannot load the neccessary fontn", argv[0]);
  exit(1);
  }
XSetFont(dpy,text_gc,fptr->fid);
/* Geometry calculations */
ty = (30-fptr->ascent-fptr->descent)/2 + fptr->ascent;
tx = (100 - XTextWidth(fptr, "Push Here", 9)) / 2;

xsh.x = 0; xsh.width = 250;
xsh.y = 0; xsh.height = 150;
/* Create bulletin board equivalent */

parent = XCreateSimpleWindow(dpy, root, xsh.x, xsh.x, xsh.width,
                             xsh.height, 2, lt_blue, sky_blue);
  .
  .
  .
}
```

Although there is more to this code than just allocating colors, it is obvious that there is a lot more involved than is required when using widgets.

References

1. K.H. Bronstein et al, "System Design for Compatibility of a High-Performance Graphics Library and the X Window System," *Hewlett-Packard Journal*, Vol. 40, no. 6, December 1989, pp. 6-12.

The HP SoftBench Environment: An Architecture for a New Generation of Software Tools

The HP SoftBench product improves programmer productivity by integrating software development tools into a single unified environment, allowing the program developer to concentrate on tasks rather than tools.

by Martin R. Cagan

THE HP SOFTBENCH PRODUCT is an integrated software development environment designed to facilitate rapid, interactive program construction, test, and maintenance in a distributed computing environment.

The HP SoftBench environment provides an architecture for integrating various CASE (computer-aided software engineering) tools. Many of the tools most often needed—program editor, static analyzer, program debugger, program builder, and mail—are included in the HP SoftBench product. Another HP SoftBench component, the HP Encapsulator, makes it possible to integrate other existing tools into the HP SoftBench environment and to tailor the environment to a specific software development process. Fig. 1 illustrates the HP SoftBench user interface.

This article describes the HP SoftBench tool integration architecture. The HP SoftBench program editor, static analyzer, program debugger, program builder, and mail are described in the article on page 48. The HP Encapsulator is described in the article on page 59.

Design Objectives

The overall goal of the HP SoftBench product is to improve the productivity of programmers doing software development, testing, and maintenance. To achieve this goal, the following objectives were defined for the tool integration architecture:

(continued on page 38)

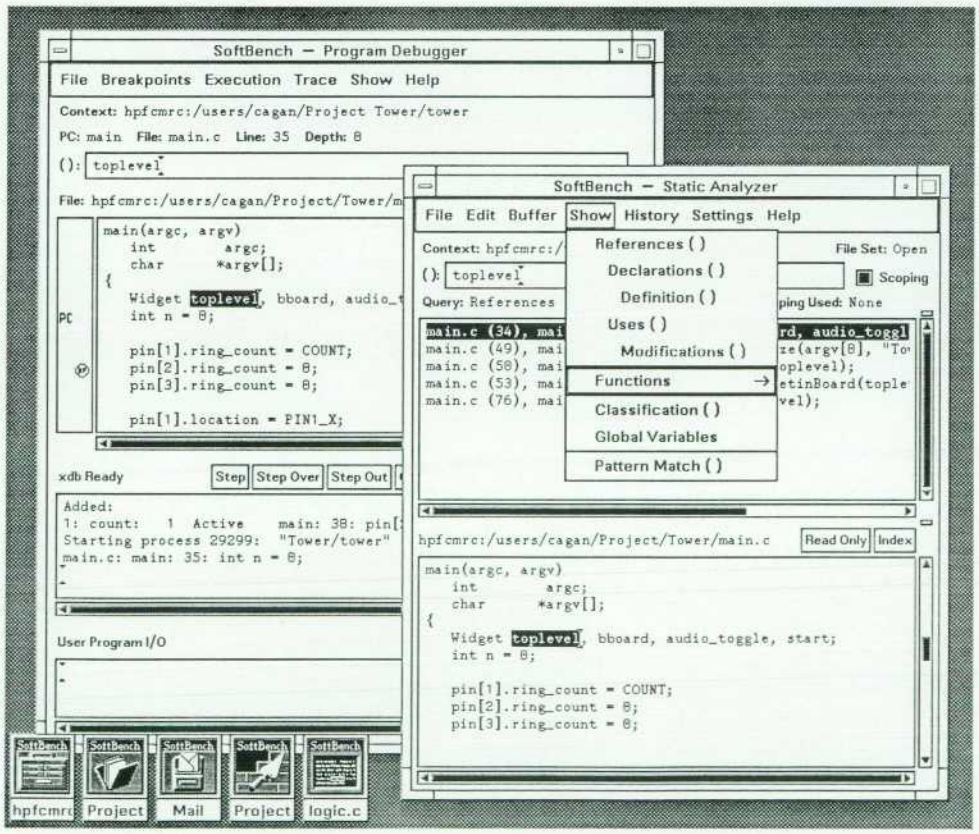


Fig. 1. *Typical HP SoftBench user interface.*

Architectural Support for Automated Testing

The importance of software testing does not have to be argued anymore.[1] There also exists a relative wealth of sources describing various aspects of software testing. Unfortunately, most of the published literature concentrates on elegant approaches to limited subproblems derived from traditional software (that is, batch-oriented input/output). Additionally, the published body of knowledge almost completely ignores the issue of how the testing activity should fit modern project life cycles (reference 2 is a rare exception).

This section is about testing a large software system: the HP SoftBench product described in the accompanying article. The goal is to describe both the process and the various tools and utilities developed to exploit the architectural advances of the HP SoftBench product to support the testing process.

The problem of testing a system such as the HP SoftBench environment is difficult and therefore interesting. The problem has the following major attributes:

- Development of the HP SoftBench system followed the spiral life cycle[3] which, because of its crucial aspect of rapid prototyping, presents a real challenge for formal testing.
- The formal testing activity started early in the project life cycle and closely tracked the project development.
- The system being tested consists of several tightly integrated tools and is event-driven.
- The system has a sophisticated user interface (window-based and mouse-driven).
- Black-box tests had to be automated (which in the case of the user interface meant developing an "automatic user").
- The testing proceeded along an unorthodox path—from black-box testing, through "grey-box testing" (driven by branch flow and complexity analysis), to white-box testing.

Automatic Regression Testing

Traditional testing methods focus on exercising and testing programs by stimulating them using controlled inputs and observing their outputs. If the input and output sets are "well-behaved" (e.g., numeric values) then it may be possible to prune the test space using the techniques of equivalence partitioning or boundary analysis.[4]

The HP SoftBench product presents a special challenge. Its user interface is almost completely mouse-driven and makes heavy use of hierarchically arranged windows. The system integrates actions of several tools through a message interface. The output is mostly visual. Finally, the system can run in a distributed environment on many processing units and varying displays.

Automatic testing of such a system implies the need for a "robot tester," blindfolded and handcuffed but capable of entering input and verifying output. One possibility is to operate at the pixel level and generate required actions (pushing buttons, etc.) at specific points on a screen. The verification of output would then require taking screen snapshots and comparing them with the expected screens. The problem is that this approach is tied directly to the screen's appearance. A mere change in fonts or other screen attribute (e.g., color scheme) would completely invalidate this testing approach. A higher-level approach is needed.

An HP SoftBench tool has two major interfaces: the user interface (mouse/keyboard/window) and the message interface. We used both interfaces to exercise and verify HP SoftBench behavior. To deal with the user interface challenge, we used two mechanisms that allowed us to stimulate inputs and register outputs independently of screen parameters. On the input side we identified inputs (buttons to be pushed or editing windows) not by their screen coordinates but by symbolic names associated with these objects. Thus, the automatic testing tools are able to find some `window9` no matter where that window is placed on the screen (or even if it is completely obstructed by other windows). To obtain higher-level verification of outputs without resorting to pixel-level screen dumps, we instrumented the code so that any window could be probed and forced to dump its contents (strings or a pointer position in the case of the edit widget, or a label in the case of a button). This approach allows selective probing of software objects (very much like having testing probes in hardware).

Testing Tools

Two companion testing tools were used to drive both HP SoftBench interfaces—user and message. Tool A allows the tester to send messages to the message server. It can intercept messages and match them against a list of expected messages. Tool A can also act on widgets. Tool A's companion utility, tool B, is capable of automatically and interactively creating a test file that mixes message and widget operations. This file becomes the input to tool A.

Tool A allows the user to send messages to the message server. Tool A will then wait for the tools to respond to the messages. The order in which the messages are sent and received is restricted by a partial order relation given by the user. This ordering can be totally unrestricted, strictly sequential, or anything in between.

Tool A maintains an active list of commands as it runs a test. As each command is executed it is removed from the active list and all of its successors are checked to see if they should be added to the active list. A command is not added to the active list until all of its predecessors have been executed.

Actions and Software Probes

Tool B can be used to log two types of events. It can intercept messages and it can also log operations on widgets into a test file. All of the widget-based commands search X11's window tree for the named widget at the time the command is executed. The search is done at this time because windows are constantly being created, destroyed, and moved about within the tree. The search algorithm does a breadth-first search of the window tree for the first name of the widget. As each match is encountered a second breadth-first search is started on the subtree of the matched window, looking for the second name of the widget. These searches continue until the tree is exhausted or all of the names of the widget are found. The search algorithm remembers that it has touched a particular application, and as a result all the subsequent widget searches use that shortcut (resulting in about a 90% speed-up in the search time).

When capturing tests with tool B, the tester can identify a widget that needs to be probed at the test time. The information dumped for a widget includes all text seen on the display, whether the text is sensitive (grayed out), and whether it is set or marked (only for menu buttons and toggles). This scheme allows us not only to describe the events to execute the actions, but also to specify what needs to be checked to verify that the actions happened correctly.

(continued on next page)

Supporting Utilities

Our tests were stored in the HP-UX revision control system (rcs) and ran in the proprietary HP Scaffold testing harness.[5] We used branch flow analysis (BFA) to monitor the coverage of the code and to steer the testing activities.[6] We combined the BFA information (annotated source code) and the results of the complexity analysis (McCabe's ACT[7]) to focus on testing the areas of the code that have high complexity and low BFA coverage (grey-box testing).

References

1. B.W. Boehm and P.N. Papaccio, "Understanding and Controlling Software Costs," *IEEE Transactions on Software Engineering*, 1988.
2. R.A. Sulack, R.J. Lindner, and D.N. Dietz, "A new development rhythm for AS/400 software," *IBM Systems Journal*, no. 3, 1989.
3. B.W. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, no. 5, 1988.
4. G.J. Myers, *The Art of Software Testing*, John Wiley & Sons, 1979.
5. C.D. Fuget and B.J. Scott, "Tools for Automating Software Test Package Execution," *Hewlett-Packard Journal*, Vol. 37, no. 3, March 1986, pp. 24-28.
6. D. Herington, et al, "Software Verification Using Branch Analysis," *Hewlett-Packard Journal*, Vol. 38, no. 6, June 1987, pp. 13-22.
7. T. McCabe, "A Complexity Measure," *IEEE Transactions on Software Engineering*, Vol. SE-2, no. 4, December 1976.

Jack Walicki
Software Development Engineer
Software Engineering Systems Division

- Support integrated tool sets. The tools should cooperate to present a task-oriented environment that lets users concentrate on what they want to do, not how to do it.
- Support interchangeable tools. HP's CASE strategy is based on the belief that there is no single solution appropriate for all users. The type of application being developed, the size of the team, the delivery constraints, and the development methodology all impact the optimal tool set. The integration architecture should permit any tool to be replaced such that no changes need to be made to the other tools and the new tools cooperate with the other tools in the environment at least as well as the original tools do.
- Support a distributed computing environment. The architecture needs to support software development in a distributed computing environment composed of combinations of X terminals, workstations, midrange computers, and servers, possibly in geographically dispersed locations. Tool execution, data, and display should all be designed for a network environment.
- Leverage existing tools. Users need to be able to integrate tools they already use, which they have either purchased or developed, into their software development environment. To do so, they should not have to modify the source code of any tool or change the other tools in the environment.
- Support software development teams. The tools and architecture should support team coordination and the management of project files in a distributed development environment.
- Support multiple work styles. The HP SoftBench product should not dictate a single style of work. The style should be based on the task. For example, if the user is primarily doing maintenance the environment should be centered around the maintenance task, and if the user is primarily doing rapid prototyping, the environment should be centered around the program construction task.
- Support other life cycle tools. The HP SoftBench architecture should facilitate the integration of additional life cycle tools such as project management, documentation, analysis, and design tools.
- Build on standards. The HP SoftBench architecture should build on the UNIX* operating system, NFS and ARPA networking, the X Window System™ Version 11, and the OSF/Motif appearance and behavior.

UNIX is a registered trademark of AT&T in the U.S.A. and other countries.

X Window System is a trademark of the Massachusetts Institute of Technology

Architecture Overview

We define a software engineering environment to be an *ensemble of tools that collaborate to support the user's software engineering process.*[1] There are several types of tool integration. The HP SoftBench tool integration architecture concentrates on providing mechanisms that support tool collaboration in a distributed computing environment. This type of integration is often called control integration or process integration.

The architectural facilities provided by the HP SoftBench product are complementary to those in other integration architectures that concentrate on providing services for sharing data between tools and managing data relationships.[2,3,4]

Over the last several years, university and industrial research laboratories have been addressing the issues of improved software tool integration facilities.[2,5,6,7] The HP SoftBench tool integration services are an implementation of much of this research in a commercial product. There are three primary components in the HP SoftBench tool integration architecture:

- Tool communication
- Distributed support
- User interface management.

Tool Communication

HP SoftBench tools communicate in a networked environment via a broadcast communication facility designed to support close communication of independent tools.

In the UNIX operating system, tool communication is typically limited to single-direction, point-to-point data streams (pipes). In the HP SoftBench environment, tool communication is two-way, one-to-many or many-to-one, and event-driven.

Message-Based Application Program Interface

All HP SoftBench tools, as well as nonSoftBench tools that have been properly integrated using the HP Encap-

Broadcast Message Server Message Structure

HP SoftBench tools communicate by sending messages, which are dispatched by the broadcast message server (BMS) to appropriate other tools. HP SoftBench messages have the following structure:

Originator Request-id Message-Type Command-Class Command-Name Context [Arguments]

Originator. The originator is the tool that sent the message. However, by convention, this field is not used by the HP SoftBench tools themselves because they do not send messages to a particular tool; they send them to the BMS so that other tools interested in the events can be notified.

Request-id. The request ID is constructed from the triple (message-number, process-id, host). This network-wide unique ID is used so that responses can be associated with their original requests. In other words, a notification sent as the result of a request has the same request ID as the original request to which it is responding.

Message-Type. The defined message types are:

R = Request message
N = Success notification
F = Failure notification.

Command-Class. The command class is the type of operation (e.g., EDIT, DEBUG).

Command-Name. The command name is the name of the operation within the command class (e.g., SAVE-FILE, STEP, STOP). The combination of the command class with the command name defines a unique operation, (e.g., EDIT SAVE-FILE or DEBUG STEP).

Context. The context is the triple (host, base directory, file). This indicates the location of the data being operated on. The context is used to distinguish between multiple instances of the same tool. For example, if the user is working on two projects at once and has two debuggers running, the context ensures that the right messages get sent to the right debugger.

Arguments. Each message may have optional, variable-format argument lists, which provide additional information regarding the operation—for example, the name of a function or variable. In the HP SoftBench product, complex data is passed by reference rather than by value. For example, if the message is a notification from the static analysis tool with the response to a request for a complex query, the arguments contain a pointer to a file containing the data.

sulator, provide access to their functionality through a message-based application program interface (API). Any action that can be initiated through the tool's user interface can also be initiated through the message interface.

When an HP SoftBench tool or an encapsulated tool wants to cause another tool to perform an operation, it sends a request message. The tool requesting the service does not know the particulars of the tool that will service the request. It only deals in terms of an abstract tool protocol. There are several predefined tool protocols in the HP SoftBench environment, one for each class of tool (e.g., DEBUG, EDIT, BUILD). Each tool protocol is composed of a set of operations (e.g., STEP, SET-BREAKPOINT, CONTINUE). As long as a new tool fully supports the appropriate tool protocol, that new tool can be substituted for the original tool, and the other tools in the environment continue to operate with the new tool just as they did with the original tool. With the HP Encapsulator, users can define new tool protocols or develop new tools for existing protocols.

There are several important benefits of having a message-based interface to all tools in the environment, but the primary reason is for task automation. Tools can be controlled by other tools instead of a person.

Other benefits of a message-based interface include programmatic application testing (see "Architectural Support for Automated Testing," page 37), computer-based training, and on-line help (see "Integrated Help," page 57). The value of a message-based API has been demonstrated in several systems. Most influential in the HP SoftBench design were the FIELD system done at Brown University,[5] the FSD system done at USC-ISI,[2] and the HP NewWave environment.[3,8]

Event Triggers

An important extension to the message-based API model is that all HP SoftBench tools and all external tools that have been integrated using the HP Encapsulator announce the action they just took after each operation they perform. This notification message is sent whether the operation was initiated from the user interface or from the message interface. The notification message also indicates whether the operation was successful (see "Broadcast Message Server Message Structure," above).

These notifications are the key to a powerful HP SoftBench concept known as *event triggers*. A trigger is a set of operations to be executed when an event occurs somewhere in the user's software engineering environment. As a simple example, when an HP SoftBench tool modifies a file, it announces this fact, so that other tools that are operating on the file can be updated appropriately. The notion of a tool announcing its operations comes from the research on FIELD at Brown University.[5]

In the HP SoftBench environment, certain triggers are predefined in the tools. More important, users can define their own triggers with the HP Encapsulator, keying off any notification in the user's environment. For example, the user might define an event trigger that automatically notifies the team whenever a successful build occurs, or metrics might be collected whenever a file is checked into version control.

Broadcast Communication

The HP SoftBench environment uses a broadcast model of tool communication provided by the facility known as the *broadcast message server* (BMS). The BMS is the dispatcher of messages between the various tools in the user's

Distributed Execution, Data, and Display

The distributed computing support capabilities of the HP SoftBench environment can support a variety of machine configurations. Fig. 1 shows an example of a configuration. Assume that an engineer has a small, inexpensive X display machine (possibly a diskless HP 9000 Model 340 workstation or an X terminal). Also assume that the HP SoftBench environment is installed on an HP 9000 Model 370 server machine, and that the engineer is developing software for an HP 9000 Series 800 machine, which is used as a central data storage facility. A typical HP SoftBench configuration will probably have one or two machines instead of the three in this example. However, it is possible to come up with configurations that use more machines.

In this example, first the engineer would start the HP SoftBench environment on the Model 370 HP SoftBench server, with the DISPLAY environment variable pointing to the X display server. One of the features of the HP SoftBench distributed execution facility is that the current environment of a process is always maintained when executing a child process, even over a network connection. Thus, if any X clients are spawned, the child X client will point to the correct display machine. Next, the engineer starts the HP SoftBench build tool, with the context set to the Series 800 machine. Finally, the engineer selects the build tool's build button. The HP SoftBench subprocess control (SPC) facility will now do two things. It will set the working directory to match the context host and directory, and it will spawn a make process on the Series 800 machine in this working directory. Alternatively, the build tool can be configured to spawn the make process on a different, possibly less loaded machine, but the working directory will still point to the same context directory.

What if a bug is discovered in this program? The engineer can start the program debugger on the Model 370 HP SoftBench server. This can be accomplished using the Actions:Debug menu pick of the development manager or the Tool:Start... menu pick of the tool manager, or by starting the HP SoftBench program debugger manually. The debugger again uses the distributed execution feature of the SPC to start an xdb process on the Series 800 machine, debugging the object files created by the build tool. Finally, the engineer can start the static analyzer to browse through the static analysis information generated by the build tool. The static analyzer is running on the Model 370 HP SoftBench server and accessing data on the Series 800 data server.

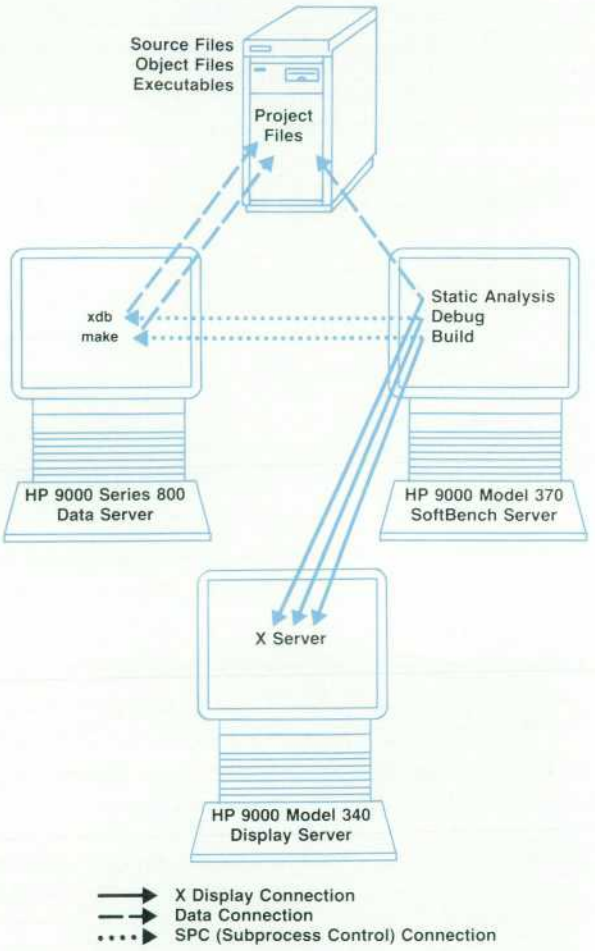


Fig. 1. *An example of an HP SoftBench distributed development environment.*

Gerald P. Duggan
Software Development Engineer
Software Engineering Systems Division

software engineering environment. Like a communications satellite, the BMS receives messages from tools in the environment and rebroadcasts these messages to all tools that have expressed an interest in each type of signal.

When an HP SoftBench tool or a tool integrated using the HP Encapsulator starts up, it establishes a connection to the BMS and announces its command class (that is, the tool protocol it supports) and the operations it will service (its message-based API). It also tells the BMS what events it would like to be notified about if and when they happen elsewhere in the environment (that is, the messages for which it wants to define event triggers). See "Broadcast Message Server Message Structure," page 39.

There are two types of messages in the HP SoftBench environment: *notifications* and *requests*. A notification is an announcement of an action, and a request is a tool asking the environment to perform an action.

When a notification message is received by the BMS, it forwards the message to the tools that have informed the BMS that they would like to receive those messages. For example, in Fig. 2, the program builder tool has sent a notification that a DIRECTORY_BUILD was successfully completed. The BMS forwards the message to tools that have

Fig. 2. *HP SoftBench tools communicate through the broadcast message server. Tools receive only messages they want to receive.*

Schemes: Interface Consistency

In the HP SoftBench environment, we wanted to distinguish regions according to the following functions:

- Unchanging system information: e.g., prompt strings
- Changing system information: e.g., the function being executed in the debugger
- User area: anywhere the user can enter text
- Read-only user area: a view of a read-only file
- Selectable regions: buttons.

The X11 window toolkit allows a choice of fonts, colors, and shadows for each region of an application main window. With the OSF/Motif appearance, regions can also be distinguished by 3D appearance. Areas can appear to be raised, lowered, or flat on the window panel. These collections of visual attributes organized by function are called *schemes*.

Information presented by the system, such as the current line number in the HP SoftBench program debugger, appears flat in the window. The label for system information uses a bolder font than the value. In the main window for a tool, there is a single background color for all system areas.

Areas where the user can enter information appear recessed in the window and have a different background color from system areas. Sometimes the user is prevented from entering information in a user area—for example, when a file being viewed has read-only protection. In these read-only user areas the background color is the same as the system areas.

Regions where the user can select using the mouse (buttons, for instance) use a large bold font and appear raised.

Windows that pop up (both menus and dialog boxes) use colors that associate them with the pull-down menu bars.

Much of the human interface is implemented in HP SoftBench library routines shared by all applications. These high-level calls create widgets of known names. As a result, human interface consistency is ensured and the number of resources needed to specify a scheme is minimized. Widget classes are used where possible to distinguish scheme components, but where a single widget class is used for more than one purpose, widget names or widget class hierarchies must be used.

Choosing a Scheme

Schemes for monochrome and color are provided in three different font sizes. If the user does not specify a particular scheme by setting the **Scheme** resource, a scheme will be chosen based on screen resolution, visual class, and screen depth. Font size is chosen based on the screen resolution. A color scheme will be used only if the screen has at least four color planes and the desired colors are available.

Implementation

Scheme files are ordinary X11 resource files. X11 resources are used to configure tools. Each application constructs a resource data base for itself. Some resources apply to all instantiations of a tool—for example, the arrangement of windows in an application. Others, the scheme resources, may depend on the particular display being used. Users can override resource values.

HP SoftBench tools have resource files not used by other X11 applications. This was done to permit the sharing of resources between applications. Resources for code implemented in shared libraries are stored in the **LibXe** resource file. Scheme resources are placed in their own file. This approach allows easy configuration with all font and color specifications isolated and shared.

John R. Diamant
Colin Gerety
Software Development Engineers
Software Engineering Systems Division

expressed an interest in this event. In this case, the development manager will use the message to trigger a directory update and the static analyzer will use it to trigger a reanalysis of its program data base.

Often, no tools have expressed interest in a given message. When this is the case, the message is discarded. Tools get only messages they have requested. This serves to simplify a tool's message processing and substantially reduce the message traffic on the network.*

Tool Execution

When a request message is received by the BMS, the HP SoftBench environment first checks to determine whether an already running tool has indicated through its API that it will service this type of request. If there is such a tool running, the HP SoftBench environment forwards the request message to that tool.** If there is no such tool running, the HP SoftBench environment checks a user-customizable data base that contains the name of the appropriate tool to start and instructions for starting it. The HP SoftBench environment then starts the tool and waits for confirmation from the tool that it will indeed handle the request. Once this confirmation is received, the HP SoftBench environment forwards the queued request message to the tool just started. The HP SoftBench module that monitors the tools that are currently executing and invokes tools when necessary is referred to as the *execution manager*.

In the HP SoftBench environment, as in the HP NewWave environment, but unlike the UNIX operating system, users do not have to start tools explicitly. They request actions on objects, and if a tool needs to be started, the execution manager starts it for them correctly and automatically.

Distributed Support

It is a fundamental goal of the HP SoftBench environment to support development in a distributed computing environment. This is defined to include configurations of several hundred computers, composed of arbitrary combinations of X terminals, workstations, servers, and larger computers. The goal of HP SoftBench distributed computing

* Strictly speaking, "broadcast" message server is somewhat of a misnomer, since only tools that have explicitly requested certain messages are forwarded those messages. The term "multicast" would be more accurate.

**This explanation has been somewhat simplified. An important factor in message dispatching is the *context* of the message. All messages contain a triple that indicates the location of the data the message is referring to. The HP SoftBench environment uses this to distinguish between multiple instances of tools.

Pervasive Editing in the HP SoftBench Environment

All HP SoftBench tools have common editing needs. All must prompt the user for input, and many provide views into source files. For consistency of the human interface, it is an HP SoftBench requirement that a common set of editing commands be used in all editable areas. The editing functionality exists in the HP SoftBench library and is shared by all the tools. Because the code is shared by all applications, consistency of appearance and behavior is ensured.

The Edit Widget

All of the underlying functionality needed by the editor was put into the *edit widget*. This includes insertion and deletion of text, cut, copy and paste, language dependent selection such as tokens or statements, and undo history. The edit widget supports 16-bit characters and has language-sensitive editing capabilities. The availability of the widget makes it inexpensive to have this exact functionality in numerous places throughout an application. In fact, this single copy of code is shared among all the HP SoftBench applications. An edit widget is used in all areas where a user can type information.

A variant of the edit widget is used to provide selection from a list of alternatives, such as a list of filenames in the development manager directory list, function names resulting from a query in the static analyzer, or a list of mail messages in the HP SoftBench mail subsystem.

One-Line Editables

The human interface needed to include many areas for displaying small user input windows that could be labeled with concise prompts, such as for a filename, an execution hostname, or a search string. These are implemented with a packaged combination of a static text widget for the prompt and an edit widget for the user data. Called a one-line editable, this type of entity provides the application writer with a single widget to specify geometry placement. The individual constituent widgets allow specification of different fonts and background colors to inform the user which areas are constant and which are modifiable. Using such an editable gives the application all the power provided by the editor in each input field, with no additional code. One-line editables are used in dialog boxes as well as in the application's main window.

View Space

Applications often need one or more windows to display a view into a possibly large piece of text, such as the contents of a file or a set of debugger output messages. Each such screen area is associated with a view space. A view space provides a scrollable area for an edit widget, a place for a title, a filename, and a collection of indicators or buttons.

The size of the view space is driven by the size of the edit widget, which can be specified in character rows and columns instead of pixels. The modifiability of the region is indicated by the background color. The application can choose to hide the indicators selectively. Buttons not prohibited by the application automatically appear as needed.

If several files or buffers share the same view space, an index button appears. This allows the user to select from a list which of the views in that space should be displayed.

Viewing Files

To insert a file into a view space, the application makes a call such as:

```
DisplayFile(ViewSpace, DataHost, DataDirectory, DataFile, LineNumber);
```

Access to a host other than the execution host is provided transparently to the application.

A similar interface allows the application writer to add annotations to a given line of a file. These annotations are displayed as small pictures in a window adjacent to the edit widget. These pictures ride with their associated line as the file is edited or scrolled. Annotations are used, for example, to denote debugger breakpoints or the program counter position.

File Synchronization

Since several applications may be viewing the same file, the file-viewing library routines provide for keeping these views synchronized with the file system, using the broadcast message server (BMS). When the edit widget successfully saves a file, a FILE-MODIFIED message is sent, alerting other applications interested in the file. By default, the other applications automatically load the new file from disk. If there is risk of destroying a user's modifications, or if the user has so requested, a prompt box appears asking whether or not to reload the file.

William A. Kwinn
Software Development Engineer
Software Engineering Systems Division

support is to facilitate the use of the network and to hide from the user any complexities the network introduces.

Remote Execution

In the HP SoftBench environment, any tool can execute on any host in the network.* To support this remote execution, HP SoftBench includes a distributed execution mechanism designed to make the remote execution transparent to the HP SoftBench tools.

To communicate with processes running on a local or remote computer, a high-level protocol is used. When communicating with a remote computer, a small daemon program known as the HP SoftBench *subprocess control* (SPC) daemon is used. This remote daemon is automatically invoked through the HP-UX inetd facility. Process control between the initiating tool and the remote process is then conducted through the SPC daemon.

HP SoftBench tools communicate in a distributed environment via the BMS, as described earlier. This communication mechanism is network-based so that tool communication works identically whether all tools are local or each is on a different computer in the network.

One of the most powerful applications of distributed execution is for special-purpose execution servers. For example, a project of ten software developers might wish to designate a dedicated (or otherwise lightly loaded) computer on the network to be used for all program builds (compiling and linking). With the HP SoftBench distributed

(continued on page 44)

* On each computer where any HP SoftBench tool or encapsulated tool will execute, the HP SoftBench product must be installed.

Native Language Support

Localizability has been one of the goals for the HP SoftBench product from the beginning. To this end, all of the code has been written to handle both 8-bit and 16-bit data (see Fig. 1). The only limitations are imposed by some of the underlying HP-UX tools, such as rcs and mkmf. The HP SoftBench edit widget can receive 16-bit input from the native language I/O facilities (see below). The fonts designed for the HP SoftBench environment include all of the HP Roman8 characters.

The HP SoftBench environment uses the configurability of X11 applications to full advantage. Instead of putting localizable strings in message catalogs, we have opted to put them in X11 resources. Localizers can redefine the values of X11 resources to translate HP SoftBench pull-down menus, button labels, prompts, and error messages. Because we have made maximal use of widgets that autoscale in size, there should be few places where localizers need to adjust screen layout parameters. By setting a single resource value, localizers can change editing commands from chorded combinations with the **Extend char** key to sequence combinations with the **ESC** key. They can also use X11 resources to specify appropriate font schemes, and even to refer to alternate icon bit maps.

In response to a request from the HP SoftBench developers, the X11 team added support for an environment variable that causes the system to search for application defaults in a specified directory, rather than in /usr/lib/X11/app-defaults. This allows the unlocalized version and one or more localized versions of the HP SoftBench environment to reside on the same system.

By setting a few environment variables and adding at most several lines to .Xdefaults, the user can use a localized HP SoftBench environment with virtually no speed penalty.

Edit Widget

The HP SoftBench edit widget is the core of all 8-bit and 16-bit data handling in HP SoftBench applications. The user's textual data, either in the form of files loaded from the file system or text entered into editable fields, is stored and presented to the user through the edit widget. This use of the edit widget makes it much easier for each tool to provide data integrity.

Native language I/O support is part of the consistent editing facilities described in "Pervasive Editing in the HP SoftBench Environment" on page 42. Wherever the user can enter text within HP SoftBench tools, native language I/O is available. Because it is provided by the edit widget, it is transparent to individual tool implementors.

The edit widget is built on the R2 version of the XtIntrinsics. At application start-up, it determines from the HP-UX environment variable, LANG, whether it needs to handle 8-bit or 16-bit data and makes appropriately configured buffer structures. It also checks whether an Asian language keyboard is attached to the X server. If the keyboard is Asian or if an environment variable, KBD_LANG, is set to an Asian language, the widget will activate a native language I/O server process. In the final product version built on the HP-UX 7.0 release, the HP X extension library provides much of the support necessary to handle all of HP's supported keyboards correctly.

The design of the edit widget is based on object-oriented principles. Since a supported object-oriented language was not available on the HP-UX operating system at the start of development, we used a set of C macros to code in a simple but effective object-oriented style. The 8-bit and 16-bit capabilities are provided by specialized subclasses of base objects, which provide

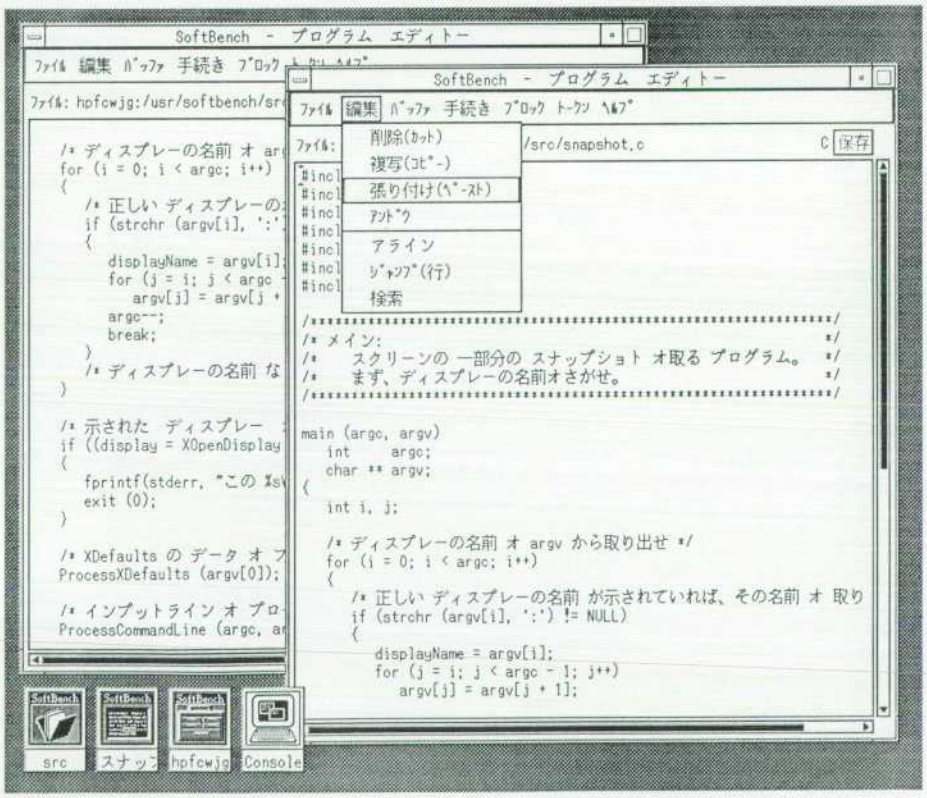


Fig. 1. *The HP SoftBench environment with Japanese localizations.*

most of the drawing and data storage functionality. At creation time the edit widget decides whether to create 8-bit or 16-bit pseudo-objects. This allowed significant code sharing. The language-sensitive actions of the HP SoftBench environment were added in a similar manner. In this case one of the primary benefits was to allow two different engineers to work on tightly coupled code with minimal interference.

Warren J. Greving
Kathryn Y. Kwinn
Software Development Engineers
Software Engineering Systems Division

(continued from page 42)

execution capability, the team's environment can be customized to execute every user's builds on the dedicated compile server. This allows the team members to maintain full performance on their personal workstations while their compiles are performed on the server, which is not burdened with other tasks that might slow down the compilations. The same notion can be applied to any of the tools.

Remote Data

In the HP SoftBench environment, data can reside on any host in the network. Regardless of where a tool is running, it can access the data. The user provides an HP SoftBench file specification, which may contain an optional host field (e.g., machine1:/usr/src/project1/file1.c). If a remote host is specified, the distributed data facilities are automatically employed to establish the path to the remote file.

With large teams, it is often easier to manage and administer data centrally than to have the data duplicated on each workstation in the network. For example, configuration management, tape backup and archiving, and project management are typically easier when the project files are centralized. This was and remains one of the benefits of timesharing systems.

The most common application of the remote data feature is the use of a data server. For projects that prefer or require dedicated computing power for each engineer, yet wish to have a common location for project data, the HP SoftBench distributed data capability facilitates this. Project members can run their tools locally but designate and use a common computer and file system location for project files (e.g., fileserver:/usr/src/project1).

Remote Display

HP SoftBench tools are built on the X Window System Version 11, which is a network-transparent window system. One of the benefits of the X Window System is that X programs can run on one system and display visually on another. In fact, to run the HP SoftBench environment, the only process that must actually be observable to the user is the X display server, which requires a bit-mapped display and adequate RAM.

The extreme example of this is the X terminal products that are now appearing. These act as smart terminals that run the X display server and connect to the network. The HP SoftBench tools the user runs actually execute on other computers in the network. Moreover, the computers that actually run the HP SoftBench tools do not need to be running X11. They must simply support the X11 client library interface and be connected to the network.

The HP SoftBench remote execution, data, and display capabilities described above enable diverse configurations of workstations, servers, and larger computers to be employed, based on the needs of the user's team, to increase software development throughput and decrease per-seat cost (see "Distributed Data, Execution, and Display," page 40).

User Interface Management

To provide a consistent appearance and behavior among the many HP SoftBench tools, and to facilitate the use of the OSF/Motif user interface style, the HP SoftBench integration services contain user interface management software. This software provides support for *schemes* (see "Schemes: Interface Consistency," page 41), pervasive code editing and viewing (see "Pervasive Editing in the HP SoftBench Environment," page 42), native languages (see "Native Language Support," page 43), and interactive help (see "Integrated Help," page 57).

User Model

The HP SoftBench tools have been designed so that they can support many different styles of work. A programmer doing rapid prototyping may use the same set of HP SoftBench tools as one doing maintenance, but they may be used quite differently, since the task is different.

The programmer doing rapid prototyping may keep a "home base" in the program editor, while one doing maintenance may have a home base in the static analyzer. However, both have easy, integrated access to the other HP SoftBench functions such as file version management and program builds.

In the HP SoftBench system model, each tool provides the actions that are appropriate based on the type of data managed by the tool. For example, in the program editor, source files are viewed and manipulated. However, the programmer can also check in and out the currently edited file, cause the file to be compiled, and ask static analysis queries, such as where a given function is defined. The static analyzer provides cross-reference and code browsing information, yet the programmer can edit the files being viewed, check them in and out of version control, and cause the files to be rebuilt. This remote access to other tools' functionality is provided by the HP SoftBench tool communication architecture. It lets the programmer concentrate on the task at hand, while the tools cooperate among themselves to perform requested operations.

Human Interface

The HP SoftBench environment provides an object-action user interface model. The user first selects the object that will be operated on, and then selects one or more actions to be performed on that object. The environment works to provide a task-oriented, rather than a tool-oriented view of the environment to the user. The HP SoftBench user interface style was significantly influenced by the HP NewWave user interface work.[9]

Mechanisms for Efficient Delivery

The HP SoftBench environment is composed of several communicating processes, all running under the HP-UX operating system and the X Window System Version 11. Each HP SoftBench tool is built on the X Version 11 C library interface, the X toolkit, the HP widgets, and the HP SoftBench common code library. Most X toolkit applications are very large because of the sizes of the required libraries. Each of the dozen HP SoftBench processes would be well over a megabyte in size if it were linked in the standard fashion, having its own private copy of all the library code.

To deliver the HP SoftBench environment effectively, we developed a delivery technology that significantly reduces the size of the tools and improves the performance. To the user, these facilities are completely transparent. The user runs HP SoftBench tools just like any HP-UX program or shell script. These facilities are not available to the end user. They are used only to ensure effective delivery of the HP SoftBench tools.

The large executable size is a problem, but not only because of the disk storage space required. With several of these programs all running at once (as they typically are in the HP SoftBench environment), the physical RAM in the computer can be exceeded by a large factor. The virtual memory system allows the system to continue to run, but performance degrades as more pages of memory are moved to the swap device.

The solution was to have just one copy of the library code, rather than many. The common library for all the HP SoftBench applications is about one megabyte in size. All of the SoftBench tools (except the HP Encapsulator) are less than 200K bytes in size when stripped of their private copies of the library.

Implementation

The idea of shared libraries is not new. Many UNIX implementations support them. However, no shared library facility existed on the HP-UX operating system at the time we needed it for our product, so we implemented our own. There were some technical choices to be made:

- Where should the shared code physically reside so that it can be accessed simultaneously by all of the tool processes?
- How can the individual tools be linked to the shared code so that the addresses of entry points and globals are properly resolved?
- How can the various tools be invoked such that the attachment to the shared library is transparent to the user?

Storing the Code

Our first approach was to use shared memory segments. These are regions of memory that can be created by one process and then accessed by many others. We loaded the library code into one or more of these segments. Any tool could then attach to these segments and execute code directly out of them. This was conceptually very simple. We could put each separate library (libc.a, libX11.a, libXt.a, etc.) into its own segment, and each application only had to attach to as many segments as it needed. However, there were problems with this approach. First of all, it required an explicit step in the initialization of the environment to create these segments and load them. It also required some user action to deallocate them when taking down the environment. Also, on HP PA-RISC computers, there was a performance degradation if an application needed more than two of these segments.

The solution was to put all the library code into a demand-loadable executable program. The HP-UX system automatically shares the code of such an executable if it is being executed simultaneously by multiple processes.

Linking the Code

At first we tried to link each of the tools statically to the library code. The HP-UX linker ld has a special option to do this. This approach would have been the most straightforward way of delivering the product to users, but we found that it was too inflexible. Once the tools had been statically linked to a particular library, any changes to the library required relinking all the tools. We needed rapid prototyping: as we changed or added features to the library, we wanted to test the changes quickly without having to rebuild everything.

The solution was to delay linking the tools until run time. A dynamic loader resolves external symbols and relocates the code when the application is loaded. This link step is very fast (1 to 2 seconds) because it all happens in memory. The tools themselves reside on disk as standard unlinked .o files.

Invoking the Tools

As a result of the decisions just described, all HP SoftBench tools are invoked by running a single, large, demand-loadable program that contains a dynamic loader and all of the common library code for the product. This program is called runprog until it becomes part of the HP SoftBench product. It is possible to run any of the HP SoftBench tools by executing runprog explicitly, but there is a quick trick that hides what's going on. We use the HP-UX ln command to give runprog several aliases. There is still only one runprog, but each of the HP SoftBench tools is actually just another name for it. Runprog figures out which tool to run by looking at its own invocation name, argv[0], then appends .o to that name and invokes the dynamic linker. Now HP SoftBench tools can be executed transparently as in any other HP-UX program or shell script.

Sam Sands
Software Development Engineer
Software Engineering Systems Division

The HP SoftBench environment follows the OSF/Motif appearance and behavior. This interface technology is largely mouse- and menu-driven, with human-computer interaction occurring primarily through dialog boxes (see Fig. 3).

Several benefits are provided by the OSF/Motif technology:

A rich set of primitives on which to build sophisticated user interfaces.

- Keyboard traversal for users who prefer to perform some or all operations from the keyboard rather than with a pointing device.
- Native-language input and output for accepting and displaying languages requiring 8-bit and 16-bit character sets.
- User-definable keyboard accelerators for common menu actions.
- Consistency with PC-based applications to facilitate in-

Application of a Reliability Model to the HP SoftBench Environment

The HP SoftBench team decided to incorporate a statistical reliability model into the data gathering process during the system test phase to help us better understand the current quality level of the code and predict how long it might take to attain a given quality level. The model is based on the work of Kohoutek,[1] with additional results from Musa, Okumoto, and Goel.[2,3] Similar models have been used in other HP Divisions.[4,5,6] We learned of it from Doug Howell.[7]

The basic idea is to fit a logarithmic Poisson execution time model to the plot of defects found versus time. At time t (in hours), the number of defects found, u(t), is given by:

$$u(t) = \theta(1 - e^{-\lambda t/\theta})$$

where θ is a scale parameter and λ is the defect finding rate.

Each week, when we had a new data point on the graph of defects found versus test hours, we used nonlinear least squares iteration to find the θ and λ that produced a best fit of the u(t) curve to our data. From the very beginning, the fit of the curve to our data points was remarkable.

The scale parameter θ is the limit of the function u(t) as t approaches infinity—that is, it is the number of defects the model predicts are in the product. We found the stability of θ over time to be an interesting subject; we will say more about this below.

The first derivative of u(t) is the rate at which defects are being found. The reciprocal of the finding rate is the instantaneous mean time between defects (MTBD) at time t:

$$MTBD = 1/u'(t) = (1/\lambda)e^{\lambda t/\theta}$$

This equation was extremely useful, since it allowed us each week to predict when a given MTBD would be achieved. We solved this equation for t for various values of MTBD, given the current values of θ and λ. We then converted t from hours into calendar time by dividing by the average number of test hours we were logging per week.

This gave us a weekly prediction of the calendar date when we would achieve an instantaneous MTBD equal to our goal. We noticed that θ (hence the prediction) was fairly unstable at first. Then we fitted u(t) to different data. Instead of computing θ based on the plot of raw unweighted defects versus test hours, we used what we call filtered weighted defects. Duplicate reports and enhancement requests were removed from the count, and defects were weighted according to the severity assigned to them by the project team (on a scale from 0 to 1).

This of course gave us an MTBD that meant something different than before. It was now the instantaneous mean time between virtual defects of weight 1 instead of the time between any defects found, regardless of severity. We decided that this new MTBD number actually meant more, given that the SoftBench product is user-interface-intensive and many people were submitting low-weight defects that were stylistic, personal preference issues.

When we switched to computing θ based on the filtered weighted defect plot, it became quite stable. Several months ahead, we predicted that we would reach our MTBD goal on a particular date. The actual MTBD on that date was 95.7% of the goal, and we reached the goal and did our final build five days later.

We feel that the use of this simple model was very successful in achieving the objectives of understanding where we were and providing a rational (as opposed to emotional or schedule imposed) prediction of when we would be finished.

References

1. H. Kohoutek, "A Practical Approach to Software Reliability Management," *Proceedings of the 29th EOQC Conference on Quality and Development*, 1985, pp. 211-220.
2. J.D. Musa and K. Okumoto, "A Logarithmic Poisson Execution Time Model for Software Reliability Measurement," *IEEE Transactions on Reliability*, Vol. R-33, 1984, pp. 230-381.
3. A.L. Goel and K. Okumoto, "Time-Dependent Error Detection Rate Model for Software Reliability and Other Performance Measures," *IEEE Transactions on Reliability*, Vol. R-28, 1979, pp. 206-211.
4. H.D. Drake and D.E. Wolting, "Reliability Theory Applied to Software Testing," *Hewlett-Packard Journal*, Vol. 38, no. 4, April 1987, pp. 35-39.
5. G.A. Kruger, "Project Management Using Software Reliability Growth Models," *Hewlett-Packard Journal*, Vol. 39, no. 3, June 1988, pp. 30-35.
6. G.A. Kruger, "Validation and Further Application of Software Reliability Growth Models," *Hewlett-Packard Journal*, Vol. 40, no. 2, April 1989, pp. 75-79.
7. D. Howell, "A Simple Method for Predicting the Duration of Software QA," Internal Memo.

Tim Tillson
Project Manager
Software Engineering Systems Division

teroperability across computing platforms.

Conclusion

We have described the various mechanisms provided by the HP SoftBench tool integration architecture for tool communication, distributed data, execution, and display, and user interface management. The communication facilities are exploited by the HP SoftBench software development tools to collaborate in presenting a task-oriented environment to the programmer. The distributed execution, data and display services are used by the tools to allow the user to make effective use of the computational, file storage, and presentation capabilities available on the network. This can improve performance, reduce per-seat workstation cost, and facilitate development for large software teams in a distributed environment. The user interface management facilities allow the tools to present a consistent, localizable, customizable environment that is easy to learn and use.

The HP Encapsulator provides these integration services to existing nonSoftBench tools, without requiring access to the tools' source code.

Acknowledgments

The HP SoftBench product began as a research project in the Software Technology Laboratory of HP Laboratories in Palo Alto, California, under the internal name Ivo. The decision was made to build a product based on this research at the Software Engineering Systems Division in Fort Collins, Colorado. The HP SoftBench product involved a great many people in R&D, marketing, and the field. Special thanks to HP SoftBench product marketing engineer Becky Hennig, our human factors engineer Greg Foltz, the HP SoftBench QA team led by Don Watt, Roy Williams, and

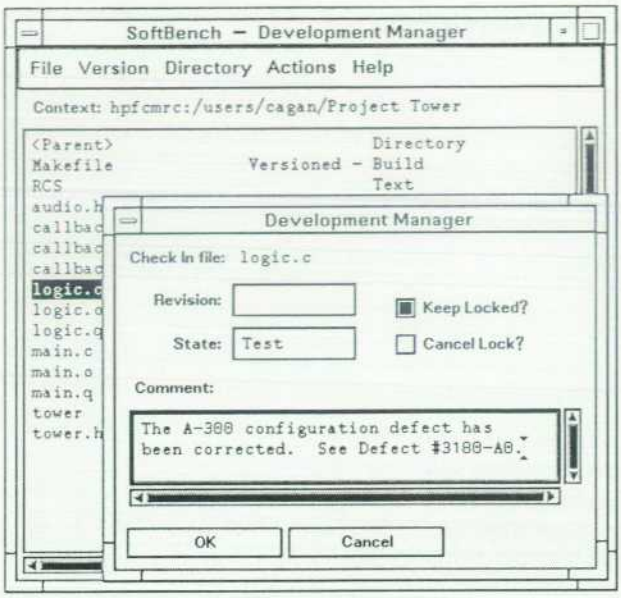


Fig. 3. *An example of a dialog box prompting the user for version information.*

Kirsten Duff, the documentation team of Mary Edelmaier, Dave Koons, and David Wolpert, and our partners at the Corvallis Workstation Operation, the California and Colorado Languages Laboratories, HP Laboratories, and the Software Engineering Systems Division in Palo Alto.

The HP SoftBench integration platform was designed and built by Michael Baumann, John Diamant, Jerry Duggan, Colin Gerety, Warren Greving, Bill Kwinn, Kathy Kwinn, Sam Sands, Gerrie Shults, Tim Tillson, Jack Walicki, and Judy Walker.

References

1. R. Ison, "An Experimental Ada Programming Support Environment in the HP CASEdge Integration Framework," *Proceedings of the International Workshop on Environments*, Chinon, France, September 1989.
2. B. Balzer, "Living in the Next Generation Operating System," *IEEE Software*, November 1987, pp. 77-85.
3. I. Fuller, "An Overview of the HP NewWave Environment," *Hewlett-Packard Journal*, Vol. 40, no. 4, August 1989, pp. 6-8.
4. G. Boudier, F. Gallo, and I. Thomas, "Overview of PCTE and PCTE+," *ACM SIGPLAN Notices*, Vol. 24, no. 2, February 1989.
5. S. Reiss, *Overview of the FIELD Environment*, Brown University, Department of Computer Science, November 1987.
6. M. Cagan and A. Ishizaki, "Ivo: An Integrated CASE Environment," *Proceedings of the Hewlett-Packard Software Engineering Productivity Conference*, 1986.
7. M. Cagan and D. Young, "The Ivo Tool Integration Platform," *Proceedings of the Hewlett-Packard European Software Engineering Productivity Conference*, 1987.
8. G. Stearns, "Agents and the HP NewWave Application Program Interface," *Hewlett-Packard Journal*, Vol. 40, no. 4, August 1989, pp. 32-37.
9. P. Showman, "An Object-Based User Interface for the HP NewWave Environment," *Hewlett-Packard Journal*, Vol. 40, no. 4, August 1989, pp. 9-17.

A New Generation of Software Development Tools

The HP SoftBench environment's development manager, program editor, program builder, static analyzer, program debugger, and mail collaborate to support task-oriented program construction, test, and maintenance.

by Colin Gerety

THE HP SOFTBENCH PRODUCT, as explained in the article on page 36, provides an integrated software development environment designed to facilitate rapid interactive program construction, test, and maintenance in a distributed computing environment. This article presents examples of computer-aided software engineering (CASE) tools that use the services of the HP SoftBench tool integration architecture.

The HP SoftBench environment is designed for software development teams that have the following characteristics:

- They need strong program construction, test, and maintenance support.
- They want to automate tasks in their development process.
- They want a task-oriented system that is easy to learn and use.
- They want to integrate their existing tools into their development environment and processes.

The HP SoftBench environment is designed so that users can focus on software development tasks rather than on the specific tools needed to accomplish the tasks. Instead of having to specify tools, arguments, and data for each step required to perform a task, HP SoftBench users select an object to operate on (for example, an executable file) and then specify what they want to do (for example, debug). The environment determines what tools to run, what machine to run them on, how to start them, what arguments are required, and where the data resides that they will operate on. On the other hand, HP SoftBench users who prefer the conventional tool-oriented mode of operation can work in that mode at their option.

The HP SoftBench tools collaborate to support five targeted software development tasks. Two of these—team file management and team communication—are pervasive. The other three—program construction, program testing, and program maintenance—support specific software life cycle phases. A set of HP SoftBench tools assists the user with each of these five tasks. The current HP SoftBench release supports software development in C, Fortran, and Pascal.

Team File Management

Teams of software developers working together need ways to manage access to and revisions of the files that compose the software project. Team members need a stable and controlled area to develop and test the project software. They also need an easy way to retrieve changes and additions to the project files made by other team members, so that they can test their own changes with the latest version. Once they are satisfied that their modifications work with the latest version, they need to submit or promote their modifications or additions into the master set of project files. The HP SoftBench development manager provides these services (see box, page 49).

The notions of reserving, locking, or checking out a file and then replacing or checking in the modified file are central to all development, test, and maintenance activities. Therefore, all HP SoftBench tools that allow modification of project source files communicate directly with the development manager to retrieve the current file from the version management system or to return it.

For example, if you are a developer who wants to modify a file, you request access to the file through the development manager. If another developer is already in the process of modifying that file, the development manager denies the request and tells you which team member is already working on the file. In this case, you can either ask the development manager to create a branch, which will need to be merged with other changes later, or you can contact the named team member, either directly or through HP SoftBench mail, to negotiate access to the file.

In addition to this team support, standard versioning operations are provided, such as storing change information for files so that previous revisions can be retrieved. Also included is the ability to tag specific file versions with symbolic names, such as Release 1.0, or with specific states, such as Experimental. Sets of files that make up particular configurations of the project can be stored and retrieved on demand. You may want to retrieve, for example, all the files that make up Release 2.0 or all the files for a certain HP-UX version that have been tested. You might also ask to retrieve the project files as they were on January 9.

Team Communication

HP SoftBench team communication support is designed to facilitate communication among members of the project team so that shared resources are efficiently used, developers are notified of key system events, and work and meetings can be arranged and coordinated. Like team file management, team communication is a pervasive task, neces-

Development Manager

The HP SoftBench development manager manages the versions of files on which the other tools operate. Fig. 1 shows the development manager user interface. The user can check files in and out, examine change histories, and compare revisions.

In the development model supported by the development manager, each person has a private, local work area, which is associated with a team or master work area. The development manager presents a view of the files in the user's work area, along with the state of each file (file type, whether it is under version control, whether it is locked or writable).

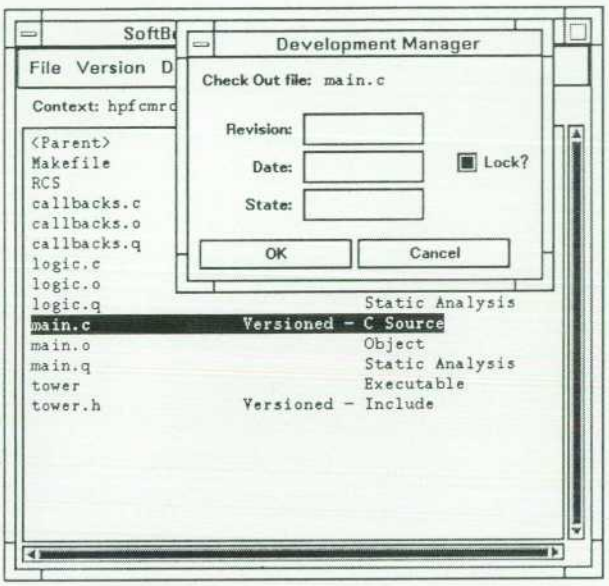


Fig. 1. *Development manager user interface.*

The version management functions provided by the development manager can be accessed directly through the development manager user interface or indirectly from the other tools via the development manager's message interface. For example, all HP SoftBench tools that provide the ability to edit source files (e.g., static analyzer, program builder, program editor) allow the user to check files in or out from version control. The tools do this by communicating with the development manager to perform the requested services. Thus the development manager's version control is pervasive throughout the environment.

The development manager also serves as an application launcher. Actions are presented based on the file type. For example, the user can run, rebuild, or debug an executable file and edit, compile, and show functions of a C source file. Some actions are serviced directly by the development manager, while others are forwarded via the broadcast message server to other HP SoftBench tools or user encapsulations.

Anthony P. Walker
Software Development Engineer
Software Engineering Systems Division

sary in all phases of software development.

The HP SoftBench product includes an electronic mail facility, HP SoftBench mail (see Fig. 1). Besides the standard electronic mail capabilities of viewing, replying to, and filing messages from others and composing and forwarding messages to others, HP SoftBench mail is designed to link into the rest of the software development environment. As an example of such a link, HP SoftBench mail can be instructed to watch for the completion of a system build and send out an announcement to members of the team if the build fails. If the build is successful, HP SoftBench mail can announce the success, letting the team know that a new release of the software is available (see Fig. 2).

More information on the HP SoftBench mail tool can be found in the article on page 59.

Program Construction

Program construction is the transformation of a design into an executable program. The HP SoftBench environment supports either writing new source code or assembling software from existing components, or a combination of the two.

Large projects often contain one or more common code libraries, designed to be reused throughout the application. Software reuse of this sort has been shown to be a major factor in improving both software quality and productivity. The HP SoftBench environment facilitates software reuse by addressing one of the common obstacles to reuse: locating appropriate functions and procedures. The HP SoftBench static analyzer (see box, page 54) assists in this task by making it easy to search the project libraries, looking at parameters, return values, and function definitions.

Programs are created, modified, or synthesized from existing pieces in a program editor. The HP SoftBench program editor (see box, page 51) assists with this task, and provides quick access to the other program construction tools. For example, the user can select a function name in a program and ask the HP SoftBench environment to show the definition of the function or other references to the function within the application. The user can also check the syntax of the program file under construction. The program editor is targeted at programmers who are new to the HP-UX operating system or who have worked in PC environments and want a mouse- and menu-based editor. If the user already has a favorite UNIX program editor, such as emacs or vi, the HP SoftBench environment allows the use of that editor instead.

Once the program source files have been created, the HP SoftBench system analyzes module dependencies and constructs a recipe, or *makefile*, for building executable programs or libraries. The HP SoftBench system then builds the program, compiling and linking only modules that are out of date. If compile or link errors are encountered, the user can correct them interactively. The HP SoftBench program builder (see box, page 52) assists with this task.

The result of the program editor, program builder, and static analyzer working together is a rapid edit-compile cycle allowing quick exploration of alternative implementations or construction of new functionality.

(text continued on page 54)

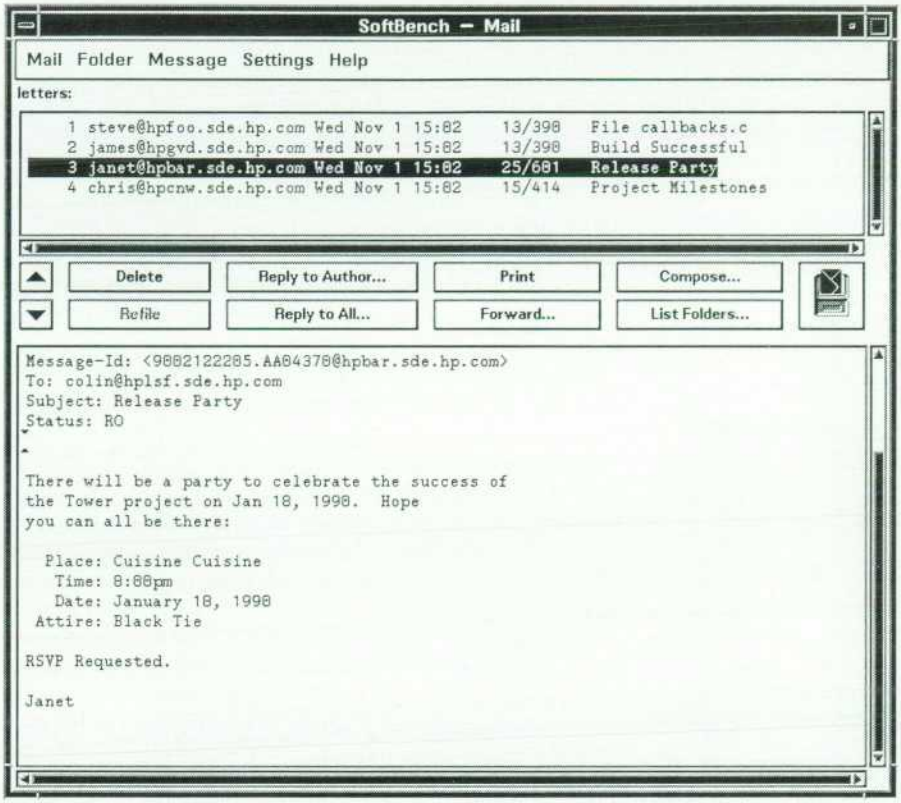


Fig. 1. *HP SoftBench mail user interface.*

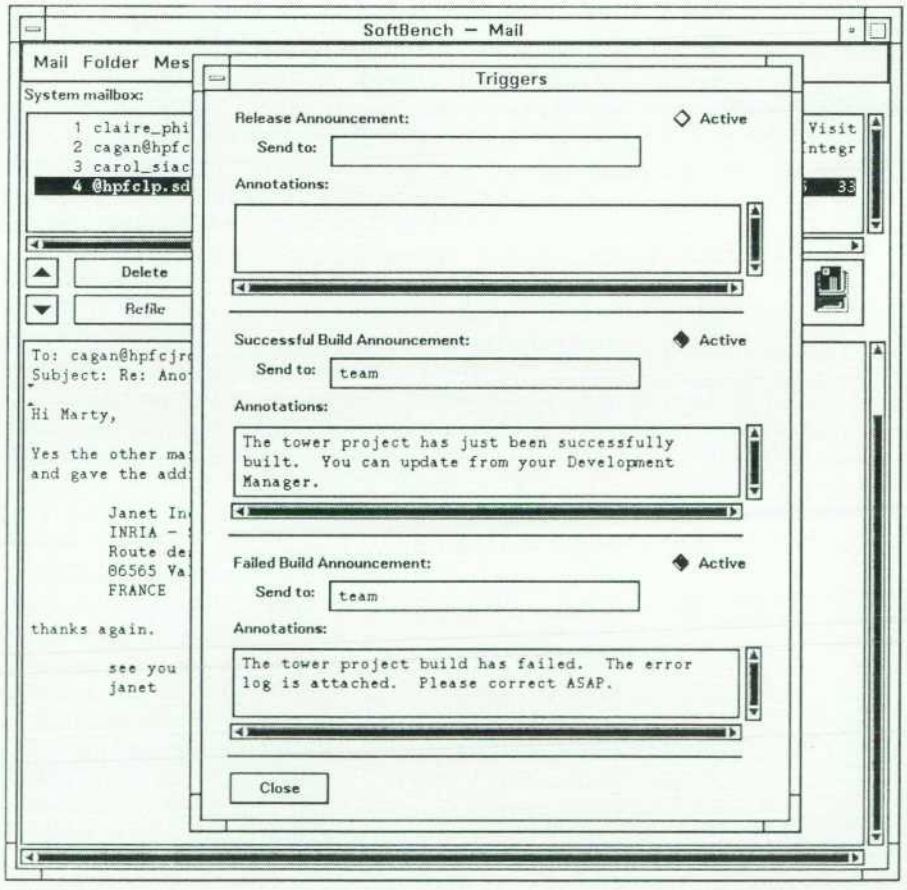


Fig. 2. *Setting up HP SoftBench mail triggers.*

Program Editor

Editors are controversial. Despite considerable reluctance, based on our conviction that there are already too many editors for HP-UX programs, we found ourselves writing a new editor for the HP SoftBench environment. The requirements that forced this decision were:

- Language sensitivity. The primary job of the HP SoftBench editor is program editing. To be effective, the editor should be language sensitive. This ruled out the TextEdit widget in the X toolkit.
- Ease of learning and use. We wanted an easy-to-use mouse/menu interface consistent with the other HP SoftBench tools.
- Embedded edit capabilities. The HP SoftBench tools must prompt the user for information. These prompts may be embedded in the tool window. Without a difficult encapsulation, stand-alone editors like emacs or vi cannot be used in this way.
- Consistent interface. It was an HP SoftBench requirement that editing commands be consistent in all editable areas of all tools.
- Broadcast message support. To be a good citizen of the HP SoftBench environment, an editor should make its functionality available through messages and make other tools (e.g., the HP SoftBench program debugger) available from the editor.

To satisfy these requirements, we created the *edit widget* (see page 42). Given the edit widget, the editor is just a sophisticated wrapper on the existing functionality.

A user can configure the system to have edit requests serviced by another editor, but one or more of the goals of the system editor will be sacrificed. If vi is chosen, consistency and broadcast message support are sacrificed. If emacs is chosen, the consistent mouse/menu human interface is sacrificed.

In the HP SoftBench program editor, each file is given its own window and menus are used to give easy access to the editing functionality (see Fig. 1). Messages are used to communicate with the HP SoftBench program builder, static analyzer, and development manager tools.

The HP SoftBench editor is designed for editing programs, as opposed to general prose. It has knowledge of program structures like procedures, blocks, and tokens and can use this information to control indentation and balance delimiters.

The easy-to-use menu/mouse interface is particularly effective for writing code. In the HP SoftBench environment there are mouse functions for selecting tokens, blocks, and procedures and there are mouse functions for cut, copy, and paste operations. Because program code involves repeating similar structures, code can be written very quickly in the program editor by copying a structure that is similar to the needed code and changing it to the desired form.

For example, once code similar to what is needed is pasted into place, a typical change is in the names of variables. The edit widget replaces selected text with whatever is typed next. To make a change in the name of a variable that occurs several times, the first instance of the old variable name is selected by double-clicking the left mouse button. The new name is typed, replacing the old name. The new name is then selected and copied into the paste buffer. Additional instances are changed by selecting the old token and pasting the new token in its place.

Colin Gerety
Software Development Engineer
Software Engineering Systems Division

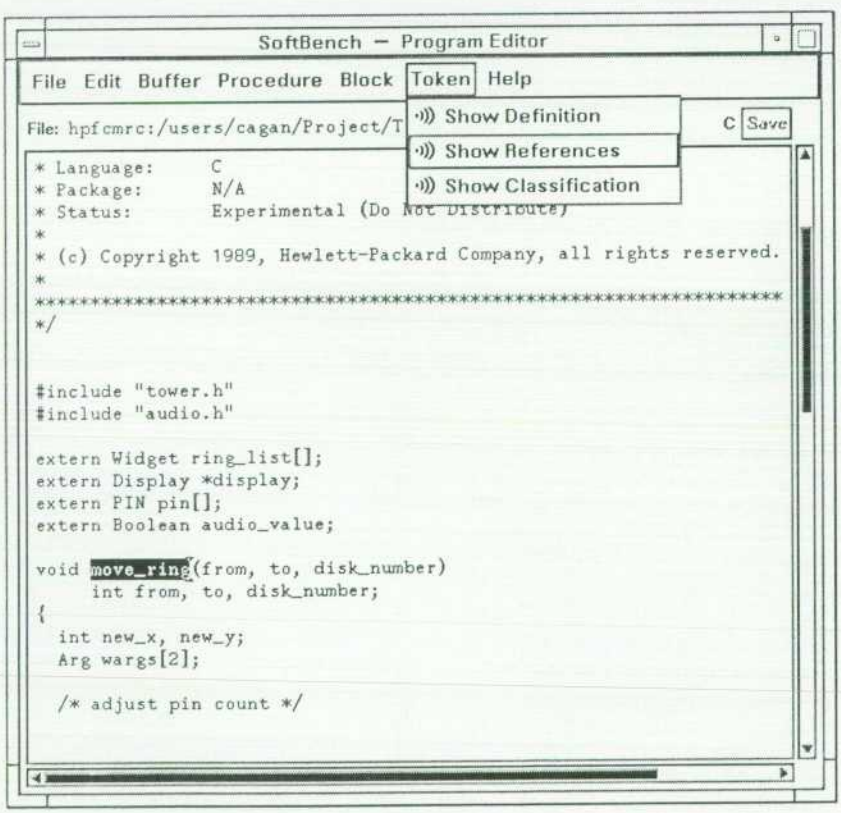


Fig. 1. *Program editor user interface.*

Program Builder

The HP SoftBench program builder is a tool that simplifies the compile and link phases of the typical edit/compile/link/debug loop used during software development (see Fig. 1). It shortens this part of the cycle by compiling only the parts of a program that require recompilation because of direct or indirect changes. In addition, the program builder provides an error browsing feature that facilitates direct access to identified source code errors, thereby speeding up the edit phase of this cycle.

In an effort to simplify further the task of building and maintaining a software project, the program builder provides facilities for creating and maintaining the dependency control files required for intelligent builds. This frees the user from having to deal with the format and content of such files.

Rebuild Only What Is Necessary

The program builder compiles only the parts of a program that have changed since the last build. If an include file common to several source files has changed, the program builder is smart enough to compile all files that depend on (i.e., include) that file. This ensures that any change will be correctly propagated through the entire software project.

In reality, it is not the program builder that performs these dependency control tasks. It is the build program invoked by the program builder that has these abilities. The program builder provides a friendly and consistent user interface to this underlying program. The UNIX automatic dependency control program make is the default program used to provide this functionality. Make does its job by using time stamps maintained by the file system to determine if source files have changed since the last time they were compiled. If any files (or dependent files) have changed, make will invoke the appropriate compiler to recompile only those files and relink them.

Although the default build program is make, almost any build program can be substituted. The program builder does not care what build program is run. This makes it very simple to adapt to new build technologies. The only thing that really needs to be known about the build program is how to pass compiler flag information through it to the compilers. The program builder must be able to control certain compiler behavior, including the generation of information required to debug a program, the generation of information required for static analysis, and the optimization of code for speed. This is necessary because the program builder message interface allows specification of parameters that request these behaviors. For example, the static analyzer tool will always request static information when it sends a build request to the program builder.

By default, the program builder passes compiler flags through well-defined environment variables that all HP compilers know to look at, effectively bypassing the build program. Therefore, almost any build program and related makefile should be usable without any changes to the program builder configuration or makefile.

For some build programs, such as the AT&T ToolChest program nmake, which keeps track of the compile options used to do a build, this scheme is not acceptable. Nmake will force all files to be recompiled if compile options change. Therefore, the program builder can be configured to pass compile options through the build program on the command line using almost any syntax that is appropriate for the build program or makefile.

Automatic Generation of Dependencies

For make to do its job, there must be a makefile that contains

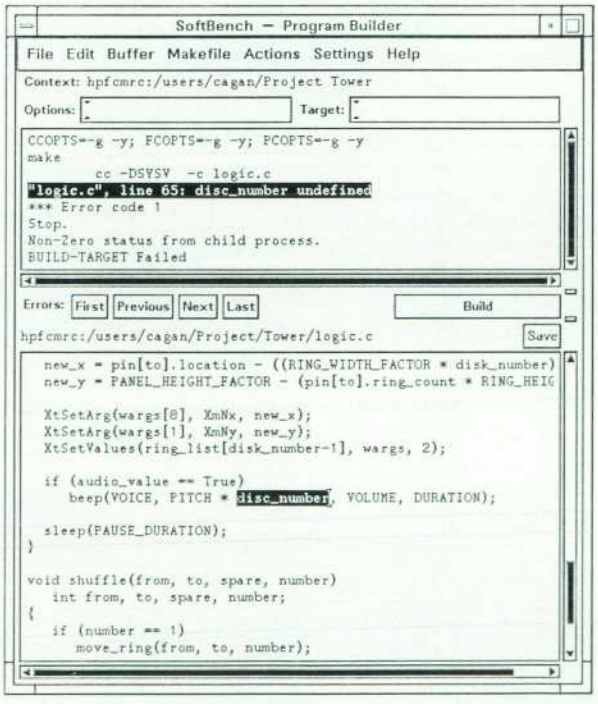


Fig. 1. *Program builder user interface.*

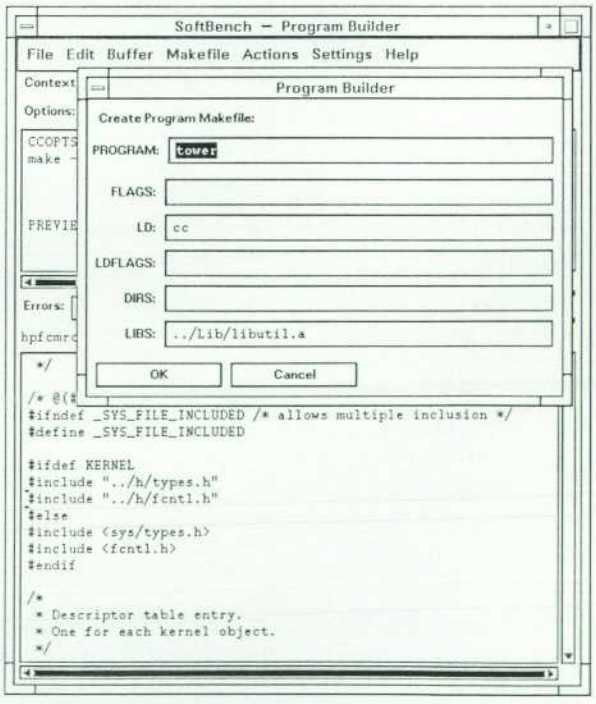


Fig. 2. *Creating a makefile in the program builder.*

a list of source files required to construct the program. The makefile must also contain dependency information (what files depend on what other files) so that efficient builds can be performed. For many UNIX users, for all but the simplest set of source files, makefiles are quite magical and for the most part, unintelligible. Maintaining them is a nightmare, and creating them is very difficult. The program builder provides a user friendly interface for the creation and maintenance of these files. It automates these tasks by using the HP-UX program mkmf (make makefile) for both creation and maintenance.

To create a makefile, the Makefile: Create Program menu selection is used. A simple fill-in-the-blanks form is then displayed in a window (see Fig. 2). All entries are optional, but the user typically will specify items such as an executable file and any extra libraries to be linked with the program. After the OK button is pressed, the program builder causes mkmf to scan all of the files in the context directory and construct a makefile based on the dependency graph generated from its analysis of the source files. The program builder supports creation of makefiles for archive libraries as well as for programs.

To update a makefile, the user simply selects the Makefile: Update menu selection. The program builder will then invoke mkmf to rescan all source files and update the dependency information in the existing makefile.

Makefiles can contain numerous targets for performing a variety of tasks other than building a program. Specifying one of these targets in the TARGET window of the the program builder and starting a build (pressing the BUILD button) causes the action associated with that target. Some of the targets provided in makefiles generated by the program builder include:

- print. Format all of the source files and output to the printer.
- lint. Run lint over the source files to check for syntax errors.
- clobber. Remove all reproducible files (.o files, program files, core files, etc).
- clean. Remove all object (.o) files.
- touch. Touch (update the time stamp of) all source files.

It is often desirable for more experienced users to customize the actions associated with these targets or to add new targets for additional functionality. To this end, a menu selection (Makefile: Edit) is available so that the makefile can be edited in the edit area of the program builder or in a separate window.

The program builder attempts to deal gracefully with build requests when no recognizable makefile is present. It is often the case that a single directory may contain numerous simple, self-contained programs. While it is currently not possible to instruct the program builder to build all of the programs in the directory without a makefile, requests to build individual programs are handled correctly.

Error Browsing

A useful feature of the program builder is the error browser. The program builder presents compilation errors to the user in a browsable list. It is a simple matter for a programmer to walk through the list of errors, fixing them one at a time.

Error recognition is based on regular expressions and therefore is easily extensible. The regular expressions are stored in a file and read into the program builder during initialization. The file supplied with the program builder contains expressions for all of the languages supported by the HP SoftBench system (HP 9000 Series 300 and Series 800 compiler errors are recognized). The user can supply a file to be used in place of the default file.

To be recognized, all errors must contain a filename and a line number. The error need not be specified on a single line, but may span several, as for the Series 800 Pascal compiler. The file that contains the regular expressions supports syntax that allows the user to specify how many expressions must be matched to recognize a line or set of lines as an error.

Each line received by the program builder from the build process is shown in the build output area and compared with the current list of known regular expressions. If it matches one of the expressions (or matches the last line of a multiple-line expression whose preceding lines have already been matched), the filename and line number are stored in the list of recognized error lines. Selecting any of these lines will display the associated source code line in the file view area. The error buttons (FIRST, NEXT, etc.) are available to cycle through the errors in an orderly manner. The program builder uses features supplied by the HP SoftBench edit widget (floating line marks) to ensure that error line references remain accurate as source code lines are inserted or deleted.

This facility is not limited to compiler errors. A useful example of its flexibility is a grep browser. The program builder can be configured to act as one by changing the build program (either via the menu or by a resource specification) to grep, specifying -n in the program builder OPTIONS window to ensure that grep generates line numbers in its output, and specifying a pattern and a list of files to search in the OPTIONS window. The regular expression for recognizing grep output is already present in the default regular expression file supplied with the program builder. Now, when a build is performed, all output from grep will be recognized as errors. Thus, selecting any of the output lines or using the error buttons (FIRST, NEXT, etc.) will display the selected file at the indicated line number in the file view area of the program builder tool. This flexibility can be used to create a browser for many tools.

Remote Builds

The program builder supports a simple distributed or remote build facility in addition to the standard HP SoftBench remote execution and remote data facilities (see "Distributed Execution, Data, and Display," page 40). This allows users to specify any machine in the network (to which they have access) to be used as a compile server. While this facility does not implement true distributed builds (builds where the various compiles and links needed to complete a build request are distributed across various computers on a network), it does allow the user to assign the compute and I/O intensive task of compiles and links to a machine that may be better able to handle these demands. Because of its simplicity, the program builder does not preclude true distributed builds. Any build facility capable of such a task can be substituted for the UNIX make program.

The HP SoftBench subprocess control (SPC) facility is used for automatic execution of the build process on the specified remote computer. Before the process is started, the SPC daemon on the remote machine establishes a data connection to the machine and directory specified by the program builder's data context. The current working directory for the build process is then changed to match that of the context host and directory, and the build is then performed as if it were run locally.

James W. Wichelman
Software Development Engineer
Software Engineering Systems Division

Static Analyzer

The HP SoftBench static analyzer aids the user in understanding source code. Fig. 1 shows its user interface. It supports language independent queries about code structure and provides cross-reference information that can help in finding defects, planning code changes, or evaluating a piece of software for reuse. Message communication with the development manager, program editor, and program builder enhances its ability to provide window-based, interactive analysis.

The static analyzer receives its information from the compiler, much the same as a debugger does. This has the advantage that source code is parsed once and the results are shared between static analysis, debugging, and program execution.

The compiler collects cross-reference information for the static analyzer on all identifiers within a program and categorizes each occurrence by how it is used. An assignment statement is categorized as a modification to the identifier being assigned to, while a variable definition is categorized as a definition. The static analyzer supports queries that return a single category or group of categories of references about an identifier. Often a user only wants to see where the value of a variable has been changed or how a function has been defined or declared in different modules. The static analyzer supports these queries directly, returning only the relevant information.

A program may have many identifiers that have the same name, but because they are different program elements or have different scopes, they refer to different objects. For instance, a structure field named slime may be defined within two different structures and therefore represent two distinct entities that should not be confused during analysis. When only the name slime is entered and its uses are requested, the static analyzer will return the uses of both fields because both are named slime and have uses. However, if the user identifies slime by selecting it within a source code view, then there is enough location information to indicate which field is wanted, and only the references to the selected slime will be returned.

Coupling these capabilities with program structure queries provides a tool streamlined for understanding and facilitating changes to software. By allowing the user to ask questions about a program, browse results to see program context, and use the built-in language-sensitive editing, the HP SoftBench environment provides the user with a productive environment for software understanding and software change.

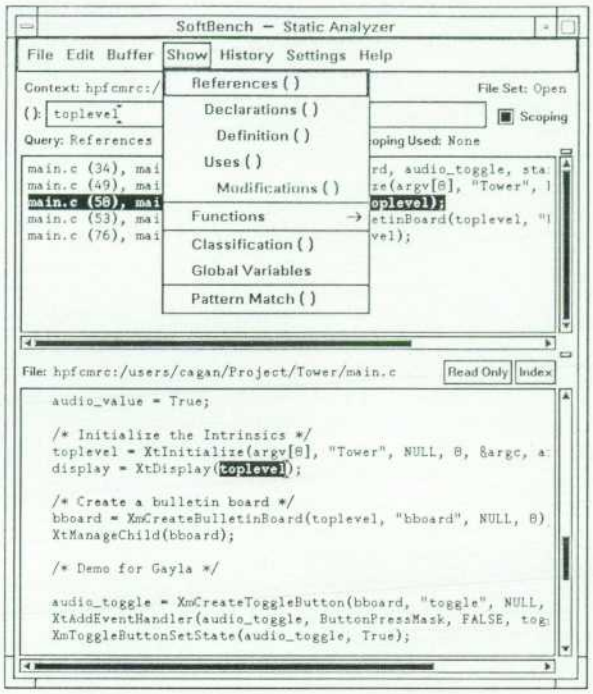


Fig. 1. *Static analyzer user interface.*

Gary L. Thunquest
John P. Dutton
Software Development Engineers
Software Engineering Systems Division

(continued from page 49)

Program Test

The program construction task results in an executable program. However, this does not mean that the program implementation perfectly meets its design requirements. The task of analyzing the program to identify and correct defects in implementation and design is known as the program test task. On large projects, this is often a very difficult process.

The HP SoftBench environment provides strong support for understanding both the structure (with the static analyzer) and the behavior (with the program debugger) of large, complex applications.

The program debugger (see box, page 55) provides program execution in a controlled environment. The user can step through the program, watching for specific conditions, pausing at any time to examine the state of data structures, and monitoring the control flow through the various paths of the program. If a variable is somehow being set to an illegal value, the user can trace the variable to locate the conditions and location of the improper assignment. If a function is being called when it shouldn't be, the user can monitor the execution, watching for the conditions which caused this call.

In analyzing the behavior of complex applications, it is often useful to view the execution at lower levels of abstraction. The program debugger can simultaneously show the program's source code, the assembly code, and the processor's register contents. The program debugger can walk through the program's execution either at the assembly statement level or at the source statement level.

The static analyzer is often used hand in hand with the program debugger. For example, if a variable is being set to an illegal value, the static analyzer identifies all locations where the specific variable is set, and the program debugger can be used to set tracepoints at each of these locations.

When a problem has been located and a change has been

(continued on page 56)

Program Debugger

The HP SoftBench program debugger provides a powerful yet simple user interface to the HP-UX symbolic debugger xdb, making users effective in their debugging tasks with a minimum of effort. We were surprised to learn how many users avoid using some HP-UX debuggers because of the difficulty of learning their command languages. These users would rather resort to some of the most tedious and time-consuming methods of tracking down simple bugs than master some esoteric tool.

Other goals for the program debugger were integration with the other HP SoftBench tools, provision of added value over the standard HP-UX symbolic debugger, and exploration of the potential for automated use of tools in the future.

User Interface

Using the HP SoftBench program debugger, software developers can become proficient at common debugging tasks in very short order, even if they are not familiar with the standard HP-UX debuggers. Pull-down menus and accelerator buttons, along with point and select operations with the mouse, provide a very simple means for entering powerful debugging commands (see Fig. 1). Almost all of the functionality of the standard HP-UX debugger is available via mouse-oriented commands. Many sophisticated sequences of commands are also made available with a single menu selection. When the user must use a more esoteric command, it is possible to type it in directly to the HP-UX debugger.

The program debugger extends the user's view of the program being debugged by allowing more simultaneous views than standard debuggers (see Fig. 2). The user can see program input/output, debugger input/output, source code, assembly instructions, register sets, and even the state of signals being handled by the

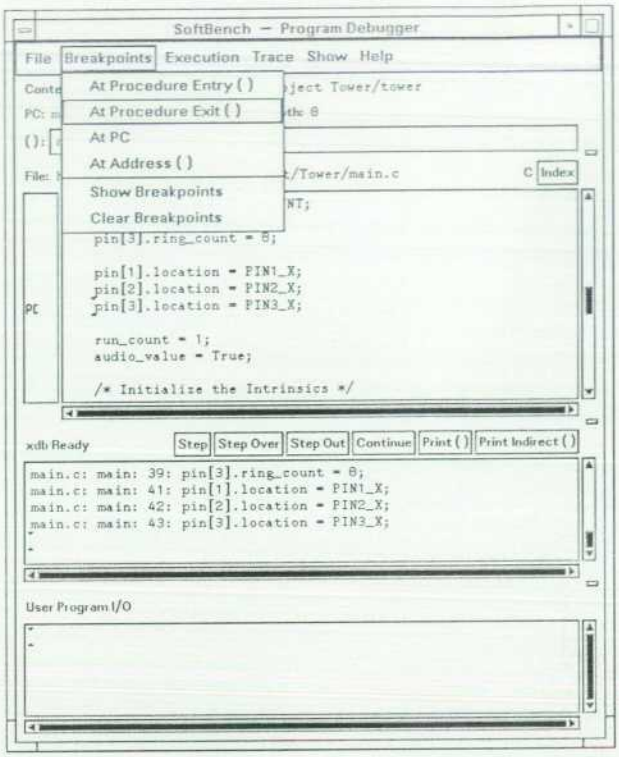


Fig. 1. *Entering program debugger commands.*

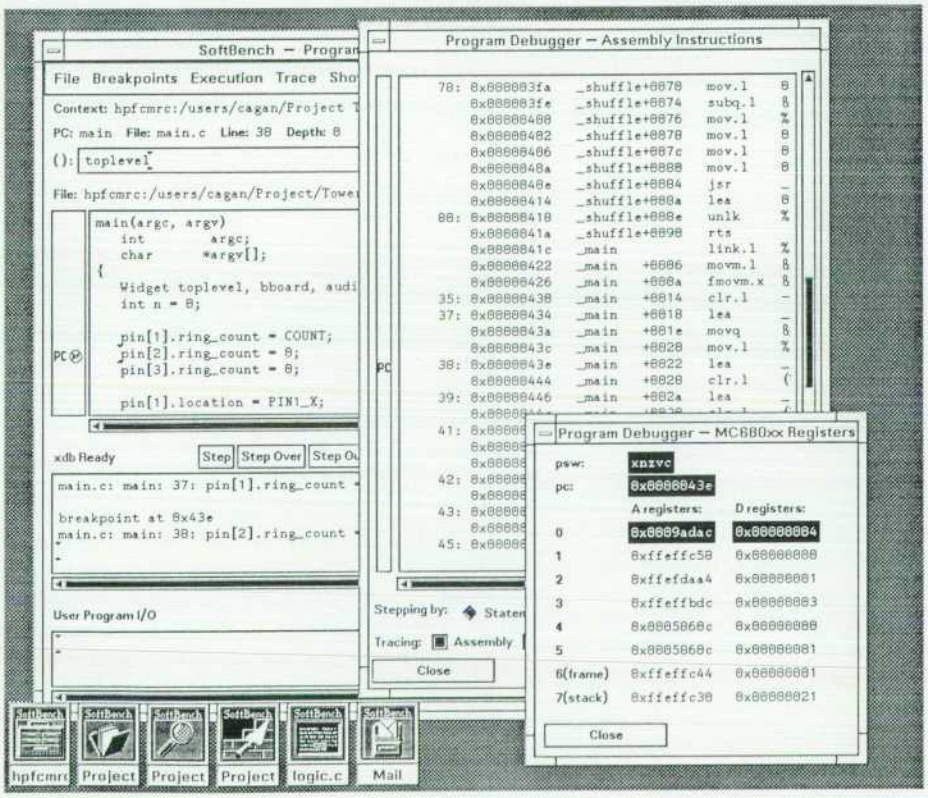


Fig. 2. *Multiple views of program execution in the program debugger.*

debugger—all at the same time if desired. Because these views are in separate windows, they are not constrained by the same space resources as in conventional, terminal-oriented HP-UX debuggers.

One of the most useful features is separation of the I/O streams for the HP-UX debugger and the program being debugged into separate window panes. This allows the user to see (and remember) what the program is doing without sorting through confusing debugger commands, prompts, and printouts. Each of these views is scrollable and editable, allowing the user to review and repeat previous entries.

Added Value

Since the program debugger is implemented using the HP SoftBench distributed support mechanisms, users can debug programs on other machines and with distributed source files without doing any extra work. The program debugger takes care of starting the HP-UX debugger on the correct machine (the one hosting the executable file) and establishes all the necessary interprocess communication. While this is not a substitute for nonintrusive distributed debugging, it does satisfy the distributed debugging needs of many users.

For applications with signal handlers that must be debugged, the program debugger has a special window to help the user monitor signals that are received, handle each signal specially, and send specific signals to the application. All of this can be done without going to a terminal window and without looking up process ID numbers and signal numbers.

Program debugger users rarely need to know about process ID numbers, even when sending signals to their applications or when adopting already running processes. The task of remembering the ID numbers when a process is forked or of looking up the process ID number (with /bin/ps) using the program's name is handled automatically.

The standard HP-UX symbolic debuggers give minimal support for monitoring the value of program variables. It is possible to set up assertions that show the value after each instruction executed (slowing execution incredibly), or to set up breakpoints to show the value at specific points in the program, but great imagination (and tedium) are required to do much more. The HP SoftBench program debugger allows the user to specify a variable to be traced and then uses a rather involved set of breakpoints and assertions to implement a primitive but often useful variable tracing facility.

Automated Tool Use

The HP SoftBench tool integration architecture allows tools to communicate with each other by means of request and notification messages. While the current HP SoftBench tools only document a limited set of requests, it is possible for tools such as the program debugger to be controlled entirely by a program rather than a person. For example, tool builders can use the HP Encapsulator (see article, page 59) to construct higher-level tools that issue requests to the static analyzer and the program debugger. These higher-level tools can monitor program data structure references or modifications, perform branch flow analysis, monitor performance, or handle dozens of other useful operations.

Robert A. Morain
Robert B. Heckendorn
Software Development Engineers
Software Engineering Systems Division

made, it is important to be able to assess the impact of the change. For example, if a function needs an additional argument, all calls to the function must be located and modified. If a function's return value must be changed, all locations in the project that use this function's return value must be identified and changed. The static analyzer facilitates these activities.

Program Maintenance

Program maintenance is similar to program test. Software requirements change over time, and modifications may be needed to the program for reasons other than defects in the program. The programmers who maintain applications are often not the original developers and may need assistance to understand the design and implementation of the program so that they can make the necessary modifications effectively. These factors make the ability to understand the application crucial. In fact, much of what software maintenance programmers do is work on understanding the applications they maintain and assess the impact of proposed changes. While the changes themselves are often small, identifying the source of the problem, designing an appropriate change, and assessing the impact of the proposed change is often very difficult and time-consuming.

The static analyzer and the program debugger are the primary tools for helping the maintainer understand the application, identify the problem, and assess the impact of proposed changes. The program editor and the program builder are used to reconstruct the modified application and the static analyzer and the program debugger are used to test the changes. If the modifications to the application caused a change in the module dependencies (for example, if a new module was added), then the program builder will update its data base of dependencies (the makefile).

In addition to locating specific conditions in the program, the static analyzer and the program debugger help the program maintainer work backwards from the symptom to the cause. For example, if the user of an application reports a problem when a specific error message is displayed, the program maintainer can ask the static analyzer to show the locations in the program where the message is displayed. The program debugger can then be instructed to monitor those points as the maintainer reconstructs the scenario that led to the error message. Once the problem situation is duplicated, the maintainer uses the program debugger to examine the state of the application and determine the function call sequence and data structure state that caused the defect situation.

Conclusion

We have described how the HP SoftBench environment is used to support team file management, team communication, program construction, program test, and program maintenance. The goal was to illustrate not only the features of each of the HP SoftBench software development tools, but to demonstrate the synergy that can be achieved by letting the tools collaborate to provide a task-oriented software development environment.

With the HP Encapsulator product described in the article on page 59, common development activities involving

(continued on page 58)

Integrated Help

The HP SoftBench help facility (see Fig. 1) is independent of the tools for which it provides help. For efficiency, the help application is combined with the tool manager.

Each tool contains a help pull-down menu containing Item Help and Application Help entries. When a user asks for Item Help, the mouse sprite changes into a question mark. The user can then point at a region of any HP SoftBench application and help text describing that part of the tool will appear in the help window.

Each piece of text is displayed with a list of related topics. Selecting the related topic causes the text for that topic to appear. The cross references can point to topics inside another tool.

The normal HP SoftBench intertool communication mechanisms are used to drive the help system. When a user selects Application Help, a request is sent to the help system to display the information. The tool is not aware of its own or any other tool's help text.

The help data base and communication between the other HP SoftBench tools and the help tool (the Item Help lookup mechanism) require no cooperation from any application as long as it is implemented with the HP widgets and the X toolkit (which stores the needed properties on the widget windows). The help menu items do require minimal cooperation from the application, of course. Ideally, dependence on the HP widgets would not have been needed, but the information provided by applications complying with the *Inter-Client Communication Conventions Manual* (see box, page 23) or by the X toolkit (Xt Intrinsics) is not sufficient for this level of detail.

Interface between Tools and the Help System

HP SoftBench tools communicate with the help system via the X protocol and the HP SoftBench broadcast message server. Requests for Item Help and Application Help are sent as request messages to the help tool. A request for Application Help includes the application class of the application, and a request for Item Help causes the help tool to grab the pointer (changing the pointer to a question mark temporarily). For Item Help, the help tool has to figure out which widget was selected. This is done without the assistance of the specific application as long as the application is built using HP widgets. Each widget window stores a property XW_CLASS, which contains the widget name and the class of the widget window. The top-level window also stores the WM_CLASS property containing the name and class of the application (as defined by the *Inter-Client Communication Conventions Manual*). When the user selects a window, the help tool determines the smallest enclosing window containing the pointer and then traverses the window hierarchy outward to determine the widget namelist and classlist used in a resource specification in the X resource manager. It looks these up in the help data base to locate the help text associated with the help window. This same mechanism is used by the automated test facilities described in "Architectural Support for Automated Testing" on page 37.

John R. Diamant
Software Development Engineer
Software Engineering Systems Division

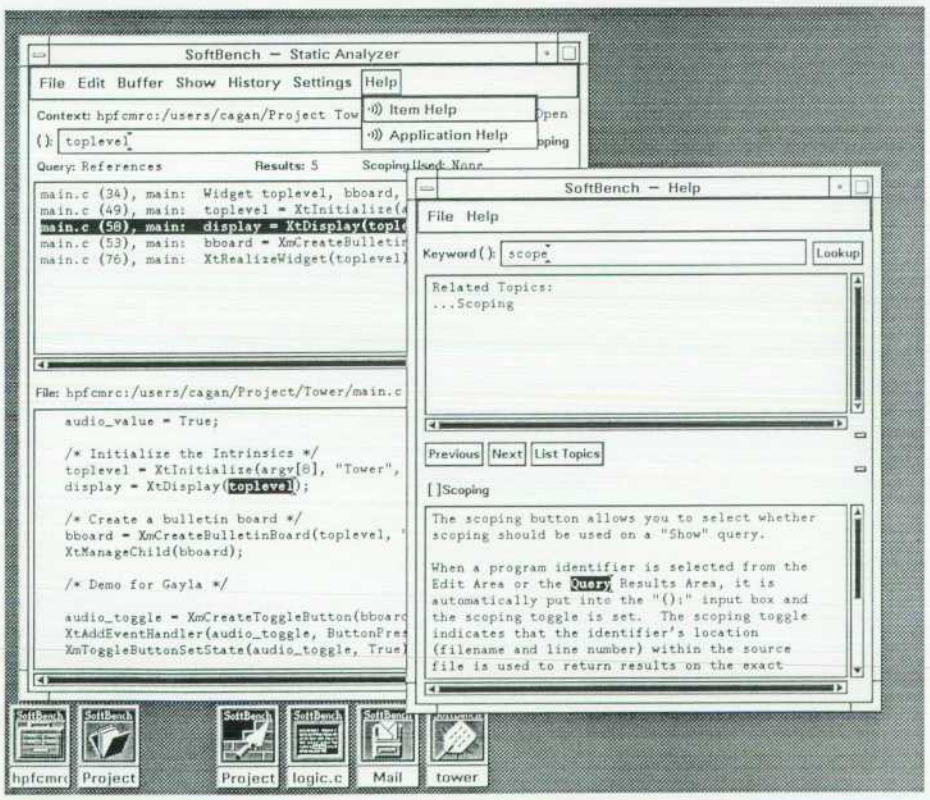


Fig. 1. *HP SoftBench on-line help.*

the HP SoftBench tools can be automated.

Acknowledgments

The HP SoftBench software development tools were designed and built by Martin Cagan, John Dutton, Jorge Gautier, Robert Heckendorn, Caroline Koff, Bob Morain, John Repko, Nancy Steffens, Gary Thunquest, Anthony Walker, Jim Whalen, and Jim Wichelman. The HP SoftBench team is grateful to the HP California Language Laboratory for providing the underlying support for static analysis, and to the HP Colorado Language Laboratory for their help with the underlying debugger support.

HP Encapsulator: Bridging the Generation Gap

By means of the Encapsulator description language, a user can integrate tools into the HP SoftBench environment without modifying their source code, and can tailor the HP SoftBench environment to support a particular software development process.

by Brian D. Fromme

THE HP ENCAPSULATOR is the tool integration and process specification facility of the HP SoftBench environment. It allows an HP SoftBench user to promote existing tools to be fully consistent, integrated HP SoftBench tools and to tailor the HP SoftBench environment to support a specific software development process. The HP Encapsulator provides customization and extension capabilities for automating organization, team, and personal software development processes using event triggers.

Integrating Existing Tools

The HP Encapsulator can handle a range of existing applications. It is designed to handle programs written in the style of programs for the UNIX* operating system, that is, programs that have a command-line interface to their functionality. Examples of this sort of program are nearly all UNIX tools (tar, prof, adb), customer-developed scripts and utility programs, and many third-party tools (e.g., McCabe's ACT, Verilog's Logiscope, Softool's CCC, and SMDS's Aide-de-Camp).

From the user's point of view, an encapsulated tool looks and behaves just as the core HP SoftBench tools do. In fact, one of the core HP SoftBench tools is actually an encapsulation—the HP SoftBench mail tool is an encapsulation of the HP-UX tool mailx. This encapsulation will be described in more detail later in this article.

The HP Encapsulator can be used either to add a new tool to the HP SoftBench environment or to replace an existing HP SoftBench tool or another encapsulated tool. The HP SoftBench architecture is designed to facilitate this substitution of tools.

Tool Encapsulation Overview

Encapsulating a tool means integrating the tool into the HP SoftBench tool integration architecture. The HP Encapsulator is the liaison between the existing tool and the rest of the HP SoftBench environment. It plays the role of translator of commands, actions, and presentation.

*UNIX is a registered trademark of AT&T in the U.S.A. and other countries.

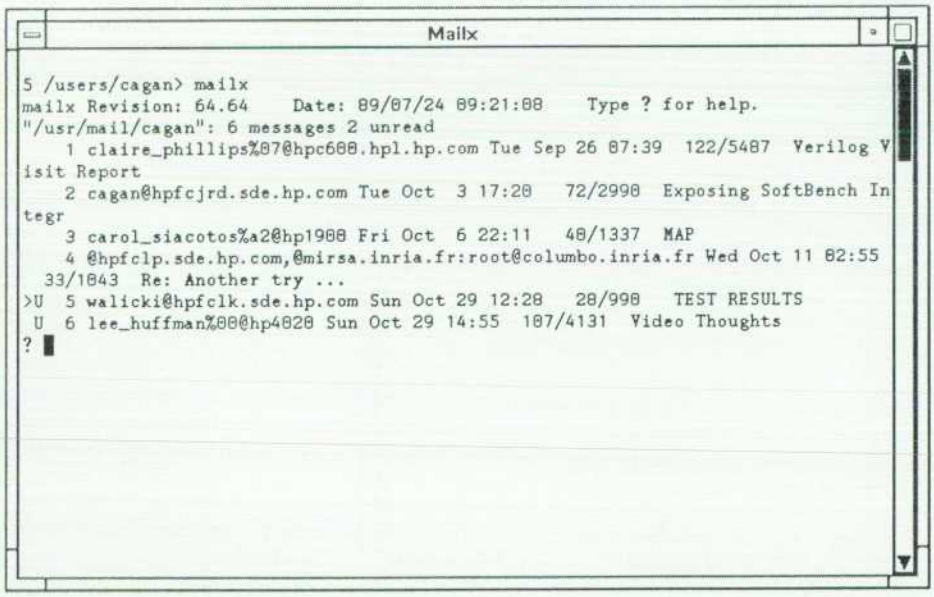


Fig. 1. *Original user interface of the HP-UX terminal-based mail tool* mailx.

Integrating a tool using the HP Encapsulator provides the following benefits:

- Provides a link to the HP SoftBench event trigger facility.
- Provides an HP SoftBench-compatible, OSF/Motif-style user interface (see article, page 6).
- Uses HP SoftBench distributed execution to support remote subprocess execution.
- Uses the HP SoftBench network-wide communication facility.

An important aspect of the HP Encapsulator is that no source code modifications are necessary to the tool being encapsulated. This allows customers to integrate purchased tools for which no source code is available.

There are also some limitations of encapsulation that should be understood before an encapsulation program is attempted. The HP Encapsulator only supports encapsulation of tools written in the UNIX command-line interface model. Tools that have highly interactive or graphical user interfaces are often not good candidates for encapsulation because the HP Encapsulator cannot understand what the tool is doing or has done. User interface potential is also limited by output from the encapsulated tool. If the tool does not provide error messages or some sort of output stream (typically stdout or stderr), the HP Encapsulator is constrained in its ability to interpret what the tool has done. Another limitation is that event granularity for triggers and notifications—that is, the level of detail at which events can be specified—is only as fine as can be initiated and recognized from the encapsulated tool. To achieve the same level of event granularity as the other HP SoftBench tools, each atomic operation needs a unique command-line interface.

The HP SoftBench Mail Encapsulation

Before the design details of the HP Encapsulator are described, a sophisticated encapsulation will be presented to illustrate the concepts that have been presented so far.

The encapsulation to be described is the HP SoftBench mail tool. This tool is provided with the HP SoftBench tool set, and most users are not aware that it is an encapsulation rather than a native tool like the other HP SoftBench tools. The HP SoftBench mail tool is actually an encapsulation of the HP-UX mailx program. The mailx program was not modified in any way.

Fig. 1 shows the original terminal-based interface to mailx, and Fig. 2 shows the encapsulation. Before encapsulating, mailx was not related or linked to the other HP SoftBench tools in any way. The encapsulated version, on the other hand, has useful links to the other HP SoftBench tools. For example, the user can configure HP SoftBench mail to send a mail message to the project team whenever a project build has completed successfully. With the HP Encapsulator, customers can customize the specific conditions and actions to meet their particular needs.

Other Encapsulations

Many other tools have also been encapsulated. Fig. 3 shows the encapsulation of the HP-UX performance profiling tool prof(1). Fig. 4 shows an experimental encapsulation of the Analysis of Complexity Tool for metrics collection and structured testing support from McCabe and Associates. Fig. 5 shows a trigger panel with which the user

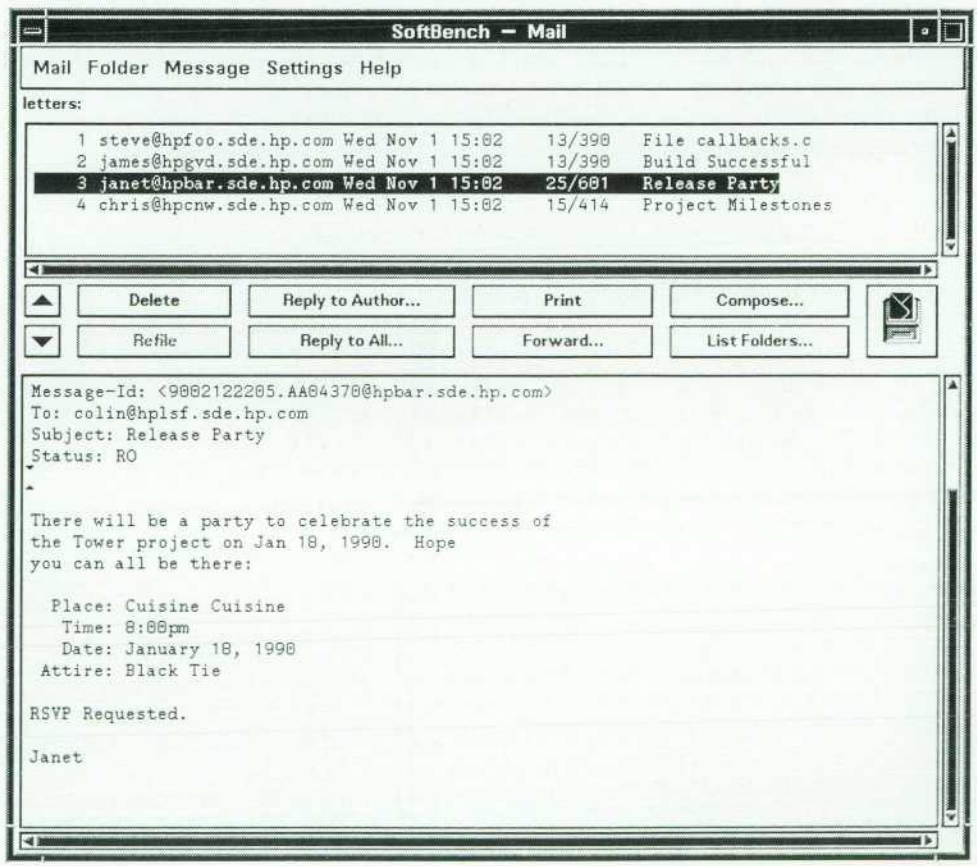


Fig. 2. *User interface of the HP SoftBench mail tool, an encapsulation of* mailx.

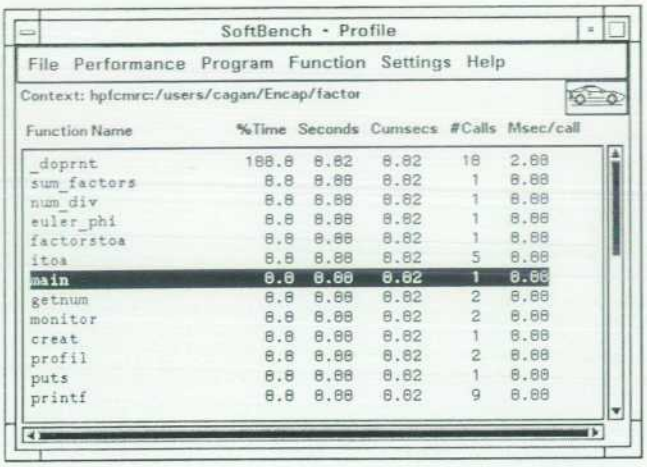


Fig. 3. *Encapsulation of the HP-UX performance profiling tool* prof.

can configure the relationship between the McCabe testing tools and the rest of the development environment.

Prototype encapsulations have been written for configuration management, documentation, testing tools, and language-based environments such as Lisp and Ada.

Encapsulator Description Language

The *Encapsulator description language* (EDL) is a specification language designed to simplify the task of describing an encapsulation. The primary reason for encapsulating a tool into the HP SoftBench environment is to allow that tool to make use of the HP SoftBench architecture, primarily the broadcast message server and aspects of the distributed environment. Therefore, these architectural features have been made accessible in EDL.

From the perspective of the encapsulation system, there are two main components in an encapsulation: interfaces and actions. Interfaces are connectors to the outside world, such as the window system or the HP SoftBench message system. Actions are the steps to be taken when certain conditions are met on an interface. EDL defines a set of interfaces and data types that link conditions in an interface to actions that the user provides to respond to that condition. The actions are EDL code to be executed.

EDL has conventional programming language constructs such as data types, variables, operators, flow-of-control clauses, and user-defined functions. EDL also contains a rich set of built-in functions, which provide a programmatic interface into the HP SoftBench architecture as well as the underlying window system. The EDL data types are string, integer, Boolean, attribute, event, and object. There are C-like operators that can be used to form expressions. There are two flow-of-control clauses: if and while. User-defined functions can be used to group and define parameters for EDL statements, but are most useful as actions for responding to conditions.

A programmer develops EDL code much as one would develop code in other specification-language-based environments, that is, by first entering the EDL source text into a file, then invoking the HP Encapsulator over that file. See page 67 for a description of how the HP Encapsulator executes the EDL code.

Interfaces

Four interfaces are defined in the Encapsulator description language. They are the user, message, application, and system interfaces. The user interface is the window system, the message interface is the HP SoftBench broadcast message server, the application interface is the encapsulated program or subprocess, and the system interface is the operating system. Conditions on an interface are called *events*.

An event is a data type in EDL. Events have three components, a *type*, a *pattern*, and an *action*. The type defines the interface to which an event corresponds. The type specifier is the identifier or the name of the interface. Thus the user interface has events of type user. The pattern is the condition to be met on an interface. A pattern is a string that identifies either the name or the form of a condition. For example, application events use the HP-UX regular

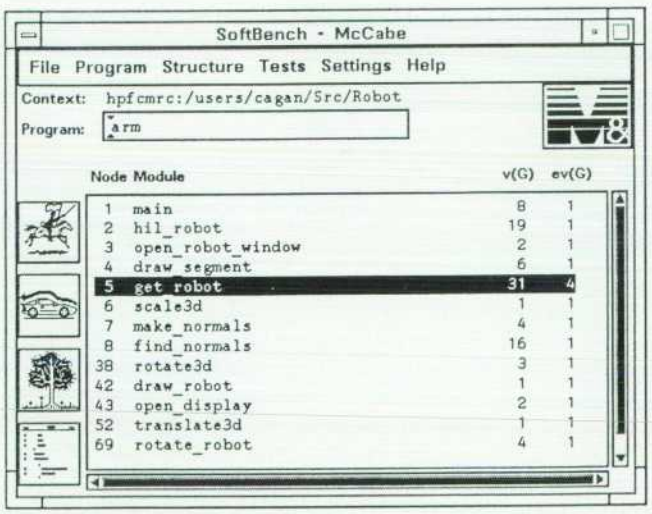


Fig. 4. *Encapsulation of the Analysis of Complexity Tool (ACT) from McCabe and Associates.*

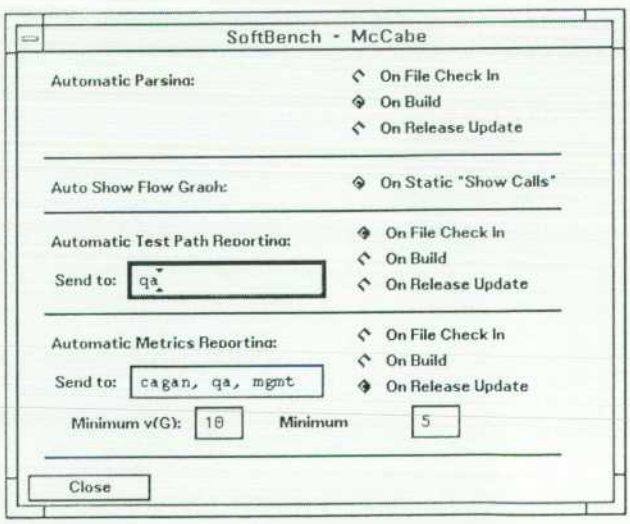


Fig. 5. *Triggers for the encapsulation of McCabe's ACT.*

expression pattern matching facility, so the pattern is a string that describes a regular expression to that pattern matcher.

An example should help clarify how an event is declared. This event will occur when the encapsulated program mailx finds that there is no new mail available.

First, we declare an event variable.

```
event mailx_event;
```

Now we assign that variable a newly created application event. The event corresponds to the condition that mailx has written the text "No mail for Fromme" to stdout (its standard output file).

```
mailx_event = make_event(Application, "^No mail for (.*)$0\n",
              no_mail($0));
```

Finally, we add this event into the list of active events.

```
add_event(mailx_event);
```

Events can be activated and deactivated via built-in functions in EDL. This allows the user to control which conditions can be met at a given time. When an event is defined in the user interface, that event must correspond to a particular object in the window system. EDL objects will be explained in more detail later.

Actions

Actions are the steps to be taken when certain conditions are met on an interface. In the example above, the action to be taken when the regular expression is matched is a call to the function no_mail(). Actions can be arbitrary statements of EDL code. In use, however, actions are typically calls to functions that are defined by the user. This makes the declaration of an event more readable and more easily changed or configured.

For example, we will define the function no_mail(), which is to be called when the HP Encapsulator sees the no-mail pattern from mailx. Note that the argument passed to the no_mail() function from the event is $0. This is the EDL syntax for the special string variables that retrieve portions of a regular expression. In this case, $0 will be the name of the user running the SoftBench mail tool.

```
/*This function is called when mailx tells us there is no new mail */
function no_mail(user)
    string user;
{
    /* Let the user know that there is no new mail in the mailbox */
    clear(headers);              /* Clear mail headers list */
    freeze_buttons();            /* Make the buttons insensitive */

    /* This is a local string variable that holds the dialog prompt */
    string info = print_to_string("User %s has no new mail", user);

    /* Now pop up the dialog box to inform the user */
    error(Information, info);

    /* No need to return data from this function */
}
```

Notice that the action function calls other functions, some of which are built-in EDL functions and some of which are user-defined functions. This example shows how we have taken a condition from the terminal based mailx application and turned it into an information dialog box via a call to the built-in EDL function error().

User Interface

The user interface is the window environment in which a user can interact with the encapsulation. Components of the user interface are referred to as *objects*. An object is any visual device that conveys information between the user and the program—examples are labels, buttons, menus, and editable fields. An object is also a data type in EDL. Objects are EDL representations of physical entities on the screen. There are two types of EDL objects: *manager* and *primitive*. Manager objects can control other objects while primitive objects cannot. The set of EDL manager objects includes Toplevel, Transient, Pulldown, and Pane, while the set of EDL primitive objects includes MenuButton, MenuSeparator, Command, Label, Edit, List, Toggle, and Image.

To define a user interface in EDL, one specifies the hierarchy of manager and primitive objects that compose each window as EDL statements. The following example will create a window with two objects, a Label object and an Edit object.

First, we declare the window and its first pane, both of which are manager objects.

```
object mail_window = make_manager(NULL, Toplevel,
                                  "MailWindow");
object pane = make_manager(mail_window, Pane, "firstPane");
```

Now we declare the components of the pane, both of which are primitive objects.

```
object target_label = make_object(pane, "target", Label,
                                  "Target: ");
object target_value = make_object(pane, "value", Edit,
                                  get_context_file());
```

In this example, the elements of a Toplevel window have been described. Manager objects are declared with the built-in EDL function make_manager(), while primitive objects are declared with the built-in EDL function make_object(). The parameters of these built-in functions describe information about how to create such an object when the time comes, such as the parent (or manager) of this object, the name, the type, and the label (or initial value to be displayed within the object). Objects described in this way are not created until their entire window is needed. This is achieved via a call to the built-in EDL function display().

To take an action in the user interface, the user often presses the left mouse key (mapped to the Select action) while the mouse cursor is over the object that describes the action. To define the action in EDL, the programmer must associate a user event with a user interface object. Thus, each user interface object is associated with distinct events or actions. Several objects can be associated with a single event. The following example creates three Toggle buttons and two events. A Toggle button is a user interface

object that represents either an on or an off state. The two events are used to determine when the user turns the button on and off.

First, we create two events that trigger actions for any of the three Toggle buttons.

```
event toggle_on = make_event(User, "Select", toggle(True));
event toggle_off = make_event(User, "Release", toggle(False));
```

Then we declare three Toggle button objects.

```
object a, b, c;
```

Notice that we pass both events to each of these objects.

```
a = make_object(pane, "a", Toggle, "Me", NULL, toggle_on,
    toggle_off);
b = make_object(pane, "b", Toggle, "Myself", NULL, toggle_on,
    toggle_off);
c = make_object(pane, "c", Toggle, "I", NULL, toggle_on,
    toggle_off);
```

This function handles both the on and the off states for these Toggle buttons.

```
function toggle(on)
    boolean on;
{
    /* Get the object with which this event is associated */
    object this_button = self();
    /* If we selected one of these buttons, then release the
     * others. This makes the buttons exclusive (only one can
     * be set at any given time). */
    if (on) {

      /* Send the "Release" event to all buttons but this one */
      if (this_button != a) send_event(a, toggle_off);
      if (this_button != b) send_event(b, toggle_off);
      if (this_button != c) send_event(c, toggle_off);
    }
}
```

This example defines a window object and its action.

Another aspect of the user interface is the appearance of an object, such as width, height, color, and character font. In EDL, these characteristics of an object are referred to as *attributes*. An attribute is a data type in EDL. There are two attribute operators: merge and associate. The merge operator is used to combine a single attribute into a set of attributes called an attribute list. The associate operator is used to combine a value with a named attribute. As can be seen in the following example, the WIDTH attribute allows an associated value, while the SINGLELINE attribute does not. The example illustrates the creation of a single-line, labeled, editable field of a certain width.

```
object label, edit;
attribute attr;
```

The following attribute merges a font description with a specified width and also tells the Label object to put the text as far left as possible.

```
attr = FONT : "hp8x16" | WIDTH : 100 | LEFTJUSTIFIED;
label = make_object(pane, "dirLabel", Label, "Directory:",
        attr);
```

The following attribute merges a font description with a specified width and also tells the Edit object to restrict its view to a single line.

```
attr = FONT : "hp8x16" | WIDTH : 300 | SINGLELINE;
edit = make_object(pane, "dirValue", Edit,
       get_context_directory(), attr);
```

Attributes are used to specify the appearance and behavior of user interface objects. The example above shows their static use, that is, to specify the object's behavior when first displayed. Attributes can be used dynamically through the EDL built-in function add_attribute(). This allows the appearance of an object to change during execution of the encapsulation.

Message Interface

The message interface is the programmatic access to a tool's functionality. This interface is the connection to the broadcast message server and allows tools to communicate with one another, just as a user would interact with a tool through the user interface. The messages that a tool can emit and receive define the tool's message protocol. The HP SoftBench tools have a predefined message protocol. When a user encapsulates a tool, a new message protocol is defined. This protocol is called the tool *class*.

To define a new tool class, one must decide what functionality of the new tool should be accessible to other tools. Most often this is the same functionality that is available to the user via the user interface. Next, the developer must decide how other tools are to be passed information specific to each tool function. Typically, information that the user interface receives by bringing up a dialog box can be received in the message interface as the data fields of a message.

There are several requirements that should be met for a new tool class to become a "good citizen" HP SoftBench tool. A *notification* message must be announced whenever the tool successfully performs a function. A *failure message* must be announced whenever the tool attempts to perform a function, but does not successfully complete it. A *request* message must be accepted for each function that the tool is able to perform.

The message model allows tools to request other tools to attempt to perform functions. Furthermore, because tools send out notification or failure messages after attempting to perform a function, a tool can determine the results of such a request. Thus, tool interaction can be either synchronous or asynchronous. For example, a tool may request that an edit of a particular file be started, but may not care whether the editor can actually perform the task. On the other hand, if a tool requests that a file be checked out of the version control system, it will need to know whether that function can be performed before continuing with the current operation.

To facilitate handling arbitrary requests, the broadcast message server defines a simple pattern matching facility. This facility is accessible through the EDL built-in function make_message_pattern(). When message events are defined, the message patterns will be passed to the broadcast message server. These describe the forms of messages to be forwarded to the HP Encapsulator.

The following example describes the message interface for the simple tool class EXAMPLE.

```
/* Define the tool class */
tool_class("EXAMPLE");
```

These are variables used in the message interface.

```
string pattern;
event plan, estimate;
```

This describes the PLAN request message.

```
pattern = make_message_pattern(Request, NULL, "PLAN");
plan = make_event(Message, pattern, plan_request());
```

This describes the ESTIMATE request message.

```
pattern = make_message_pattern(Request, NULL, "ESTIMATE");
estimate = make_event(Message, pattern, estimate_request());
```

Now we activate these events.

```
add_event(plan);
add_event(estimate);
```

These are the functions for the PLAN and ESTIMATE messages. The PLAN message takes two data parameters: the name and the engineer-months. The ESTIMATE message takes no data parameters.

```
function plan_request()
{
   string name, months;

   /* Extract and check the data parameters */
   name = message_data(1);
   months = message_data(2);

   /* If these aren't passed, then its an error */
   if (!name || !month) protocol_error();

   else
      /* Perform the PLAN request */
      if (plan(name, month))
         /* Succeeded */
         send_message(Notify, NULL, "PLAN", name, months);
      else
         /* Failed */
         send_message(Failure, NULL, "PLAN", name, months);
}

function estimate_request()
{
   /* Perform the ESTIMATE request */
   if (estimate())
      /* Succeeded */
      send_message(Notify, NULL, "ESTIMATE");
   else
      /* Failed */
      send_message(Failure, NULL, "ESTIMATE");

}
```

In this example, we have declared a tool class EXAMPLE and registered two patterns with the broadcast message server, each of which has a function to handle the request when the corresponding message is forwarded to the HP Encapsulator. The functions check that each message has the appropriate data and call other functions to attempt the requested action. If the actions succeed, then a notification message is sent to the broadcast message server. Otherwise, a failure message is sent.

By knowing the message protocols of a tool, one can use the HP Encapsulator to:

- Create a tool that interacts with other HP SoftBench tools.
- Create a tool that drives other HP SoftBench tools.
- Create a tool that replaces an existing tool (substitution).
- Create agents. Agents are tools that orchestrate other tools to perform tasks.

Tool Triggers

A *trigger* is a cause-effect relationship between tools. In the HP SoftBench environment, a trigger occurs when a notification or failure message is sent from one tool and one or more other tools respond to that notification by taking some new action. For example, when a file is saved from any tool, the HP SoftBench development manager tool will update its directory listing, if needed. This is a predefined trigger in the HP SoftBench environment.

The HP Encapsulator allows the user to define two types of triggers; those that take action in a tool and those that request some third tool to take an action. The HP SoftBench development manager example above is a trigger that takes action in a tool. The following code is an example of a trigger that requests a third tool to take an action. It listens for a notification message from the development manager and asks the HP SoftBench program builder tool to attempt a build.

When we see a VERSION-UPDATE-DIR message notification, send out a BUILD-TARGET request.

```
event trigger;
string pattern;
```

This pattern and event describe the message to be seen.

```
pattern = make_message_pattern(Notify, "DM",
            "VERSION-UPDATE-DIR");
trigger = make_event(Message, pattern, request_build());
add_event(trigger);
```

This function requests that a build be started when an update of the version directory is done in the development manager.

HP Encapsulator CASE Case Study

Frederick Brooks wrote: "Plan to throw one away; you will, anyhow." The day before the U.S.A. announcement and demonstration of the HP SoftBench environment, we threw away the mail tool. Work began immediately to rewrite it almost completely, using ideas we had learned from the previous effort. This time, instead of writing an entirely new mail program, our approach was to use the HP Encapsulator to encapsulate the HP-UX mail program mailx.

Why Another Mailer?

There were several reasons for writing yet another mail program:

- Provide a bridge from HP SoftBench messages to mail messages. When a particular HP SoftBench message is sent, a software developer might want to send mail to notify the team, the developer, or others. This would be especially true for processes that run unattended or at night.
- Improve the user interface. Most mailers that run under the X Window System have a human interface best described as nonideal. Some confuse new users with clutter, some require a lot of customization, and some are hard to maintain in a changing environment. HP SoftBench mail tries to provide a better user interface.
- Teach the HP Encapsulator language by a nontrivial example. We wanted future developers to learn advanced techniques by studying the code and its comments.
- Fine-tune the HP Encapsulator. The new technology of the HP Encapsulator needed to be used to get it ready for commercial use.
- Improve the usefulness of the HP SoftBench environment. Studies show that technical users buy a computer to solve important or costly problems like software development, not for office automation tasks like mail. Nevertheless, they expect the manufacturer to provide a mail system.

Using the Encapsulator

The Encapsulator description language (EDL) is a new language. A new language allows one to sail the seas of new higher-level ideas, explore uncharted waters of new constructs, and breeze past the rocks of low-level details so prominent in such libraries as the X toolkit. However, new languages usually include a new paradigm—a new way of thinking about the programming problem. This is certainly true for EDL.

Encapsulating a complex tool like the HP-UX mailx application has its challenges, too. As we progressed on the rewrite, we learned more about mailx that caused us to modify our design. For instance, the code to manage folders was rewritten twice as we learned subtle interactions in the way that mailx handles folders. Even though we did have access to the source files for mailx, we looked at them only once—to discover that we couldn't find the answer in the code! It turned out to be much easier simply to set up conditions, run mailx in a terminal window, and observe its behavior. On the other hand, encapsulating an existing program is code reuse at its best. Someone else had already solved hard problems of mail delivery, folder management, alias creation, message presentation, and so forth. To paraphrase Isaac Newton, HP SoftBench mail sees farther than its predecessors because it is standing on the shoulders of the giants that already provided part of the solution.

Bob Desinger
Software Development Engineer
Software Engineering Systems Division

```
function request_build()
{
   /* convert a wildcard for directory to a nil */
   string mfile = message_file();
   if (string_compare(mfile, "*")) mfile = "-"; /* Message
                                                 * server nil */

   /* Get the current context */
   host = get_context_host();
   dir = get_context_directory();

   /* Now set the context from the incoming message */
   set_context(message_host(), message_directory(), mfile);

   /* Send the request to the BUILD tool class */
   send_message(Request, "BUILD", "BUILD-TARGET", "-", "-", "-");
}
```

The ability to define triggers allows the user to customize the HP SoftBench environment to meet process-specific needs. The following section describes the benefits of user-definable triggers.

Process Integration

One of the promising new areas in software engineering environment research has to do with providing automated support for the user's software development process. The HP Encapsulator is one of the first products to provide a language for describing local organizational, team, and personal processes. We refer to an EDL program that supports a user's development process as a *process specification*, and we refer to this type of environmental support as *process integration*.

Process Specifications

While not all development processes are amenable to being described with EDL process specifications, most can have at least some aspects automated. In particular, activities and tasks that are essentially event-driven are prime candidates for automated support.

The author of an EDL process specification tells the HP SoftBench environment what to do when specific events occur. There are two keys to the successful implementation of EDL process specifications. First, the notification message events must be announced so that the proper actions can be triggered by the EDL process specification. All HP SoftBench tools and all properly encapsulated tools issue

these notification messages. Second, when a trigger in an EDL process specification occurs, the resulting action needs to be able to control other tools in the environment. All HP SoftBench tools and all properly encapsulated tools provide this by means of the message-based interface to their functionality.

As an example of an EDL process specification, the following team process could be automated:

- Whenever a team member checks a file into the master source file repository (directly or indirectly through the development manager or a substituted configuration management system supporting the DM versioning command class), with a state having the value Release, cause complexity metrics to be calculated.
- If the complexity metrics for the file are not acceptable as defined by the team, create a metrics report detailing the unacceptable functions, and mail it to the user. Also notify the user of the problem via a Warning notification box.
- If the complexity metrics for that file are acceptable, cause a tape archive to be created.
- When the tape archive has been successfully made, cause a mail message to be sent to the project team announcing a new release.

It is important to realize that each team member may have slightly different versions of the above specification, for very legitimate reasons. For example, a software quality assurance engineer team member might want to take action when a metric is found to be unacceptable.

Linking Events and Actions

Linking HP SoftBench Tools. Every action of every HP SoftBench tool provides the hook needed for that action to act as a trigger for other actions. By default, certain actions are predefined, such as the view synchronization that causes tools to know when the files they are displaying become out-of-date so that the user can be informed. With the HP Encapsulator, the user can define additional triggers for situations where actions need to be automatic. For example:

- When a file is checked out, cause the editor to display it automatically for editing.
- Cause a file to be checked in whenever it has been saved from the editor.
- Cause a build to be initiated whenever a file is checked in.
- Cause the debugger to reload automatically whenever a build is successful.
- Cause the static analyzer to update its data base whenever a build is successful.

Linking HP-UX Tools. The HP Encapsulator can be used to encapsulate and link UNIX tools. For example:

- Encapsulate the tape archiver (tar) and cause tape backups to be made whenever a release is built.
- Encapsulate the job scheduling commands (at or cron) and cause builds to run at night.
- Encapsulate the source analysis program (lint) and cause it to analyze a file automatically when it has been checked into version control or saved from the editor.
- Encapsulate the source file printing programs (pr, lp) and cause listings to be printed whenever a project release is made.
- Encapsulate the performance display programs (prof, gprof) and cause the performance data to be displayed after the program has been executed.
- Encapsulate the symbol searching program (nm) to identify libraries that must be added to the library list when the linker finds symbols it cannot resolve.
- Encapsulate the control flow program (cflow) and cause the output to be displayed whenever a file is checked into version control.

Linking Local Tools. After visiting several large customer installations and presenting the ideas and capabilities of the HP Encapsulator, it became clear that an important source of tools to encapsulate and processes to automate would be local tools developed on site. Examples of some of the more common encapsulations and process specifications are:

- Encapsulate local metrics collection tools and cause them to process the files when they have been checked into version control.
- Depending on the nature of the local project management tools, cause them to do their processing whenever a release is made or when a file is checked in.
- Depending on the precise capabilities and structure of the defect tracking mechanisms, cause the defect resolution component to prompt for its data whenever a new local build is successful or whenever a file has been checked out or in.

Linking Target Machines. When developing for a remote target machine, there is typically a great deal of manual intervention involved in transferring the application to the target computer and building and testing it there. Assuming there is some basic file transfer and remote job entry capability from the host to the target system, then using the remote execution capability of the HP SoftBench environment and the Encapsulator, the cross-development process could be improved as follows:

- Whenever a file is checked into version control on the development system, cause it to be copied automatically to the remote target.
- When a build is requested, optionally cause it to execute on the target computer (typically running make or any local compile mechanisms, such as a batch job).
- Encapsulate the job control facility on the target system and cause builds and tests to be run there, initiated from the local system.
- Encapsulate any test scaffolding on the target system and run tests of the application on the target system under the control of the local development machine.
- Encapsulate any performance measurement facilities available on the remote target and monitor performance behavior remotely.

Linking Events with People. Perhaps the most important form of link is the connection between people—individuals and the team—and important events. The definition of an important event likely varies as the development project progresses. This is why the ability to change this definition frequently and from user to user is an important capability of the HP Encapsulator. Some typical triggers for coordinating teams are:

- Send mail to the project leader when files are checked

out.

- Send mail to the project team when a new version of a common include file is checked in.
- Send mail to the project team when a successful system build completes.
- Send mail to the project team when a system build fails.
- Send mail to the project team when a new release is made.
- Send mail to the project team when weekly static analysis data is available.
- Use encapsulated write(1) or talk(1) to initiate interactive discussion of changes when an include file is checked in.
- Initiate an announcement tool (e.g., news or notes) to inform the team of a new release.

Process Specifications in the Future

The examples given above show the types of process specifications that can be designed to assist with software development tasks and to link tools, computers, and people.

EDL process specifications are among the most interesting applications that exploit the HP SoftBench tool integration architecture. The technology is quite new. More data needs to be gathered on the types of EDL process specifications that users write and how much of their process they wish and are able to automate.

Languages in general are difficult to design. Special-purpose specification languages are often more difficult since they are breaking new ground and trying to express new ideas. EDL is a language used for specifying both tool encapsulations and process specifications. However, its design leans towards expressing those concepts necessary for integrating a tool. In certain cases it feels awkward writing a process specification with EDL. Language design for process specification languages is a current research topic being pursued at several university and industrial research labs and ongoing progress in this area is sure to be seen.

HP Encapsulator Implementation

The Encapsulator description language is a special-purpose specification language. It is implemented by means of a compiler and an interpreter. The compiler is responsible for parsing an EDL input file and generating intermediate code. The interpreter is responsible for the execution of that intermediate code. The HP Encapsulator is the development environment for producing EDL code. The HP SoftBench environment contains a portion of the HP Encapsulator, which is the run-time environment for executing compiled EDL code. This allows HP SoftBench and the HP Encapsulator to be two separate products. Users wishing to develop EDL code can do so with the HP Encapsulator and can deliver the production EDL code in binary form to any HP SoftBench system.

Compiler

The compiler is responsible for generating intermediate code from the user's source file or files. It does this in two passes. The first pass is an invocation of the C preprocessor over the EDL source file. This allows the programmer to make use of C preprocessor constructs such as macros, include files, and conditionally compiled code. The second pass invokes a parser over the preprocessed source code. The parser scans the input source code into tokens, recognizes and stores symbols, and forms *productions*. Productions are groups of tokens that form an EDL statement. When a production is formed, intermediate code can be generated. The scanner and parser were produced from the HP-UX tools lex and yacc, respectively. These tools accept descriptions of tokens and grammars and generate the source code for the scanner and parser components.

Symbols are identifiers such as variable names within the source program, and are stored in the *symbol table*. The symbol table is used to record information about each symbol, such as its type, value, printable name, and function address, and whether it represents a function. Because there are often many symbols in an EDL program, a hashing algorithm is used to make symbol lookup more efficient.

Intermediate code is generated by the compiler and stored in the *statement table*. The statement table has three components: the *tag*, the *head*, and the *tail*. The tag identifies the current instruction or operator. The head and tail refer to the left and right operands of the current instruction, respectively. The following is an example of the code generated by the simple EDL statement, "assign the variable X the value Y plus 10."

```
/* Here are the declarations of the integer variables. These
 * two symbols are stored in the symbol table. No code is generated */
integer X, Y;
/* Here's the assignment statement for which code is generated
 * (below)*/
X = Y + 10;
```

Table I
Intermediate Code Stored in Statement Table

Tag	Head	Tail
(1) Symbol	Symbol Index of X	NULL pointer
(2) Symbol	Symbol Index of Y	NULL pointer
(3) Integer Constant	10	NULL pointer
(4) Plus	Pointer to stmt 2	Pointer to stmt 3
(5) Assignment	Pointer to stmt 1	Pointer to stmt 4

Table II
Identifier References Stored in Symbol Table

Name	Type	Value	Function
Pointer to X	Integer	0	None
Pointer to Y	Integer	0	None

Table I is the statement table representation of the intermediate code generated from the assignment example. The intermediate code organization and symbol table were modeled after interpreters for lambda calculus languages.[1] The variables referenced in this example have their symbol table indexes stored in the symbol table, as shown in Table II. These indexes are returned from the hashing algorithm during parsing and allow fast variable value lookup and assignment during evaluation.

Interpreter

The interpreter is responsible for the execution of intermediate code. The interpreter is referred to as a recursive evaluator because it looks at the tag of a statement in the statement table and then calls itself (recursively) to evaluate both of its operands (the head and the tail). The result of calling the evaluator is a typed value. That value can be used as part of an expression or statement. Here is the pseudocode for the evaluator:

```
/* Pseudocode for the recursive evaluator, called eval() */
procedure eval( statement_pointer)
   /* eval() takes a single argument, a pointer to a new statement */
{
   /* For each operator there is a particular section of
    *evaluator code */
   switch (on tag of statement) {

   case OPERATOR1:
      /* Code specific to handling operator1 ... */
      break;

   case OPERATOR2:
      /* Code specific to handling operator2 ... */
      break;

   /* and so on for all operators */
   ...

   } /* End of operator specific handling code */

   /* Now return the result of the operation */
   return result;
}
```

Each case of the evaluator has code specific to executing the specified operator. If that operator has operands, they will be stored in the head and tail of the current statement. In the example described above, the assignment operator has two operands. The head is the left-hand side of the assignment statement, which is the variable reference in which to store the result of the assignment. The tail is the right-hand side of the assignment statement, which is the expression Y + 10, itself a separate operation in the statement table.

Other Components

Other essential components of the HP Encapsulator implementation include the broadcast message server, event handler, and pattern matcher interfaces, the EDL built-in functions, the compile-time and run-time stacks, and the dump/load facility.

Of these components, the dump/load facility has the most significant product implications. This facility allows the developer to compile an EDL program into a binary format. This has two effects. It makes the subsequent loading of the EDL code much faster and it allows the HP Encapsulator to have a run-only version. This run-only version is bundled into the HP SoftBench product. Thus, an EDL developer can use the Encapsulator to implement an encapsulation program and can deliver that encapsulation to any HP SoftBench installation. The run-only version of the HP Encapsulator is implemented by removing the code modules that handle source code parsing, adding an intermediate code relocation module, and restoring code from a file into tables in memory that the HP Encapsulator can interpret. The term relocation refers to the task of relocating an address in the code file into an address in memory.

Acknowledgments

I would especially like to thank Martin Cagan for his continued enthusiasm for an often controversial piece of software. It was his early use, inspiration, and guidance that helped create the HP Encapsulator. I would also like to thank the development team—Elizabeth Carpenter, Hillary Davidson, Gary Fritz, Nancy Kirkwood, and Lisa Rogers—for helping make the HP Encapsulator a product. Special thanks goes to the HP SoftBench mail team—Bob Desinger and Nick Baer—for implementing the most advanced encapsulation we have to date and for contributing to the success of the HP SoftBench product. Finally, thanks to Bill Campbell for teaching me the beauty of interpreters and to my wife, Cathy, for never tiring of the term "encapsulation."

Reference

1. A. Church, "The Calculi of Lambda-Conversion," *Annals of Mathematical Studies*, Vol. 6, Princeton University Press, 1941.

Introduction to Particle Beam LC/MS

Particle beam liquid chromatography/mass spectrometry (LC/MS) yields classical, library-searchable electron impact spectra for compounds that are too thermally labile or nonvolatile to be analyzed by gas chromatography/mass spectrometry (GC/MS).

by James A. Apffel, Jr. and Robert G. Nordman

A MASS SPECTROMETER (MS) is an analytical instrument that is used to measure the molecular weights and chemical structures of molecules introduced into it. Samples are first ionized, and then the charged fragments are separated and analyzed according to their mass-to-charge ratios. HP makes quadrupole mass spectrometers, which accomplish this separation using a quadrupolar electrostatic field.

The ionization method determines the nature of the fragmentation of the molecules. So-called soft ionization causes less fragmentation and hard ionization causes more. Electron impact (EI) ionization, a hard ionization technique, has evolved as the most useful technique, especially for getting information on the structure of the molecule. When the electron energy is controlled, the resultant fragmentation is extensive, thus elucidating the structure of the molecule. The process is also quite reproducible. The mass spectra that result can be compared to spectra stored in a library and thus used to identify the compound being analyzed.

Chemical ionization (CI) is a softer ionization technique than EI. It is often used when EI fragments the molecule to the point where little or no unfragmented molecules are left and thus the analysis will not yield the molecular weight of the compound. CI spectra do not contain sufficient fragmentation information to be library-searchable.

In addition to identification, mass spectrometers are also used for quantitation down to very small concentrations—parts per million and sometimes parts per billion.

Many samples of interest are not single compounds but are mixtures of compounds. Examples are metabolytes in a biological sample or the seepage from a toxic waste dump. Before the mass spectrometer can do its job, the sample must be separated into individual compounds. This separation is accomplished using a gas chromatograph (GC) or a liquid chromatograph (LC). Both types of instruments are produced by HP. In a gas chromatograph the sample is first vaporized and then separated into its components as they are carried by a gas stream. This gas stream can then be introduced directly into the mass spectrometer and each compound in the mixture analyzed as it enters. This combined technique is known as GC/MS.

Many compounds are not volatile enough to be readily changed into the gaseous state without decomposing. With mixtures containing these relatively involatile and thermally labile compounds an LC is usually the separation instrument. In an LC the compound mixture is dissolved in a suitable solvent such as water or methanol and then separated into the individual compounds.

A problem is that this liquid stream cannot be introduced at normal LC flow rates directly into the mass spectrometer without overloading the vacuum system. Even if this were not a problem, the liquid would have to be evaporated and the solvent pumped away before the ionization would be possible. Various techniques have been tried over the years to solve this dilemma. With the moving belt interface,[1] the liquid is deposited on a moving belt, which carries it through a desolvation stage and finally into the mass spectrometer. With the direct liquid introduction interface,[2] a portion of the LC flow is nebulized directly into the mass spectrometer. While the moving belt interface can be used in a variety of ionization modes which generate a range of analytical information, mechanical and thermal limitations make operation difficult and problematic for compounds of limited volatility or stability. Although direct liquid introduction shows improved capability with respect to thermally labile compounds, the technique generates only chemical ionization spectra with the reagent gas limited to HPLC (high-performance liquid chromatography) mobile phase components. HP has made a direct liquid introduction product in the past.

None of these earlier techniques has earned liquid chro-

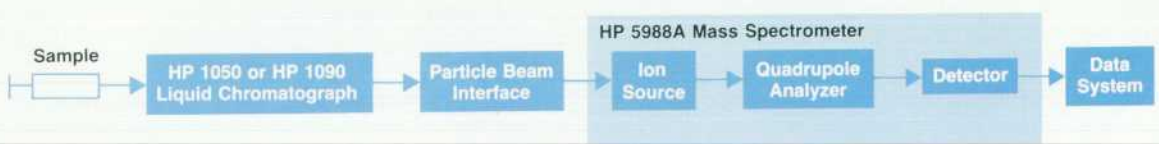


Fig. 1. *For particle beam liquid chromatography/mass spectrometry (LC/MS), a special interface is mounted between the LC and MS systems. The data system can be either an HP 59970C ChemStation for single-instrument operation or an HP 1000 computer system for multiinstrument, multitasking, multiuser operation.*

matography/mass spectrometry (LC/MS) the relatively broad and routine use now enjoyed by GC/MS. More recently, thermospray LC/MS[3] has been introduced and is gaining in use. This interface requires the use of an additional chemical in the liquid stream which, when the liquid is nebulized in a thermal nebulizer, causes charged droplets to be formed which eventually become ions. A portion of this stream is then sampled by the mass spectrometer. HP now offers thermospray LC/MS as an option on the HP 5988A Mass Spectrometer. This technique significantly improves both ease of use and the ability to analyze thermally unstable or nonvolatile compounds, but spectral information is still limited to CI spectra. Furthermore, thermospray also suffers from problems with quantitative performance and analytical predictability. The combination of thermospray LC/MS and tandem mass spectrometry, or MS/MS, offers improvements in the generation of structural information, but MS/MS is costly and the spectra lack sufficient reproducibility to be used in computer-based spectral libraries.

Particle Beam LC/MS

Particle beam LC/MS is a new approach to interfacing high-performance liquid chromatography and mass spectrometry. It yields classical, library-searchable electron impact (EI) spectra for compounds that are too thermally labile or nonvolatile to be analyzed by GC/MS. Particle beam LC/MS can also be used in chemical ionization (CI) mode with free choice of reagent gases. This versatility, in combination with ease of use and good quantitative performance, means that particle beam LC/MS is a viable alternative, or in some cases a complement, to existing LC/MS interfaces. Although particle beam LC/MS is a relatively new technique, it has great potential because it satisfies the two major trends in LC/MS development described above: it is applicable to a wide range of compounds with low volatility or thermal stability and it generates high-information-content EI spectra.

Hewlett-Packard's system for particle beam LC/MS is shown in Fig. 1. It uses existing HP LC, MS, and data systems and a special particle beam interface mounted between the LC and the MS.

The HP particle beam interface is based on the initial development of MAGIC LC/MS (Monodispersed Aerosol Generation Interface Combining LC and MS) by Browner and others.[4] Significant design improvements have been made in performance, ease of use, and robustness.

Principle of Operation

A schematic of the particle beam LC/MS system is shown in Fig. 2. A pneumatic nebulizer generates an aerosol from the HPLC effluent. As the aerosol passes through a desolvation chamber, the volatile components (such as HPLC mobile phase) are vaporized, leaving less volatile components (such as analytes) as submicrometer particles. This mixture of vapor and particles enters a two-stage momen-

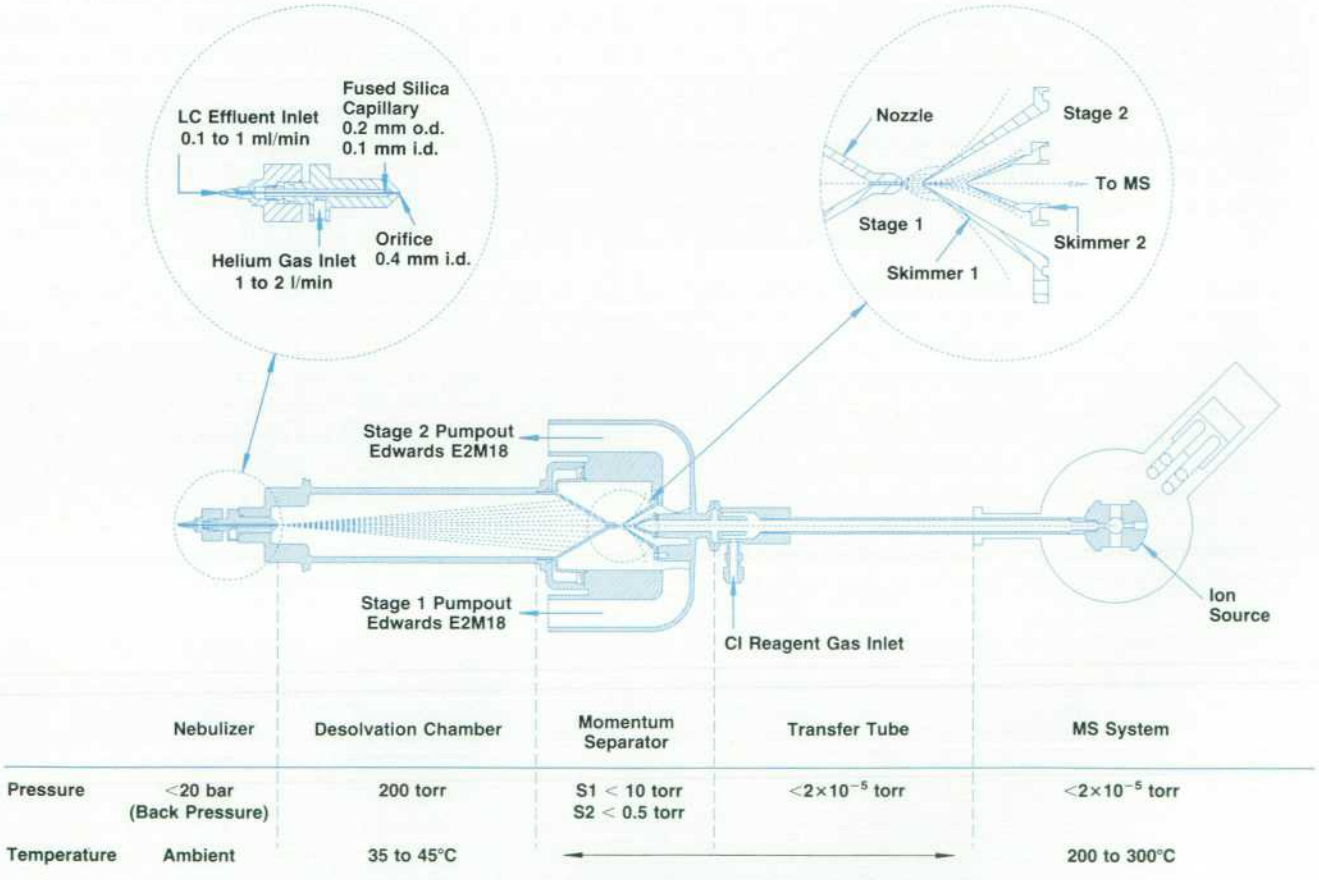


Fig. 2. *Diagram of the HP particle beam interface for LC/MS.*

tum separator in which the relatively low-momentum vapor molecules are pumped away while the higher-momentum particles continue into the source of the mass spectrometer, where they are vaporized, ionized, and finally mass analyzed.

The nebulizer, which is pictured in Fig. 3, consists of a simple coaxial pneumatic nebulizer. The HPLC effluent passes through the fused silica capillary running down the center of the nebulizer while the helium nebulization gas flows around the capillary and forms the aerosol at the exit. The HPLC effluent can be any commonly used HPLC mobile phase, such as water, methanol, acetonitrile, hexane, or chloroform. The single restriction on the mobile phase is that if a mobile phase buffer is necessary, the buffer must be volatile. HPLC flow rates between 0.1 and 1.0 ml/min are acceptable. Typical helium flows are between 1 and 2 l/min. The nebulizer has a replaceable 2-μm filter to prevent clogging of the fused silica capillary. However, since the capillary internal diameter is 100 μm, the in-line filter is merely a precaution.

The desolvation chamber is a hollow cylindrical section between the nebulizer and the momentum separator. The purpose of the desolvation chamber is to allow sufficient time and travel space for effective thermal transfer—via the carrier gas—from the desolvation chamber walls to the aerosol droplets to evaporate the solvent. The desolvation chamber is maintained at approximately 200 torr and 45°C. Although earlier attempts in similar interfaces used lower pressures at this stage, the relatively high pressure is necessary for the helium gas to maintain thermal transfer between the desolvation chamber wall and the solvent droplets. The desolvation chamber is thermostatically controlled to replace the heat that is absorbed as the solvent droplets evaporate.

The momentum separator consists of three main parts: a nozzle and two skimmers. The nozzle funnels the flow of the vapor particle mixture, creating a supersonic jet expansion (and consequently the particle beam). In a supersonic jet expansion, the fluid flow exits from the nozzle at supersonic velocity. At a specific distance from the nozzle orifice, a region known as the mach disk exists, where the flow makes a transition from supersonic to subsonic velocity. The first skimmer orifice is placed just inside the mach disk so that the central particle beam is efficiently transmitted while allowing the vapor molecules to be pumped away. Similarly, the second skimmer removes more of the vapor while transferring a high proportion of the analyte particles. The ultimate result of this two-stage process is a pressure reduction from 200 torr in the desolvation chamber to approximately 5 torr in the space between the nozzle and the first skimmer, then to 0.2 torr in the space between the two skimmers, and finally to 2×10^{-5} torr in the mass spectrometer source manifold.

After exiting the momentum separator, the particle beam passes through a transfer tube into the mass spectrometer ion source. The particles strike the inner walls of the source body (which are held at approximately 250°C) and are vaporized. The vapor phase molecules thus generated can be ionized by either electron impact or chemical ionization.

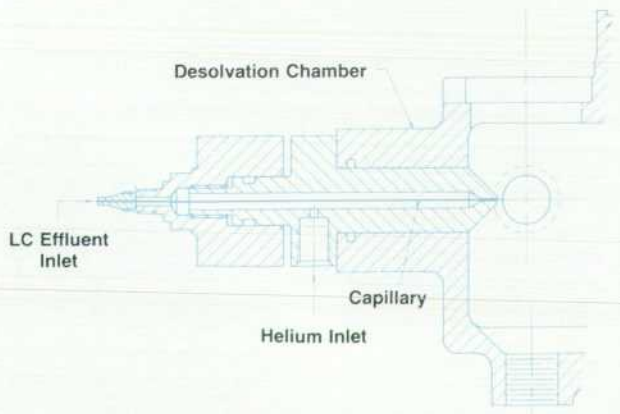


Fig. 3. *Nebulizer detail.*

Performance Optimization

One of the most important features of the particle beam LC/MS system is its ease of use. In most cases, optimization of performance can be accomplished by a single adjustment, the nebulizer. It is also possible to adjust several other operating parameters, such as desolvation chamber temperature, helium flow rate, and source temperature, but default values for these parameters usually work well. All of these parameters are mobile phase dependent except for the MS source temperature, which is compound dependent.

A knob on the back of the nebulizer can be turned to adjust the axial position of the fused silica capillary in the nebulizer body. Since the optimal position is mobile phase dependent, the position must be adjusted when running different mobile phases. For gradient elution, a compromise position is chosen.

Fig. 4 shows the signal intensity for 10-ng injections of caffeine as a function of nebulizer position and mobile phase. Although the exact shape of this surface will change for different fused silica capillary tips, the two-maxima shape is typical, the minimum being at a point where the fused silica capillary tip is almost flush with the nebulizer orifice. Note that when higher proportions of water are used in the mobile phase, there is a loss of signal. This

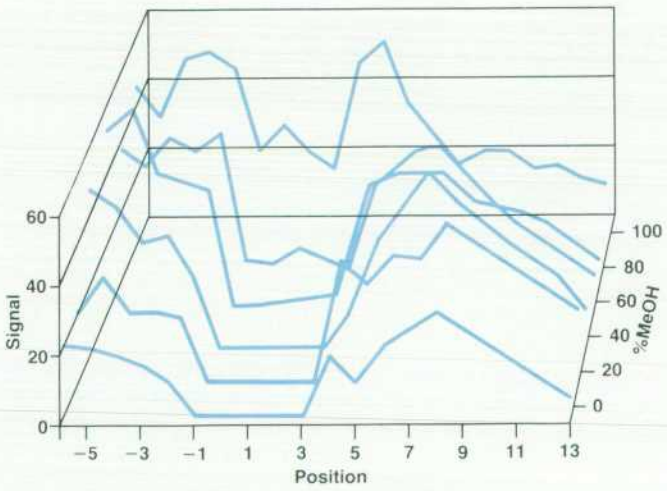


Fig. 4. *Signal intensity for 10-ng injections of caffeine as a function of nebulizer position and mobile phase.*

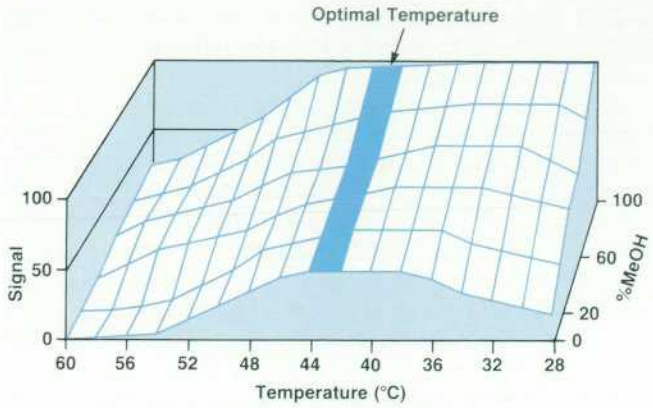


Fig. 5. *Effects on sensitivity of desolvation chamber temperature and mobile phase composition.*

trend is seen with other reversed-phase solvents as well, such as water/acetonitrile and water/THF mixtures. If the nebulizer is adjusted properly, the loss of signal is approximately 50%. In the case of normal-phase solvents, such as hexane and chloroform, this signal loss is not seen.

Although the desolvation chamber temperature can have a significant effect on sensitivity, it is generally unnecessary to adjust it, since a temperature of 45°C is near optimal for most mobile phases. The effects on sensitivity of desolvation chamber temperature and mobile phase composition are shown in Fig. 5. Again, higher proportions of water show poorer sensitivity than pure organic solvents. In cases where high proportions of water are used exclusively (isocratic), it is sometimes advantageous to raise the desolvation chamber to 50°C.

The helium nebulization gas flow rate also has an effect on sensitivity for high aqueous mobile phases. In most cases, a flow of 2 l/min (approximately 40 psi inlet pressure) is optimal, but when using high organic mobile phases, it is possible to reduce the flow to 1.5 l/min. The effects on sensitivity of mobile phase characteristics and helium flow rate are shown in Fig. 6.

Operation of the particle beam LC/MS system has been optimized for liquid inlet flows of 0.5 ml/min or less. Although flows as high as 1 ml/min can be used, there is a loss of sensitivity at higher flows as illustrated in Fig. 7. Below approximately 0.5 ml/min, the response plateaus at a maximum level. This trend is true for both high aqueous and high organic solvents. In practice, the need to use a flow rate of approximately 0.5 ml/min or less for maximum performance is not a limitation since this is the ideal flow range for 2-mm i.d. columns. These columns can be used with conventional HPLC equipment as a simple alternative to more conventional 4.6-mm i.d. columns while offering the additional advantages of reduced solvent consumption (approximately fivefold), reduced packing consumption for exotic stationary phases, and increased mass sensitivity when used with concentration dependent detectors such as UV. Columns are commercially available with a wide range of selectivities and efficiencies comparable to 4.6-mm i.d. columns.

Finally, the mass spectrometer source temperature plays an important role in particle beam LC/MS spectral quality. If the source temperature is too high, significant thermal degradation may occur. Actually, this effect is similar to that seen for all MS operation modes. The effect of temperature depends upon the thermal stability of the analyte. Caffeine, for example, shows very little change in either signal intensity or spectral characteristics over a wide temperature range (150 to 350°C), while some corticosteroids show a very strong effect. Typically, signal intensity and thermal degradation increase with increasing source temperature. Thermal degradation is evidenced by a decrease in the ratio of higher-mass ions to lower-mass ions.

Quantitative Performance

Because of its relatively recent introduction, particle beam LC/MS has not been fully characterized. In particular, the quantitative performance has been evaluated in depth for only a small number of compounds. In addition to studies already under way, however, user-generated results will quickly help to clarify this area in the near future. This section summarizes the performance to date.

Response factors for particle beam LC/MS are not totally uniform. As a result, different compounds exhibit a range of detection limits. This is illustrated by the histogram shown in Fig. 8. The data in this plot shows the minimum detectable quantities (MDQs) for 92 pesticides in nanograms. The tests were run on an HP 5988A mass spectrometer in flow injection analysis (plug injections) and calculated as the amount required to generate a signal to peak-to-peak noise ratio (S/N) of 2 using total ion current (TIC) in full-scan (scan width of 50 amu to the molecular weight

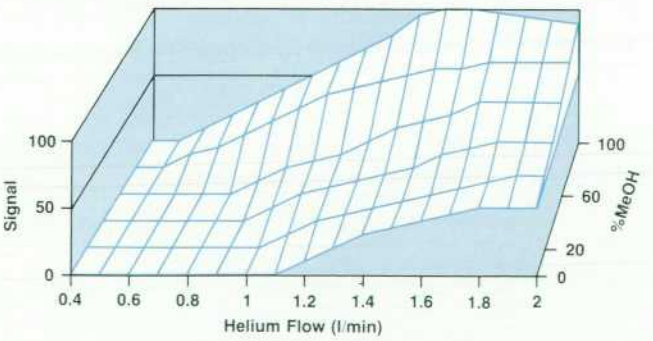


Fig. 6. *Effects on sensitivity of mobile phase composition and helium flow rate.*

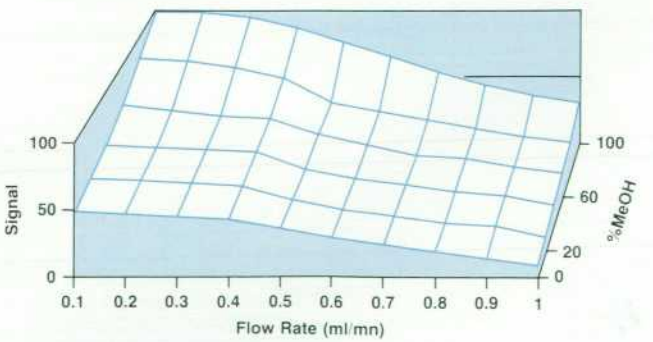


Fig. 7. *Effects on sensitivity of mobile phase flow rate and composition.*

+ 50 amu) data acquisition. It can be seen from this data that 50% of the compounds tested show detection limits below 25 ng. As a rule of thumb, 100 ng of most compounds will generate useful mass spectra. There are compounds that perform especially poorly in EI, such as glyphosine (MDQ = 635 ng) and compounds that perform especially well, such as diuron (MDQ = 2 ng). Although tests for the MDQs of these pesticides had not been run on the newer HP 5989A mass spectrometer at the time this paper was written, the results are expected to improve by about a factor of five when using the newer instrument. It should be noted that while the response factors do show some variation, the range is less than that seen for thermospray on similar compounds.

The linearity of particle beam LC/MS is relatively independent of sample characteristics. Although the calibration curve is fit well with a standard linear function (r^2 = 0.9995), there appears to be a reproducible deviation from linearity at lower levels. This can be seen in Fig. 9, which is a calibration curve for caffeine. The data shown is from selected ion monitoring for a mass-to-charge ratio (m/z) of 194. (The molecular weight of caffeine is 194.)

The reproducibility of particle beam LC/MS is excellent. Typically, full-scan EI TIC data shows relative standard deviations (RSD) of 10% or less, while extracted ion currents or selected ion monitoring data shows RSDs of 5% to 7%, depending on the concentration level. This reproducibility is partially the result of noise characteristics that are relatively independent of HPLC flow pulsations, yielding smooth spike-free chromatograms.

Applications

Particle beam LC/MS has potential utility in solving varied analytical problems. From the confirmation of identity and quantitative analysis of environmental contaminants to structure elucidation of pharmaceutical and bioscience related samples, the capability of generating EI or CI spectra from previously unapproachable compounds opens a number of exciting possibilities.

One of the application areas of greatest potential for particle beam LC/MS is the analysis of samples of environmental interest. Particle beam LC/MS solves two critical problems in this area. First, it allows samples to be analyzed that cannot be analyzed by GC/MS. In a recent study of a California site,[5] it was determined that only 10% of the total organic halogen (TOX) could be accounted for using conventional analytical methods. This brings up the frightening question, "What potential health hazards are we exposed to but unable to monitor because of analytical limitations?" Particle beam LC/MS may allow us to expand the range of compounds that we are able to examine. Second, because of the legal and economic ramifications of environmental monitoring, it is critical that the identification of environmental contaminants be unequivocal. In many cases, the CI spectra provided by thermospray or other LC/MS techniques simply lack sufficient information content to make these identifications. The EI spectra provided by particle beam LC/MS, on the other hand, are generally specific enough to be a source of positive identification.

In 1987, approximately 150 compounds were removed from the U.S.A. Environmental Protection agency's Appendix VIII to form Appendix IX, for the most part because of insufficient analytical methodology. Fig. 10 shows the separation of some of those compounds. Note the excellent chromatographic resolution and fidelity. Of the compounds shown, the spectra of all but one (amitrole) were found in the Wiley/NBS spectral library and were, in fact, the first choice identified by probability-based matching (PBM) using a computer-based library search in spite of the fact that none of them is amenable to GC/MS separation. Many of the spectra found in the Wiley/NBS spectral library were generated by direct insertion probe (DIP).*

A similar environmental application, the analysis of triazine pesticides, is shown in Fig. 11. Although some of the triazine pesticides can be separated using GC, HPLC is the separation method of choice because of its simplicity

*Direct insertion probe is a technique whereby a small amount of the material is analyzed by placing it on a heated probe, which is inserted directly into the mass spectrometer ion source. If the material can be vaporized without decomposing, mass spectra can be obtained.

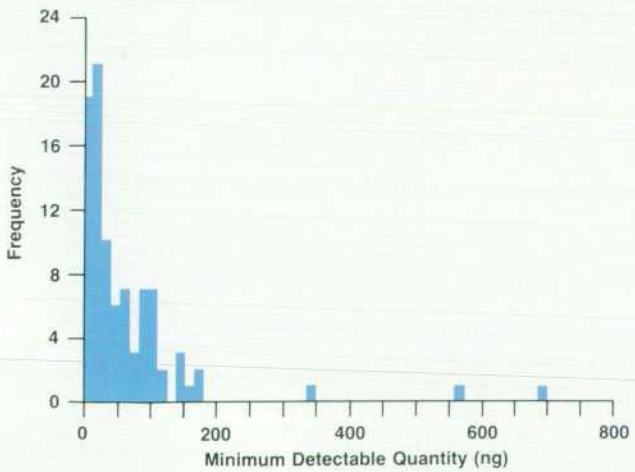


Fig. 8. *Minimum detectable quantities for 92 pesticides for a signal-to-noise ratio of 2 in full-scan data acquisition.*

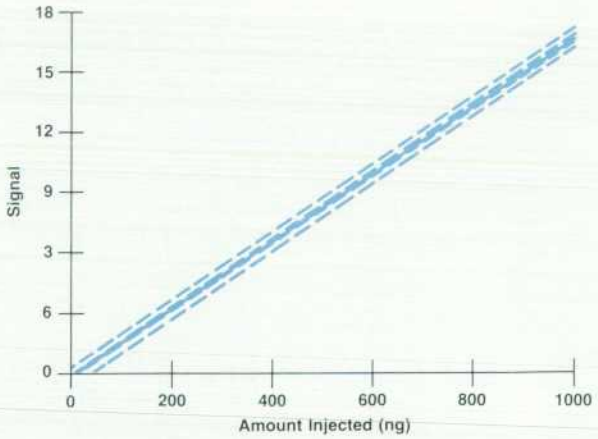


Fig. 9. *Particle beam LC/MS linearity. The solid line is the linear regression line. The inner dashed lines are the 95% confidence limits and the outer dashed lines are the 98% confidence limits.*

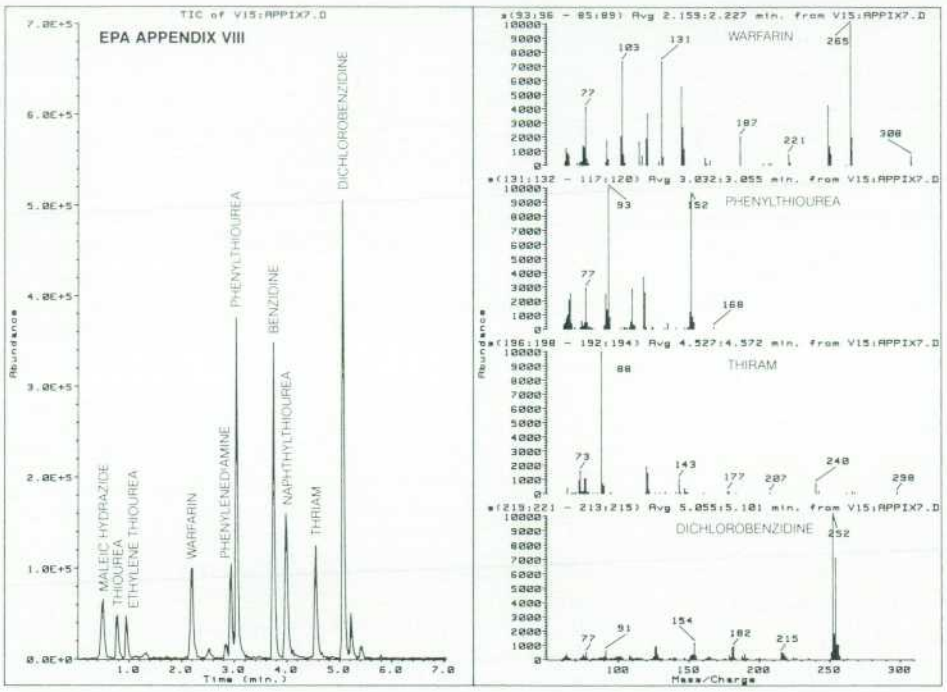


Fig. 10. *Separation of compounds in U.S.A. Environmental Protection Agency Appendix IX by particle beam LC/MS. The left side shows the total ion current (the chromatographic signal) out of the mass spectrometer. The right side shows the spectral data out of the mass spectrometer.*

and the on-line sample handling characteristics of automated HPLC. Using a very simple valving configuration, it is possible to preconcentrate relatively large volumes of environmental samples automatically. The spectra shown on the left in Fig. 11 are extracted from a peak in the total ion chromatogram that appears to be a single homogeneous peak but is, in fact, two nearly coeluting components. Through selected ion profiles, it is possible to separate and quantify these two compounds.

Fig. 12 shows an example of a sample of pharmaceutical interest, the separation of corticosteroids. As mentioned above, corticosteroids show a strong temperature dependence. Although signal intensity can be enhanced by using a higher source temperature, this results in increased thermal degradation. The chromatogram shown represents 100 ng per component at 250°C.

Another example of a life-science application is the separation of aflatoxins B1, B2, G1, and G2, as shown in Fig. 13. Through the use of selected ion monitoring, it is possible to detect and confirm the presence of these compounds at relatively low levels.

Finally, the analysis of monosaccharides and disac-

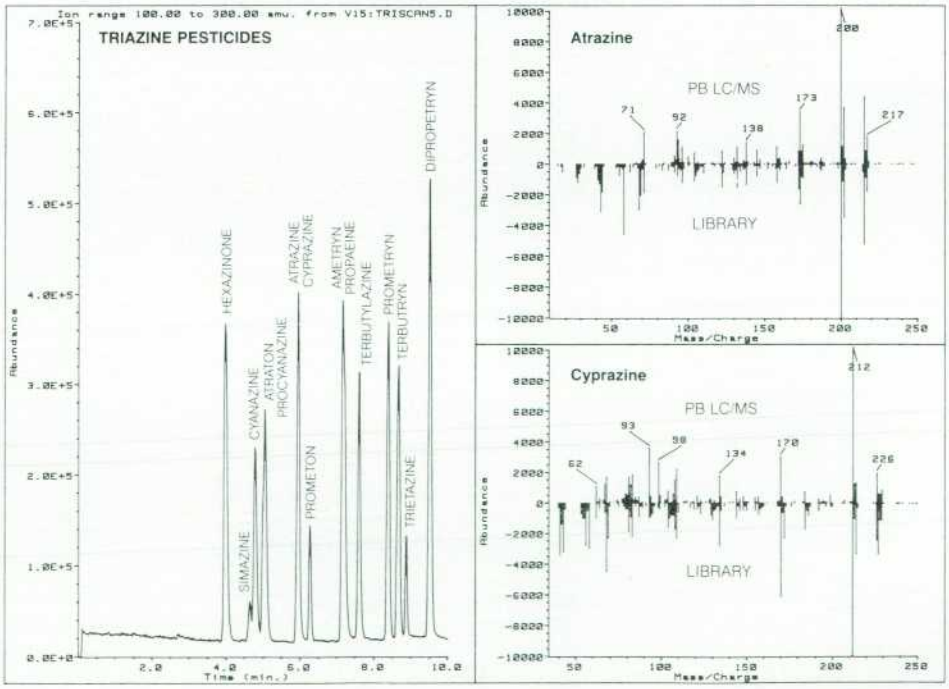


Fig. 11. *Analysis of triazine pesticides by particle beam LC/MS.*

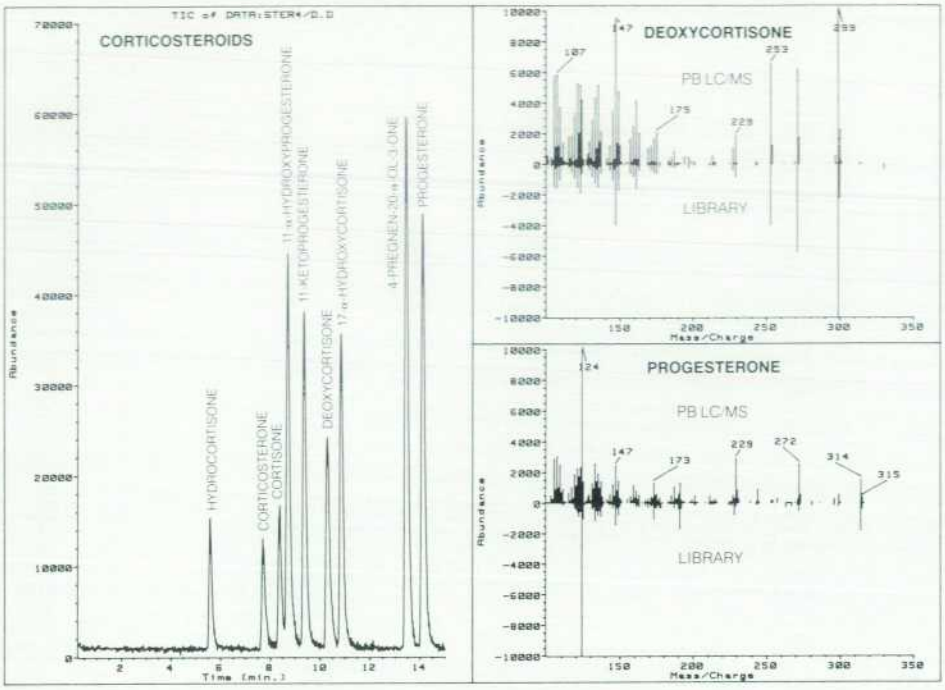


Fig. 12. *Separation of corticosteroids by particle beam LC/MS.*

charides using ammonia chemical ionization particle beam LC/MS is shown in Fig. 14. This application provides a means of performing group-type separations since all pentose monosaccharides, hexose monosaccharides, and homogeneous and mixed disaccharides yield group-specific spectra. Note that the detection limits are well below those obtainable with conventional HPLC and refractive index detection.

Conclusion

Particle beam LC/MS offers an exciting new approach to interfacing LC and MS. Its EI capability, excellent quantitative performance, and ease of use promise to make it the new standard for routine LC/MS operation. It should be noted that, like any technique, particle beam LC/MS does have its limitations, and there are samples that will be better approached using other techniques such as thermospray LC/MS. Furthermore, particle beam LC/MS is a relatively new technique and is not totally mature. There are a number of areas that remain to be understood and characterized. Hewlett-Packard's goal in implementing particle beam LC/MS is to provide a tool for solving analytical prob-

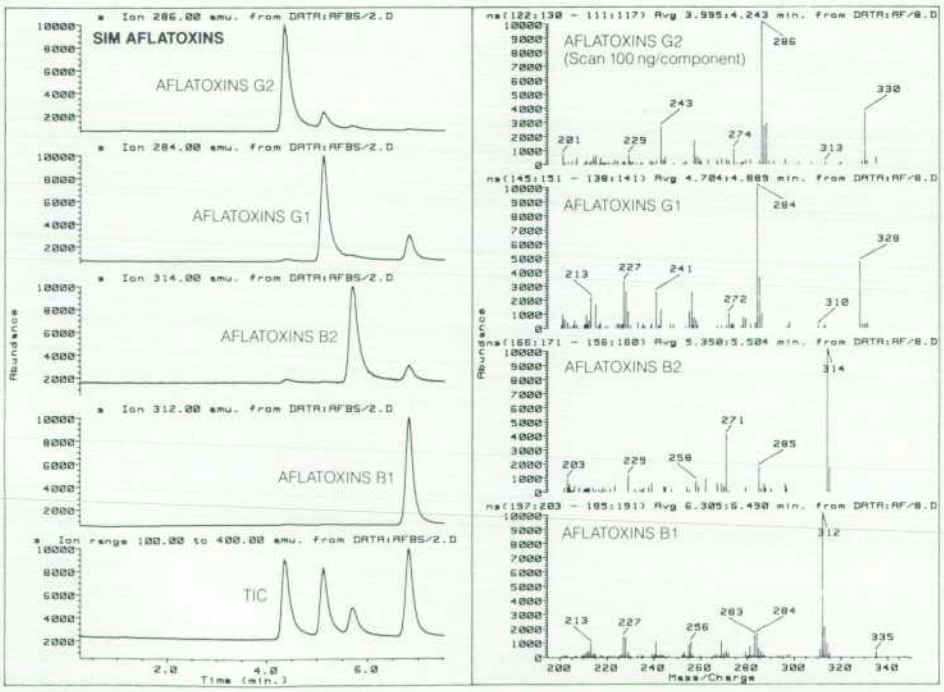


Fig. 13. *Separation of aflatoxins by particle beam LC/MS.*

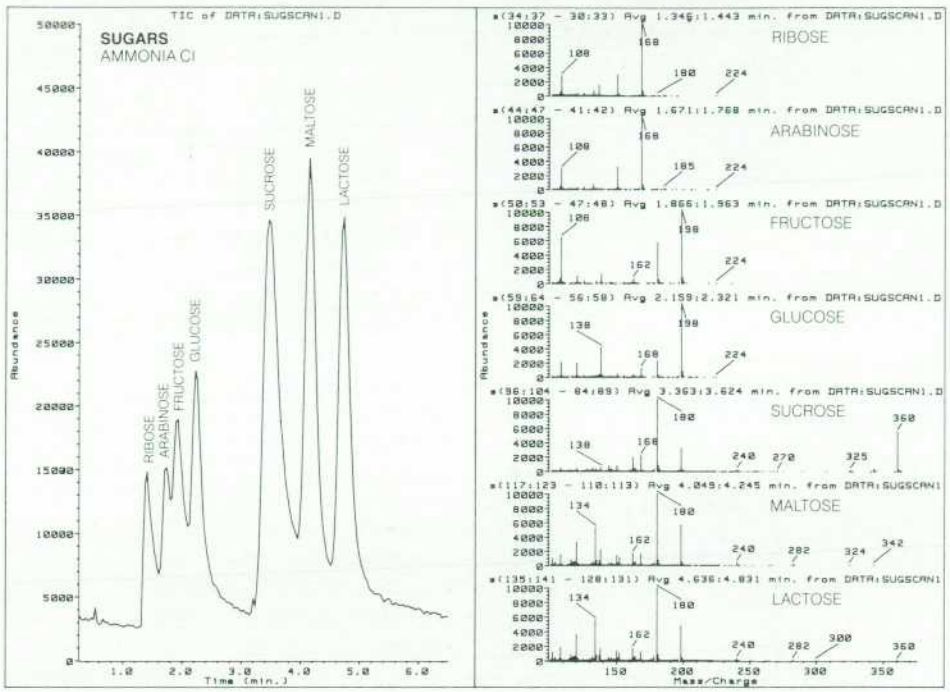


Fig. 14. *Analysis of monosaccharides and disaccharides by ammonia chemical ionization particle beam LC/MS.*

lems. As experience accumulates, strengths and limitations of the technique will become more clearly defined.

Afterword

As this article was going to press, the HP 5989A mass spectrometer was introduced, along with a new particle beam interface. The data in this article was acquired using the older HP 5988A system. Data taken to date on the newer HP 5989A system indicates an improvement in sensitivity on the order of five to one over the HP 5988A. This improvement is compound dependent.

Acknowledgments

The authors wish to thank Rune Brandt and Mirko Martich for their outstanding contributions to the design of the instrument, Laura Cerruti for her diligent running of test samples, Jean-Luc Truche for his gentle and knowledgeable leadership, John Michnowicz who recognized from the start the power of the technique, Dave Gunn and Ernie Strehlow for modelmaking bordering on magic, and all the people at the HP Scientific Instruments Division who worked so hard to make this product a success.

References

1. D.E. Games, "Combined High-Performance Liquid Chromatography and Mass Spectrometry," *Biomedical Mass Spectrometry*, Vol. 8, no. 9, 1981, pp. 454-462.
2. J.B. Crowther, T.R. Covey, D. Sivestre, and J.D. Henion, "Direct Liquid Introduction LC/MS," *LC Magazine*, Vol. 3, no. 3, 1985, pp. 240-254.
3. C.R. Blakely and M.L. Vestal, "Thermospray Interface for Liquid Chromatography/Mass Spectrometry," *Analytical Chemistry*, Vol. 55, 1984, pp. 750-754.
4. R.C. Willoughby and R.F. Browner, "Monodispersed Aerosol Generation Interface for Combining Liquid Chromatography with Mass Spectrometry," *Analytical Chemistry*, Vol. 56, 1984, pp. 2626-2631.
5. M. Brown and R.D. Stephens, "Non-Conventional Pollutants in Ground Water as Characterized by LC/MS," *Proceedings of the EPA Symposium on Waste Testing and Quality Assurance*, 1988, G-28.

Advances in IC Testing: The Membrane Probe Card

Conventional integrated circuit wafer test probes have mechanical and electrical weaknesses, especially for testing high-frequency or high-speed devices and chips that have large numbers of inputs and outputs. Membrane probe technology overcomes most of these limitations.

by Farid Matta

WAFER TEST TERMINOLOGY and practices vary between IC manufacturers. However, most test procedures fall into one of two general categories: parametric testing and die-sort testing.

Parametric testing is intended to check basic device data such as threshold voltages and sheet resistances. It is performed on special patterns, known as test chips, included on the wafers. For a given wafer or wafer lot, passing the parametric test is a necessary but not sufficient condition for yield.

Die-sort testing is performed on all individual chips to sort out the good from the bad. It is normally designed as a sequence of increasingly complex routines so that gross failures are detected early and test time is not wasted on useless chips. Ideally, die sorting culminates in an at-speed test that exercises the chips at a frequency at least as high as the intended application. This way, the manufacturer ascertains that the parts will perform to specifications before more resources are spent on their packaging.

The hardware involved in either of the above processes typically consists of an electronic tester, which executes the test program, a prober, which performs the mechanical manipulation of the wafers, and a probe card, which provides the electromechanical interface between the tester and the device under test (DUT). Usually, the probe card is connected to the tester via a printed circuit board known as the probe-card motherboard or the performance board, which is customized for individual ICs or IC families.

The test software is the multitude of programs that control the electronic tester, commanding it to apply to the inputs of the DUT specific combinations of voltages and currents (known as test vectors), and to measure certain voltages, currents, and time intervals at the outputs of the chip. The measured responses are then compared with predetermined allowed ranges, and accepted or rejected accordingly.

Testing High-Performance Devices

Wafer test requirements vary with a number of factors. For example, certain specifics are dictated by the technology (bipolar, MOS, GaAs), by functionality (logic, memory, linear), by the nature of the signal (digital, analog, mixed-signal), and by other factors. In any of these categories, there are typically a majority of low-to-moderate-performance products and a smaller number of high-performance ones. The definition of what constitutes a high-performance IC may not always be clear. However, at least for wafer testing, two categories of chips pose known challenges: devices with high input/output (I/O) counts and ICs designed to operate at high frequencies or high switching speeds.

Despite the fact that the probe card is the smallest and least expensive component of the test setup, it is usually the main cause of the difficulties experienced in testing high-performance devices. Specifically, when the number of contact pads on the chip is greater than about 150, and/or when the contact pads are spaced particularly closely, it is difficult or impossible to procure conventional probe cards that will work reliably in factory conditions. Also, when the operating frequency or switching speed is high, the probe card's parasitic effects can distort the test conditions as well as the measurement results.

Distortions resulting from probe-card parasitic effects can occur when either of the following conditions is present:

- The wavelength of the highest significant frequency is of the same order as the linear dimensions of the probe. When this is true, at any given instant the voltages and currents at different points of the probe can be substantially different. In this case the probe represents a transmission line section, in which such phenomena as signal reflections can become dominant.
- Parasitic reactances in the probes are sufficiently high to redistribute the circuit voltages and currents. In digital

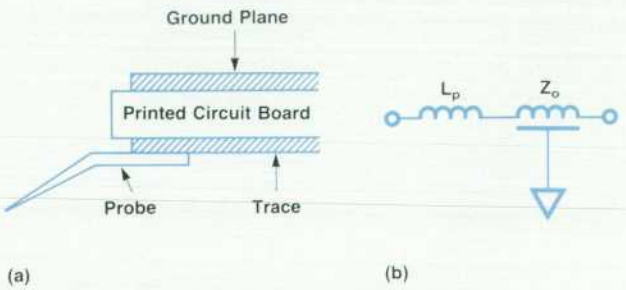


Fig. 1. *Conventional wafer probe. (a) Structure. (b) Equivalent circuit.*

situations the same criterion is formulated differently; namely, one is faced with a high-speed testing situation when the DUT switches faster than the time constant of the circuit containing the parasitic reactance.

Chip makers sometimes circumvent the difficulty of obtaining high-pin-count probe cards by using self-test, so that only a subset of the pads needs to be accessed. However, this approach imposes a penalty in wafer "real estate" and results in incomplete test coverage. No way to circumvent the signal integrity problem has yet been found.

Failure to perform at-speed testing at the wafer level leads to the wasteful packaging and retesting of a certain volume of bad chips that could have been identified and rejected earlier. Depending on the complexity of the part and on the signal frequencies involved, that fraction can be as high as 10% of the total. The financial impact depends on the incremental packaging and testing cost. To illustrate the magnitude of the problem, consider the following conditions, which are not unusual for a high-performance IC:

Annual production volume: 100,000 chips
Fraction of rejects at packaged test: 10%
Cost of packaging per chip: $100
Cost of final test per chip: $2
Total waste per year: $1,020,000.

Hence, the availability of at-speed testing capability at the wafer level is of pressing importance to the IC industry, and the main element of that objective is the development of an IC probe card capable of addressing a large number of inputs and outputs and of maintaining signal integrity for a wide range of frequencies and operating speeds. To develop such a technology, it is necessary to identify and formulate the problems preventing conventional probe cards from delivering the needed prformance.

Conventional Probe Card Technology

A conventional wafer probe card consists of a set of fine styli, or probes, mounted on a carrier substrate, typically a printed circuit board. The probes are arranged so that their tips form a pattern identical to that of the DUT's contact pads. The outward ends of the probes are soldered to traces on the carrier printed circuit board, which extend to a connector that interfaces the probe card with the performance board.

The probe card is normally mounted face down on the prober, which brings the wafer to be tested to a position under the probe card, aligns it so that its contact pads are against the tips of the probes, and raises the wafer until contact is made. In practical conditions the tips of the probes may not be precisely in the same plane, and the probe card may not be exactly coplanar with the surface of the DUT. To compensate for such variations, the prober raises the wafer beyond first contact by a controlled amount, called the overdrive.

The probes on the probe card are usually held at a low angle to the plane of the DUT, so that when they are pushed by the wafer, the tips slide along the surfaces of the pads. This horizontal movement, called scrub, helps remove the oxide films on the surface to ensure good electrical contact, and is an important element in the art of wafer probing.

A number of variations on this basic technology exist, which attempt to improve the mechanical and electrical performance of the probe card. In some advanced versions, the carrier printed circuit board is designed to provide a controlled-impedance environment in which each trace presents a section of transmission line of known characteristic impedance. A schematic illustration of a line on a conventional probe card is shown in Fig. 1a.

Conventional probe cards have a number of inherent limitations. Some are related to their mechanical properties and others to their electrical performance.

Mechanical Limitations. Because the contact element, the probe, is a thin long structural member, it tends to change its spatial location under the repeated stresses of normal

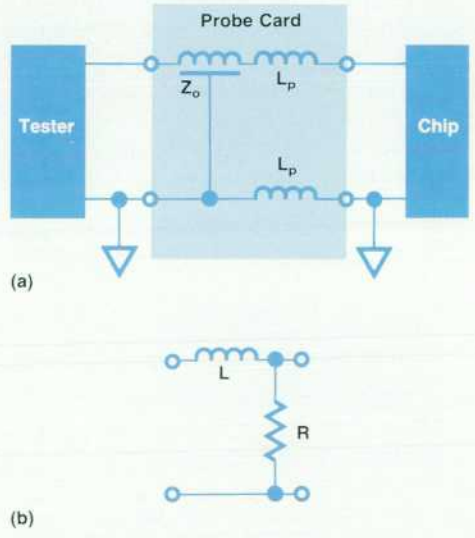


Fig. 2. *The conventional probe as a signal line. (a) Equivalent circuit. (b) Equivalent low-pass filter.*

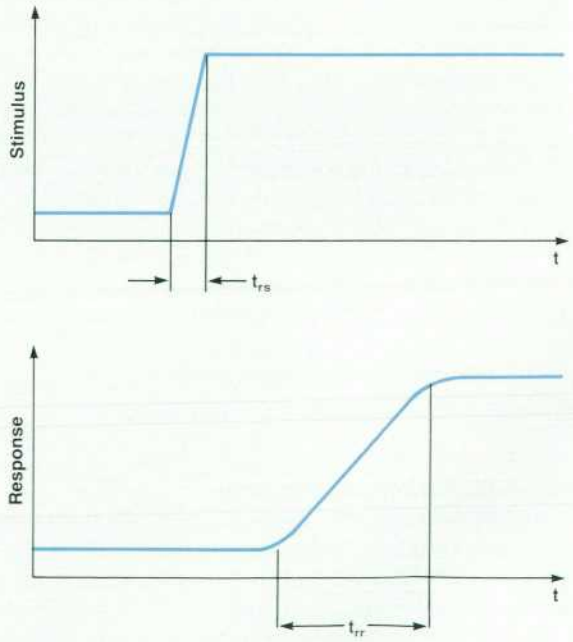


Fig. 3. *The effect of parasitic inductance in the time domain.*

operation. Consequently, the user needs to realign the probes in the horizontal plane and along the vertical axis after some number of touchdowns. This tedious operation translates to a significant increase in the cost of ownership.

At higher probe densities the problem of maintaining registration is further aggravated. High densities require that the individual probes be made even thinner, longer, and closer together. Such conditions are not only more conducive to the loss of registration, but they also make the realignment procedure too sensitive to be performed by the user, and the probe cards need to be returned to the factory for costly maintenance.

Electrical Limitations. The equivalent circuit of a single line in a conventional probe card is shown in Fig. 1b. Here, L_p is the inductance of the probe and Z_0 is the characteristic impedance of the transmission line formed by the trace and the ground plane on the printed circuit board carrier. Typical values of these parameters are: $L_p = 10$ nH and $Z_0 = 50\Omega$. Such a line may be used either to transmit a signal from the tester to a DUT input (or from a DUT output to the tester), or to supply a power or a ground connection to the DUT. We will discuss its behavior in each of these two situations.

Fig. 2a shows the equivalent circuit of a conventional probe card in a simplified input-connection configuration. (An output connection would not be much different in principle.) In the frequency domain, the parasitic inductance L_p of the probe causes the circuit to behave like the low-pass filter shown in Fig. 2b, and determines the bandwidth of that filter. Assuming matched conditions, the upper 3-dB limit of the band is the frequency f at which:

$$(2\pi f)(2L_p) = Z_0.$$

For $Z_0 = 50\Omega$ and $L_p = 10$ nH we find the bandwidth to be about 400 MHz.

The effect of the needle inductance in a high-speed switching situation is better illustrated in the time domain (see Fig. 3). When the tester sends to the DUT a pulse stimulus having a rise time t_{rs}, the signal received at the input of the DUT will have a longer rise time, t_{rr}, because of the circuit's parasitic inductance $2L_p$. The increase in rise time is approximately 2.2 times the time constant $2L_p/Z_0$, or about 1 ns.

Generally, a circuit's bandwidth BW and its effect on the rise time t_r are related by the known expression:[1]

$$BW(3\text{-dB}) = K/t_r,$$

where K is a constant that ranges between 0.35 and 0.45. With this in mind, the bandwidth and the rise time can be used interchangeably.

Waveform deterioration is not the only problem caused by the parasitic inductances of the probes. In Fig. 2a, the chip ground node and the tester ground node are not always at the same electrical potential. This can lead to erroneous testing and/or to unwanted coupling between different signal lines served by a common ground probe. Another problem is that the discontinuity between the probe and the transmission line causes multiple reflections of the signal, which result in a long settling time. Also, the mutual inductance and the coupling capacitance between the long unshielded needles contribute to unacceptable levels of cross talk between the signal lines.

In the case when a line of a conventional probe card is used for power delivery, the high inductance of the probe can cause significant variations in the voltage levels of both the bias line and the associated ground connection. Consider, for example, the circuit of Fig. 4, in which the power line is at a voltage V. If the DUT switches a current I in a time interval dt, a voltage dV will develop across each of the probes, temporarily reducing the voltage across the DUT to V − 2dV. This change in the voltage can be roughly expressed as:

$$dV = L_p I/dt.$$

For example, when eight drivers are simultaneously switching 10 mA each in 1 ns, and the probe inductance is 10 nH, the power supply disturbance 2dV will be in excess of 1.5V.

In addition to the just-described phenomenon, the voltage drop developing across the parasitic inductance of the ground connection again causes the chip's ground potential to deviate temporarily from that of the tester. The difference, known as the ground bounce, is coupled into other signal lines as unwanted and unpredictable noise.

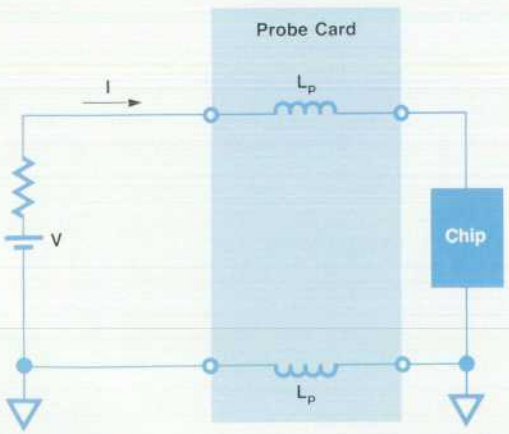


Fig. 4. *A power line in a conventional probe.*

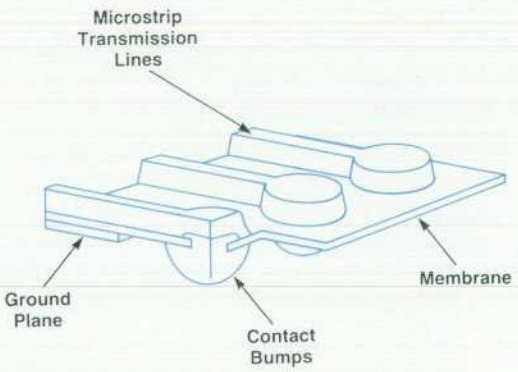


Fig. 5. *The membrane probe concept.*

Requirements for High-Performance Probe Cards

Based on the analysis of the present probe-card technology and its shortcomings on the one hand, and of the current and future needs of the industry on the other, one can formulate the requirements for the more advanced probe card that is needed. Since the shortcomings of existing probes have been identified as pertaining to the areas of contact density and signal fidelity, it is natural to define the incremental requirements in the same terms.

Requirements related to contact density include:

- The technology should allow the creation of a large number of contact points (more than 500).
- The minimum contact pitch, that is, the center-to-center distance between the closest contact points, should be as low as 0.004 inch.
- The contact points should have a fixed alignment in the plane parallel to the wafer under test.
- The contact points should be able to move with respect to each other in the verical direction to accommodate normal variations in wafer topography.

Requirements related to electrical performance include:

- The bandwidth of a line should be at least 10 to 20 times the clock rate of the targeted digital systems (i.e., 2 to 3 GHz).
- A controlled-impedance, reflectionless electrical environment should extend from the tester to within at most 1 mm from the I/O pads of the DUT.
- The uncompensated inductance of the contact must not exceed 0.1 nH.
- The cross talk between adjacent lines should be at least two orders of magnitude less than in a conventional probe card of the same density.

The Membrane Probe Concept

The membrane probe is a proprietary wafer probing technology developed at Hewlett-Packard's Circuit Technology R&D Laboratories as a solution to the high-performance wafer-level test problems described above. The concept of the membrane probe card is depicted in Fig. 5. As shown in the illustration, a thin and flexible dielectric film (a membrane) supports a set of microstrip transmission lines that connect the DUT to the test electronics. Each microstrip transmission line is formed by a conductor trace and a common ground plane positioned on the opposite side of the flexible membrane. The conductor traces and the ground plane are patterned on the membrane using photolithographic techniques.

Given the thickness and the material of the membrane, the width of a signal trace is chosen to obtain the desired characteristic impedance Z_0 of the microstrip transmission line. Typical values of Z_0 in common use are 50 and 75 ohms.

Contact to the DUT is made by an array of microcontacts which are plated up at the ends of the transmission lines through holes in the insulating membrane. The membrane is operated under low tension in a drumhead configuration so as to planarize the contact array. The tension in the membrane is carefully controlled to allow a degree of independent motion of the contact points in the vertical direction, thus accommodating small variations in the heights of the contacts or in the topology of the device under test.

With regard to the need for higher pin counts and contact densities, the membrane probe card technology offers a quantum jump in comparison with the conventional technology. In the membrane probe, the leads and the contact points are created by photolitographic means with inherently high resolution and positioning accuracy. This allows the creation of fine, dense patterns, and makes the manufacturing process, and ultimately the cost, virtually independent of the complexity of the pattern.

Since the contact points are fixed on a common carrier (the membrane), they are fundamentally aligned for life in both the vertical direction and in the plane of the DUT. This eliminates the need for probe realignment, which is an extremely labor-intensive and costly operation in the conventional technology.

In the area of electrical performance, the membrane probe technology presents an equally significant advance. The transmission-line configuration extends all the way to within 0.1 mm of the DUT's I/O pads, thus providing a carefully controlled electrical environment in practically the entire path of the signal. There is very little uncompensated lead inductance to cause waveform degradation or power or ground potential bounce, or to generate signal coupling through a common ground inductance.

Furthermore, the presence of a ground plane so close to the leads concentrates the electric field under the traces, which minimizes the coupling capacitance between them, thus greatly reducing cross talk.

Architecture of the Membrane Probe

The architecture of the membrane probe card is shown

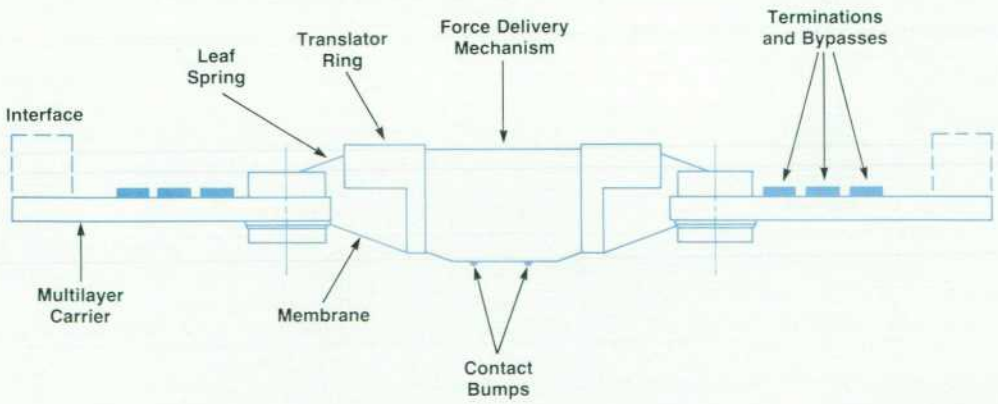


Fig. 6. *Architecture of the membrane probe card.*

in Fig. 6. The membrane, configured as described above, is attached to a printed-circuit-board carrier, which also carries termination resistors, bypass capacitors, or any other necessary components. A force delivery mechanism is mounted on the printed circuit board carrier and is designed to perform three distinct and independent functions:

- Apply a force to the microcontacts sufficient to obtain a low and stable contact resistance.
- Activate a scrubbing motion to ensure the removal of surface insulating layers, including oxides, from the surface of the DUT.
- Provide the mechanical degrees of freedom necessary to ensure continuous conformance of the plane of the microcontacts to the plane of the DUT.

The required contact force is exerted on the membrane by the two leaf springs shown in Fig. 6 through a rigid translator ring attached to the membrane. The primary factor determining the total force needed is the force per contact, so the total force depends on the number of contact pads in the DUT and is set by a proper choice of leaf spring thickness. It was empirically determined that for the selected bump material to make a low-resistance contact with aluminum (the most widely used pad material in ICs), the force per bump must be at least 10 grams. Taking this into account along with other considerations, the range was determined to be 15 ± 5 grams.

As mentioned earlier, the scrubbing motion of the contact bumps with respect to the probed surface is of critical importance to obtaining a low, stable, and repeatable contact resistance, especially with non-noble materials. For aluminum, the minimal acceptable scrub action was found to be about 10 micrometers. The upper limit is defined by the size of the contact pads on the DUT, and for most practical cases is about 25 μm. In the membrane probe, the scrub action is built into the force delivery mechanism rather than implemented using external actuators.

The compliance of the probe's contact points to the surface of the DUT is perhaps the most critical prerequisite for successful probing. The importance of compliance stems from the fact that in practical conditions the position of the probe card in the prober can never be adjusted accurately enough to make it perfectly coplanar with the wafer's surface. Even if such an adjustment could be made, every wafer has a different bow and taper, and therefore presents a new surface to the probe. It is impractical to readjust the probe for every wafer.

In the conventional wire probe, compliance is achieved "naturally" because the long needles are flexible and there is virtually no mechanical linkage between individual contact points. In the membrane probe, special provisions need to be made to achieve adequate surface compliance. These provisions must accommodate two modes of deviation from coplanarity: a short-range mode and a long-range mode. The short-range mode consists mainly of variations in the bump height and in the topography of the DUT. The long-range mode is a general tilt of the probe's surface with respect to the plane of the DUT. The two modes are fundamentally independent of each other. The force delivery mechanism of the membrane probe has been designed to provide both short-range and long-range compliance with the DUT surface at every touchdown through independently acting micromechanical means.

At the outer edge of the printed circuit board carrier, connectors are provided to interface with the tester's performance board. The size and shape of the probe's printed circuit board carrier and the type of interface connector are different for different probe/tester combinations, but the core remains the same.

Performance of the Membrane Probe Card

The performance of the membrane probe card has been fully characterized both parametrically and in actual use at alpha sites. A parametric evaluation is one that is carried out in the laboratory under controlled conditions, and is based on special test structures designed for the purpose. The products of a parametric evaluation are the basic parameters of the tested probe card. They reflect its intrinsic qualities, and are independent of the other components involved (tester, DUT, etc.). Examples of parametric measurements are characteristic impedance and bandwidth.

An alpha-site evaluation is one in which the probe card is used to test a real IC in a factory atmosphere in conjunction with all the other components of the test setup. Its purpose is to verify the probe card's performance and uncover any issues that may occur in realistic conditions.

Parametric Evaluation. The parametric evaluation of the membrane probe card covered both its dc and its ac properties. The dc parameters measured were the contact resistance and the current carrying capacity. The ac parameters

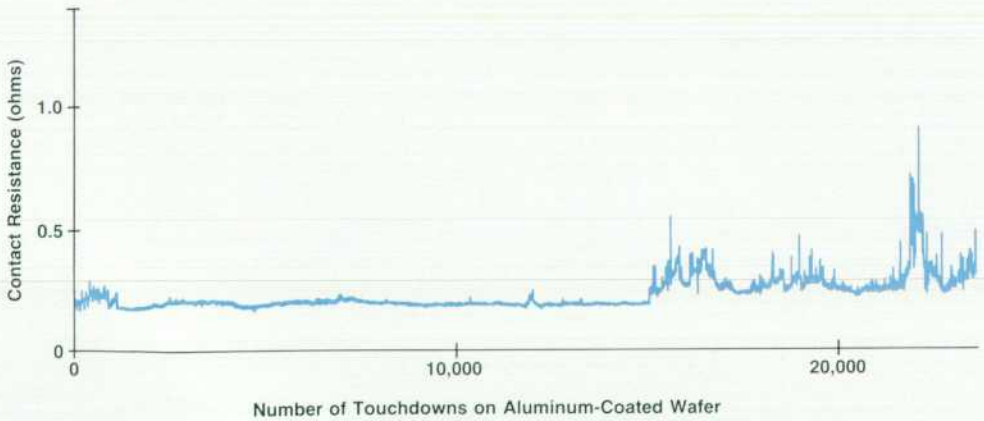


Fig. 7. *Contact resistance as a function of the number of touchdowns.*

were the characteristic impedance, the bandwidth, the pulse rise time, and the cross talk between lines.

Contact resistance was evaluated by measuring the total resistance between one of the probe card's lines and an aluminized silicon wafer, then subtracting the known trace resistance. Aluminum (as a 1-μm-thick film) was chosen for two reasons. First, it is the most widely used IC metallization, and second, it is the most difficult metal to make contact with because of its propensity for oxide formation.

A plot of the contact resistance to aluminum as a function of the number of touchdowns is shown in Fig. 7. As can be seen, the contact resistance remains low and fairly constant for over 20,000 touchdowns. It then starts to deteriorate as oxide debris accumulates on the microcontacts. After a simple cleaning, however, the low resistance is restored and the behavior is repeated. An important utility of this plot is that it defines the cleaning frequency required to attain stable performance. Membrane probes were found to deliver low and stable contact resistance for up to 1 million touchdowns when cleaned once every 20,000 cycles.

Current carrying capability of a trace was defined as the dc current that can be continuously passed through that trace without an observable change in its appearance. For a standard signal line the current carrying capability was found to be about 300 mA and was relatively independent of trace thickness in the range of 0.5 to 1 oz/ft^2. Coincidentally, roughly the same current carrying capability was measured for the contact between the bump and the wafer's aluminum metallization. However, in this case it was defined as the maximum dc current at which no hysteresis is observed in the VI characteristic curve.

Characteristic impedance measurements were made using time-domain reflectometry (TDR).[2] The TDR profile of a signal line is shown in Fig. 8. Parts of the plot corresponding to various sections of the circuit are marked: a 50Ω controlled-impedance connector, the trace on the probe card's printed circuit board carrier, and the microstrip line on the membrane. According to this TDR signature, taken with a 50-ps system, the characteristic impedance of the line tested is within ±10% of the target value of 50Ω.

Transmission response to a 50-ps step function excitation was recorded for the same 50Ω signal line and is shown in Fig. 9. The 10-to-90% rise time of the response was determined to be about 180 ps, a considerable fraction of which occurs in the 80-to-90% region.

Bandwidth measurements were made by plotting the frequency response curve of one of the membrane probe card's lines. This curve, also known as a Bode plot, is shown in Fig. 10. The 3-dB bandwidth of the tested line was found to be in the range of 2.5 to 3.0 GHz.

Cross talk between the probe card's lines was measured as a function of frequency, and the results are plotted in Fig. 11. The measurements were made on a card containing 272 traces, which gives an idea of the spatial density of the lines in this case. One of the two curves shown is for adjacent lines, while the other curve is for alternate lines. At 100 MHz the cross talk is −45 dB for adjacent lines and −78 dB for alternate lines. In comparison, a conventional wire probe card showed adjacent-line cross talk of −38 dB at a density less than one third that of the membrane probe card of Fig. 11.

Alpha-Site Testing. A number of membrane probe cards were fabricated to test specific chips of various technologies, I/O counts, pad metallurgies, and functionality. The testing of these chips was conducted in HP's wafer fabrication plants. Below is a brief description of the tests and their results.

1. Bipolar ECL Flash Analog-to-Digital Converter. This 1.5-watt, mixed-signal chip with 50 I/O pads and gold metallization was tested at a 10-MHz sampling rate using a tungsten wire probe card. It was then retested with the membrane probe card, and the test results were compared. The superior accuracy of the membrane probe card is demonstrated by Fig. 12. In this plot, the voltage at test point 1, denoted here as V_{tp1}, is recorded for a sample of 120 dice. This voltage is offset from the chip ground by the drop across a forward-biased junction, and its measurement is intended to detect variations in the chip's ground potential. The results show a progressive deterioration in the stability of the measurements made with the wire probe card, while the membrane probe card measurements are consistent over the entire test.

The accuracy of the membrane probe data was confirmed

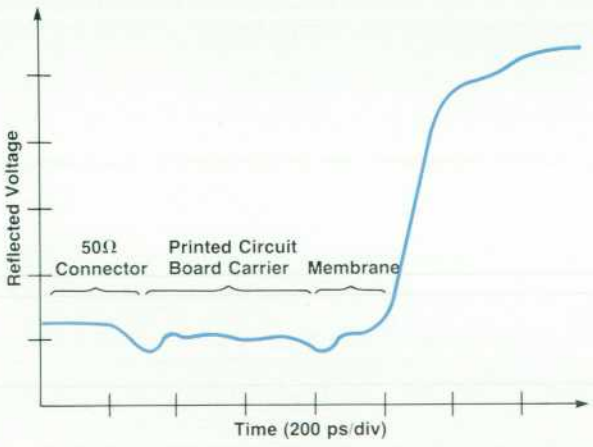


Fig. 8. *Time-domain reflectometry profile and characteristic impedance.*

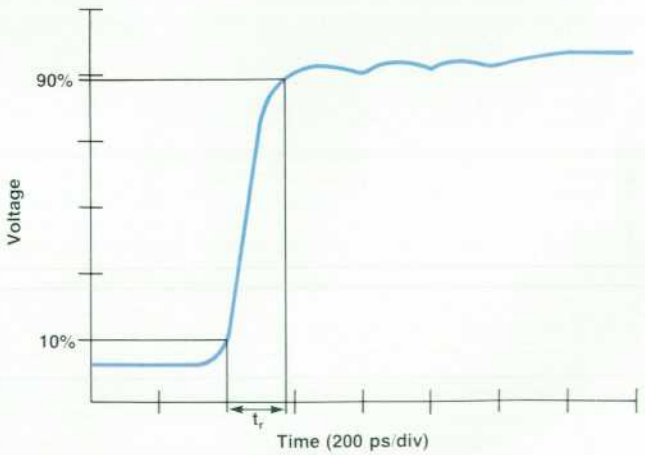


Fig. 9. *Transmission response and rise time.*

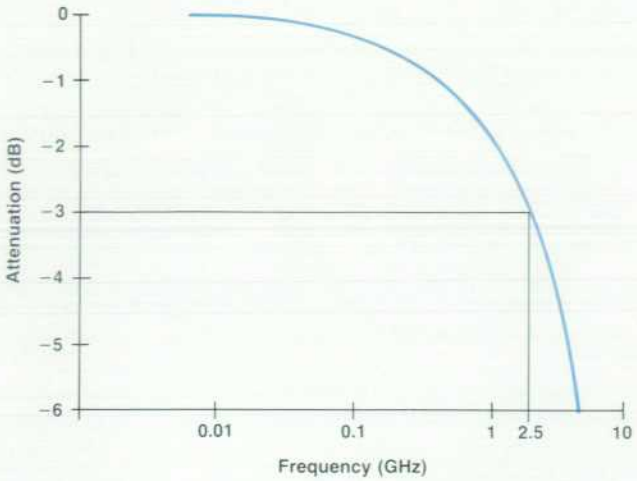


Fig. 10. *Bode plot and the 3-dB bandwidth.*

by packaging a number of dice and repeating the measurements on the packaged parts. The spread of the membrane probe card measurements translates to a contact resistance stability of better than 5 mΩ, which is 40 times better than that of the wire probe card.

The membrane probe's ability to make repeated touchdowns on the same die without appreciable damage to the pads is illustrated by Fig. 13. For 200 touchdowns the total variation is less than 1.2 mV, which corresponds to a contact resistance variation of 6 mΩ. By contrast, using a wire probe for more than two touchdowns on the same chip usually causes enough damage to render the pads unbondable. This capability is of significant value to chip manufacturers, who sometimes lose up to 5% of their chips to pad and passivation damage.

2. Bipolar ECL Digital-to-Analog Converter. This represented an at-speed analog test of an industry standard product using the HP 9840 VHF linear tester and the membrane probe card shown in Fig. 14. The analog output response of the device at a 10-MHz clock rate is presented in Fig. 15 for both the membrane probe card and a conventional counterpart. The rise and fall times are about the same at 1.3 and 1.0 ns respectively (they are essentially determined by the switching characteristics of the device rather than by the probe's performance). However, the settling time measured by the membrane probe card is only 9 ns compared with 32 ns for the conventional probe card. This significant difference is a result of the improved impedance matching in the membrane probe card.

3. NMOS CPU. A membrane probe card was used to test a 32-bit microprocessor at 85°C (Fig. 16). The 8.4-mm-square, 15-watt chip has 272 peripheral aluminum pads arranged in two staggered rows at an effective pitch of 110 μm. This type of device has very large current transients, and the stability of power buses is of special concern.

The device was normally tested at speed only after packaging. With the membrane probe card, it became possible to run the at-speed package test on the wafer for the first time. Careful membrane layout and close positioning of over 130 bypass and termination components helped obtain the desired performance.

4. CMOS ASIC. In this alpha-site test, a membrane probe card was used to test high-pin-count ASICs (application-specific ICs) before TAB (tape automated bonding) inner-lead bonding. The probe card addressed 180 gold mesa bumps, each 75 μm square, placed on a 150-μm pitch.

One part of the test was designed to evaluate the membrane probe card's fitness for the specific purpose of probing bumped wafers. In that part, 600 passes were made on one wafer, which had about 100 dice. The average contact resistance variation was found to be less than 13 milliohms,

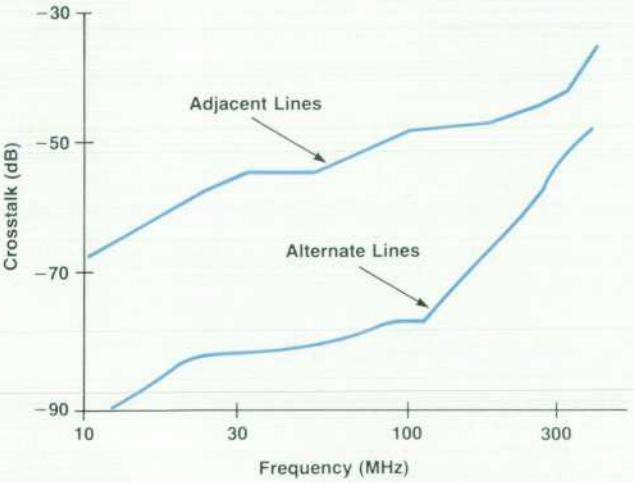


Fig. 11. *Cross talk.*

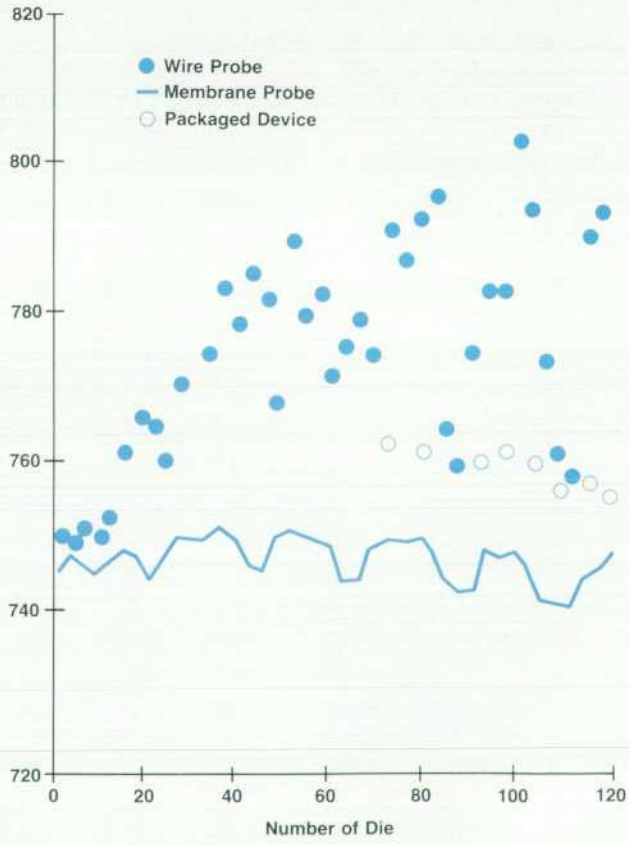


Fig. 12. *Measurement accuracy.*

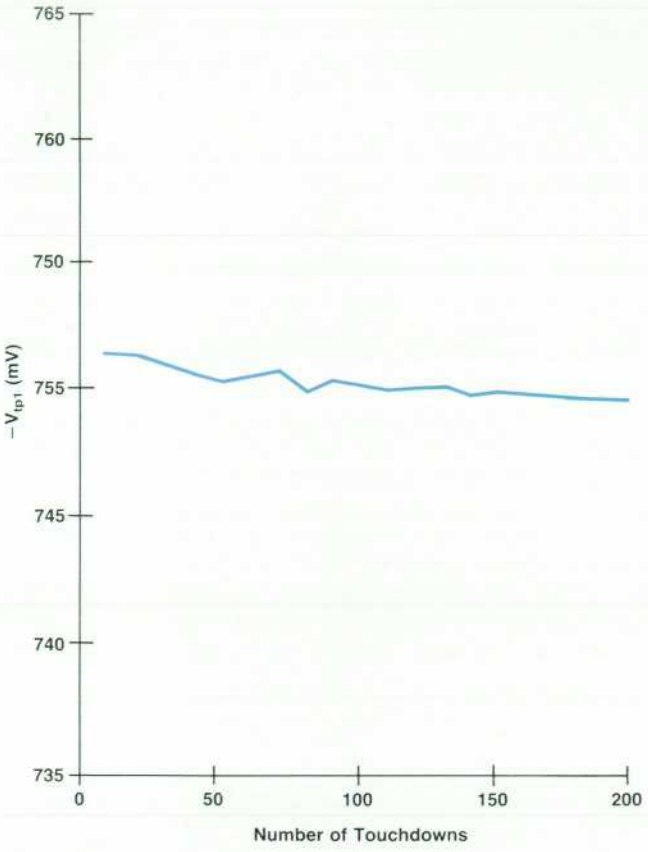


Fig. 13. *Measurement repeatability.*

and no significant damage to the gold bumps on the wafer was observed. Attempts to repeat the same test with a conventional wire probe card failed after only about 5000 touchdowns. The gold bumps on the wafer were severely damaged, especially at the edges, and the probe needles went so badly out of alignment that on-site repair was no longer feasible.

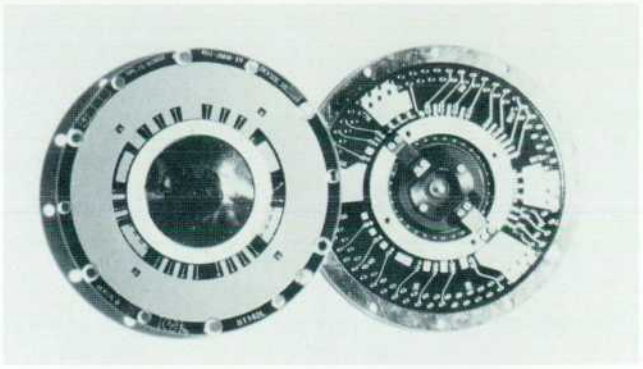


Fig. 14. *The membrane probe for DAC testing.*

One benefit of using the membrane probe card to test bumped wafers is the probe's ability to detect individual short bumps, a commonly encountered defect. Such bumps, if undetected, would cause assembly rejects at the subsequent inner-lead bonding step.

Conclusions

An advanced wafer probing technology, the membrane probe card, has been developed in response to an increasingly acute problem in the IC industry. The technology allows at-speed testing of high-performance integrated circuits at the wafer level, and significantly extends the limits of pin count and density that can be accessed by the IC test engineer. The new probe card has been fully evaluated, parametrically as well as in a number of alpha sites.

Acknowledgments

Key members of the membrane probe development team were Betty Belloli, Sam Burriesci, Brian Elliott, Michael Greenstein, and Rick Huff. Valuable contributions were made by Walker Colston, Jack Foster, Frank Perezalonso, Kazuo Ishii, Hiroshi Sakayori, and Miklos Perlaki. Brian Leslie managed the effort.

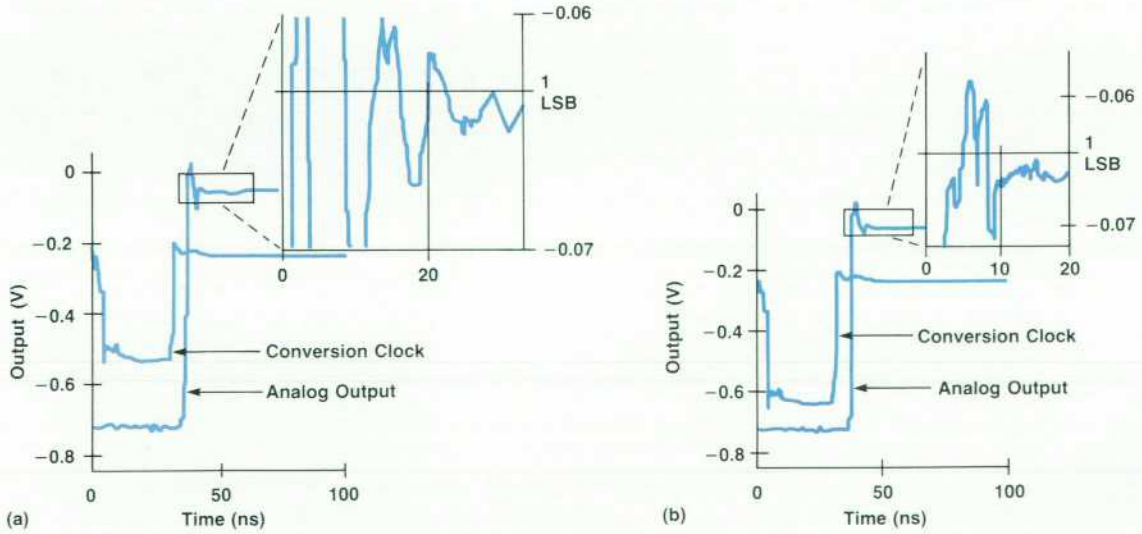


Fig. 15. *Settling time of the DAC analog output. (a) Conventional probe card. (b) Membrane probe card*

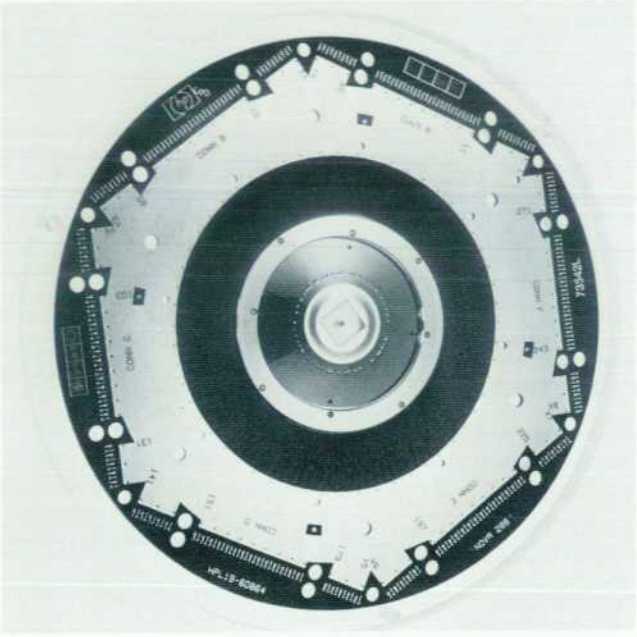


Fig. 16. *The membrane probe for microprocessor testing.*

References

1. R. E. Mattick, *Transmission Lines for Digital and Communication Networks*, McGraw-Hill, 1969, p. 191.
2. B. Oliver, "Time-Domain Reflectometry," *Hewlett-Packard Journal*, Vol. 15, no. 6, February 1964, pp. 1-7.

Additional Reading on the Membrane Probe:

3. B. Leslie and F. Matta, "Membrane Probe Card Technology (The Future for High Performance Wafer Test)," *Proceedings of the 1988 IEEE International Test Conference*, pp. 601-607.
4. B. Leslie and F. Matta, "Wafer-Level Testing with a Membrane Probe," *Design and Test of Computers*, February 1989, pp. 10-17.

Authors

June 1990

6 HP OSF/Motif

Axel O. Deininger

Making computers more intuitive and fun to use is the professional interest of Axel Deininger, a software design engineer who joined HP in 1982. He was HP's technical representative to OSF for user interfaces, the architect for defining OSF/Motif behavior, and coauthor of the style guide for the HP OSF/Motif product. He was a Learning Products project leader with the X Window Systems marketing development team at HP, and a manufacturing systems manager. Currently, Axel is investigating enhanced typographic capabilities for the X server. Before joining HP, he was an information systems specialist with Hughes Aircraft Company in El Segundo, California, and a systems analyst with Marsh & McLennan in Seattle, Washington. He has written HP manuals on X systems, the HP Vectra CS, HP Integral personal computers, and HP calculators. Born in Potsdam, East Germany, Axel resides with his wife in Corvallis, Oregon. He enjoys hiking, gardening, and traveling around the Pacific Northwest.

Charles V. Fernandez

After joining HP in 1988, learning products engineer Charles Fernandez coauthored the OSF/Motif styleguide and the HP OSF/Motif styleguide, and helped document the X Window System and related products. Charlie received a BA degree (1972) in English from the University of Detroit, an MA degree (1975) in English from the University of Oregon, and a journeyman carpenter certificate (1978) from Carpenter College in Adair Village, Oregon. Currently, he is the documentation project leader for the HP Visual User Environment (VUE). His professional interests include user interface design, on-line help and documentation, and minimalist hard-copy documentation. Charlie is the author of articles on fly-fishing, a member of the Federation of Flyfishers, and the Mid-Valley Chapter co-president of the Society for Technical Communication. Born in Gloversville, New York, he is married and lives in Eugene, Oregon. His hobbies include fly-fishing, science fiction writing, reading detective stories, camping, and cross-country skiing.

12 HP OSF/Motif Window Manager

Keith M. Taylor

As a software development engineer, Keith Taylor was a member of the engineering team for the HP window manager and HP OSF/Motif products. A graduate of Oregon State University with a BS degree (1978) in electrical engineering, and Stanford University with an MS degree (1989) in computer science, he joined HP in 1981 in Corvallis, Oregon. Keith also worked on the Microsoft Windows driver for the portable HP Vectra CS, on Microsoft Multiplan for the HP Integral personal computer and HP Series 200 computers, and on HP Word/80 for the HP Series 80 computers. He is now working on a new version of HP OSF/Motif window manager. Before joining HP, he designed software for real-time command and control systems for GTE in Mountain View, California. Keith is a member of the ACM, the IEEE, and the IEEE Computer Society. His professional interests include user interface software and issues related to the UNIX community. He served in the U.S. Army Military Police. He is active in civic affairs as a member of the Corvallis Citizens Advisory Commission on Bicycles. Born in Honolulu, Hawaii, Keith lives with his wife and two children in Corvallis. He enjoys swimming, bowling, bicycling, reading, and watching off-beat movies.

Brock C. Krizan

Soon after graduating from the Massachusetts Institute of Technology with BS and MS degrees in computer science in 1977, Brock Krizan joined HP's General Systems Division. He was a summer co-op student at HP's Data Systems Division for four years prior to joining HP full-time. He was a member of the HP window manager and HP OSF/Motif window manager development teams. Brock worked on a microcode assembler for the HP 3000 system, an HP 2100 data communications driver, and an HP 300 data communications package. He provided quality assurance support for the HP Series 80 computer and developed the HP Personal Application Manager (PAM) for the HP Integral personal computer and HP 9000 Series 300 computer. He was a member of the initial X development team at HP, and developer of an X-based emulator package for an HP proprietary window system. He is now a project manager for system user interface components such as the HP OSF/Motif window manager. His professional interests include user interface software and artificial reality. Brock authored a previous *HP Journal* article on the PAM screen for the HP Integral PC. Born in Yonkers, New York, he lives with his wife and two daughters in Corvallis, Oregon. His hobbies include mountain biking, cross-country skiing, white-water canoeing, hiking, and fine art appreciation.

26 HP OSF/Motif Widgets

Benjamin J. Ellsworth

Sailing and furniture making are the hobbies of Benjamin Ellsworth, an R&D software engineer who helped design the HP OSF/Motif widget product. He joined HP in 1985, shortly after graduating magna cum laude from Brigham Young University with a BS degree in computer science and statistics. In the past, Benjamin worked on HP common X interface widget development and CAEE ASIC vendor support. Before joining HP, he performed data communications test work for IBM in San Jose, California. He is the author of two technical articles on the HP common X interface. His professional interests include computers and human interaction. Benjamin lives in Corvallis, Oregon.

Donald L. McMinds

Don McMinds is the documentation project leader for the X Window System portion of the HP-UX Release 8.0 operating system. He joined HP in 1982 as a programmer-analyst in MIS. In 1986, he became a learning products engineer and wrote the HP Basic Programmer's Manual, the HP Vectra CS Service Manual, and many other user's manuals for HP Vectra peripheral equipment. He received a BS degree (1964) in engineering, and an MA degree (1973) in management from the University of Nebraska. During a 23-year career in the U.S. Air Force, Don served as a Titan II missile crew commander and in staff positions at the Strategic Air Command (SAC), North American Air Defense Command (NORAD) headquarters, and at Military Airlift Command (MAC) headquarters. He retired as a lieutenant colonel in 1982. Born in Wenatchee, Washington, Don lives with his wife in Corvallis, Oregon. He has twin daughters who are both sophomores in college. His hobbies include racquetball, ham radio (licensed since 1963), model railroads, and fly-fishing and fly-tying.

36 HP SoftBench Environment

Martin R. Cagan

As a project leader with HP's Software Engineering Systems Division, Martin Cagan helped develop the HP SoftBench system. Marty joined HP in 1981 as a software engineer and worked on commercial applications and environments for the HP 3000 computer. He also developed software tools, including the HP AI workstation, as a project leader and man-

ager at HP Laboratories. He earned a bachelor of science degree in computer science in 1981 from the University of California at Santa Cruz. Marty, whose professional interests include application integration and software development environments, recently left HP to join a CASE company in San Francisco.

48 Software Development Tools

Colin Gerety

After joining HP in 1985 in Fort Collins, Colorado, software development engineer Colin Gerety developed the HP SoftBench editor, remote data access, automatic menu generation, and dialog box handling systems. He also worked on HP Common Lisp and expert system shell investigations. He is named a coinventor on two patents pending for HP SoftBench—execution management and the broadcast message server. Colin earned a bachelor's degree (1979) in music at the University of New Mexico and a master of science degree (1988) in computer science from Oregon State University, with a focus on truth maintenance systems. Colin was born in New Haven, Connecticut, and he and his wife and four children reside in Fort Collins, Colorado. His hobbies include music, bicycling, and studying human society.

59 HP Encapsulator

Brian D. Fromme

Brian Fromme designed and developed the HP SoftBench subprocess control facility and the HP Encapsulator as a software development engineer with HP's Software Engineering Systems Division. Earlier, he developed a compiler for HP Business BASIC/3000 and, at HP Laboratories, he worked on Lisp compiler technology and ported Lisp to HP Precision Architecture. Currently in sales development, Brian joined HP in 1983. He has also worked in software development at the Artificial Intelligence Center of SRI International. A graduate of the State University of New York College at Brockport, he earned bachelor of science degrees (1982) in computer science and mathematics. He is named an inventor in a pending patent for the HP Encapsulator. Brian volunteers teaching time to schools through the HP Visiting Scientist Program near his home. He was born in Rochester, New York, and he and his wife and two children live in Fort Collins, Colorado. He enjoys all sports and plays baseball, football, and basketball.

69 Particle Beam LC/MS

Robert G. Nordman

A project manager for the HP 59980A particle beam interface for LC/MS, Bob Nordman has worked on a variety of HP products since he joined the company in 1969. He was a development engineer for the HP 7970A tape unit, HP 7920A disc drive, HP 2644A terminal, and HP 8450A diode array spectrophotometer. He was a project leader for the mechanical portion of the HP 8451A diode array spectrophotometer and a project manager for the HP 8452A diode array spectrophotometer. Bob is named an inventor in three patents on tape drive technology and one on the particle beam system. Before joining HP, he helped design mechanisms for aerial cameras for Hycon Manufacturing Company, computer peripherals for Burroughs Corporation, home appliances for Fisher & Paykel (New Zealand), and computer peripherals for Bell & Howell Corporation. He received his BSME degree (1951) from the Massachusetts Institute of Technology, and his MS degree (1973) in engineering from Stanford University. A first lieutenant in the U.S. Air Force from 1951 to 1953, Bob was born in New York City, and he and his wife now live in Palo Alto, California. He has three sons and three granddaughters, and enjoys letterpress printing, gardening, reading, and playing guitar.

James A. Apffel, Jr.

Since joining HP in 1983, Alex Apffel has helped develop the HP particle beam LC/MS system and the HP AminoQuant amino acid analyzer. After earning a bachelor of science degree (1978) at the University of Hawaii at Manoa and a PhD degree (1981) at Virginia Polytechnic Institute, and after completing postdoctoral studies (1981-83) in analytical chemistry at the Free University of Amsterdam, Alex joined HP in Waldbronn, West Germany. He has published ten technical journal articles on chromatographic methods, and is named an inventor in patents on amino acid analyzer chemistry and automated precolumn derivatization. Before joining HP, he worked as an applications chemist for Varian Associates. Alex is a member of the ACS and the ASMS. Born in Coronado, California, Alex is married and lives in New Almaden, California. He enjoys skiing, reading, and working with personal computers.

77 Membrane Probe Card

Farid Matta

As an R&D project leader in the HP Circuit Technology Group, Farid Matta was involved in the development of the membrane probe technology. He has also served as process engineering manager for CMOS IC production and is now working on advanced TAB packaging as an R&D project manager. He joined HP in 1981. Farid is an author or coauthor of 20 technical publications on IC processing, testing, and packaging, and is coauthor of a book on electrical contacts and interconnects published in the USSR in 1974. He is named an inventor in a number of patents issued or pending on IC testing and packaging, and is a member of the IEEE and the CHMT. Before joining HP, he worked on bipolar IC fabrication at Advanced Micro Devices in Sunnyvale, California, and was an assistant professor at the Institute of Electronics in Menouf, Egypt. He received his BSEE degree (1965) and his PhD degree (1972) in microelectronics from the Leningrad Institute of Electrical Engineering. Farid was born in El Menya, Egypt, and lives with his wife and son in Mountain View, California. He enjoys painting, art history, and hiking.

FR: DICK DOLAN 00152529
445

TO: LEWIS, KAREN
CORPORATE HEADQUARTERS
DDIV 0000 203R

Hewlett-Packard Company, 3200 Hillview Avenue, Palo Alto, California 94304

ADDRESS CORRECTION REQUESTED

Bulk Rate
U.S. Postage
Paid
Hewlett-Packard
Company

HEWLETT-PACKARD JOURNAL

June 1990 Volume 41 • Number 3

Technical Information from the Laboratories of Hewlett-Packard Company
Hewlett-Packard Company, 3200 Hillview Avenue
Palo Alto, California 94304 U.S.A.
Hewlett-Packard Marcom Operations Europe
P.O. Box 529
1180 AM Amstelveen, The Netherlands
Yokogawa-Hewlett-Packard Ltd., Suginami-Ku Tokyo 168 Japan
Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive, Mississauga, Ontario L4V 1M8 Canada

CHANGE OF ADDRESS: To subscribe, change your address, or delete your name from our mailing list, send your request to Hewlett-Packard Journal, 3200 Hillview Avenue, Palo Alto, CA 94304 U.S.A. Include your old address label, if any. Allow 60 days.

5953-8581