The Mesa Programming Environment

Richard E. Sweet
Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

1. Introduction

People everywhere are developing multi-window, integrated programming environments for their favorite computers and languages. This paper describes the Mesa programming facilities of the Xerox Development Environment (XDE). It is interesting for several reasons. It has existed in something similar to its current form for about 5 years. It has more than 500 users, many interacting with it 8 or more hours a day. Several million lines of code have been written by these users, including large, multi-author systems.

Previous papers have dealt with the Mesa language [Geschke77, Mitchell79], the operating system [Redell79, Lampson80] and the processor architecture on which it runs [Johnsson82, Sweet82]. This paper describes the programming environment: the user illusion, the set of programming tools, and the facilities available for augmenting the environment. Section 2 gives a short history of the environment, including some of our original design goals. Section 3 describes the current state of the user interface and discusses a few of the schemes that were tried and discarded. Section 4 describes some of the program development tools available and discusses how features of the language have influenced their design, and indeed influenced what tools are in the set. Section 5 describes other tools that, although valuable to the programming task, are largely language independent. Section 6 talks about how easy it is to make additions to the system, and gives examples of user additions-some that modify the environment and some that simply provide new tools. Section 7 discusses what we feel are major successes and what we feel needs to be done in the future.

2. History and Philosophy

Much of the early history and design motivation behind the Mesa language project was reported in Early Experience with Mesa [Geschke77]. This section deals with the history and design goals of the programmer's *environment* that grew up around the language. This paper principally concerns the facilities on individual programmers' workstations, although a good bit of their power comes from being part of a larger network *environment*. In addition to the workstations, this network contains a number of server machines. Some servers are used for central storage of files, some for printing, some for communications with the outside world, and some provide small specialized services. The individual workstation, nonetheless, provides the majority of a programmer's computing power.

2.1 Chronology

Mesa was first implemented on the Alto [Thacker79] in 1976. The development environment was the Alto operating system [Lampson79] whose user interface was the Executive, a BCPL program that operated much like the time-sharing executives of the day. When one invoked a tool, such as the compiler, it ran in the entire machine (along with pieces of the operating system), and when it was finished, the Executive was reloaded to process the next command. There were provisions for chaining together commands by means of disk files with distinguished names.

The Mesa debugger was patterned after the BCPL debugger to the extent that it used the same world *swap* principle. When the debugger was invoked, the contents of memory were written to a disk file, the *client outload* file, and then memory was loaded from another file, the *debugger* outload file. The net result was that you were now running inside the debugger and could examine and modify the client world by reading and writing the client outload file. The debugger had the ability to set breakpoints and to display program data in a format determined by the type of the data. It had a crude ability to call procedures in the client code. The debugger had multiple windows patterned after those of Smalltalk [Kay76]. One, a typescript window, was used for

interaction with the debugger executive for setting breakpoints, examining data, etc., and another was used **to** show the source context at breakpoints.

In 1977 (Mesa 3.0), we introduced strong *intermodule* type checking by means of *interfaces* and the *Binder*. The ability to check procedure parameters at compile or load time was a clear win. It often took a while to produce a consistent version of a large system, but once it could be loaded, it would usually run well enough that the programmer could start looking for logic errors. The early days of binding are described in some detail in a paper by Lauer and Satterthwaite [Lauer79].

In 1978 (Mesa 4.0), we added monitors and condition variables to the language to facilitate parallel computation. These features were of particular use in implementing the multi-level communications protocols in our internet environment.

By April 1979 (Mesa 5.0), the window package was much more integrated into the debugger. Breakpoints were set by pointing to the source, and text could be copied from one window into another to avoid unnecessary typing. The source window was readonly, however; there was no way to edit programs from within the debugger.

Version 6.0 of Mesa (October 1980) was the first that could reasonably be called an integrated environment. Programmers tended to do most of their development work inside the debugger. There were several advantages to this: the world swap discipline meant that the debugger world persisted across the running of applications; one could edit in several windows at once, and use a file transfer window to get needed files from remote servers. There were, however, still applications that required the programmer to exit from the debugger and talk to the Alto executive.

The major shortcoming of the Alto was the lack of virtual memory, so in the late 70's several machines were developed to replace the Alto and the Pilot operating system [Redell79] was written to replace the Alto operating system. The machines included the Dolphin (1978), the Dorado (1979), and the Dandelion (1980). The Dandelion is the machine on which the Xerox 8000 Series products are based (including Star and Network Services). It is also the machine on which the current Mesa development environment runs. For the sort of applications that it typically runs, a Dandelion has computational power somewhere between a VAX 750 and a VAX 780.

The first Pilot based debugger looked a lot like the Alto one, even sharing a lot of source code for the debugger proper. The major addition was a new Executive window. This window was the access to a program that was a transliteration of the Alto Executive. It allowed the user to run the more batch-like applications such as the compiler and binder, and included such niceties as command file expansion. Many of the Alto applications were rewritten to be run from this executive. The major difference was that the world was not reloaded after an application finished; an application had to be more careful about freeing up any resources it had acquired. There was still a one-thing-at-a-time mentality. For example, the compiler turned off the display when running in order to get more cycles.

Various experiments were made where tools would fork a (pseudo)parallel process to do their actual work. This met with limited success since some of the system resources, notably the file system, were not particularly reentrant. As a result, the file system was replaced [Reid83], and by April 1982 (Mesa 8.0), there was a truly integrated environment that allowed parallel execution of most of the tools. To the typical programmer using the environment, this was a quantum leap forward. I suspect that coffee consumption went down, since there was no longer the need to find something else to do while the compiler was running; you could read your mail, or edit the next program to be compiled.

In the next two years, the underpinnings of the system were largely rewritten (often with little visibility to the applications running on top). The display management was rewritten to improve performance and further work was done to make concurrent execution of applications more robust. The virtual memory and low level file management of Pilot were rewritten and the instruction set of the Dandelion (implemented by RAM based microcode) was reworked [Sweet82]. Of course, the collection of tools increased in number.

2.2 Design Goals

There were a number of goals and features of the language effort that had an impact on how the environment was structured. From the beginning, our goal was to allow development of large complex software projects with sufficient performance to be usable in a production environment.

- Mesa supports *modular* programming with strong compile-time type checking. The strict type checking necessitated the creation of several tools in the environment for helping to maintain consistency.
- Procedures are first class values; they can be passed as parameters or stored in data structures. This aids *object oriented* programming and delayed binding of implementations.
- There is a strong commitment to source level debugging. As the language and compiler evolved, special care was taken to ensure that programs could still be reasonably debugged.
- There is sufficient runtime efficiency and access to underlying architecture that all levels of the system may be written in a high level language.

Around 1978, a "Tools" project was started to design a software development environment. This project was later merged into the Mesa project, and the user interface portion was named Tajo (pronounced TAH-hoe). Its design was influenced significantly by previous work of Kay and of Swinehart [Kay69, Kay76, Swinehart74]. Below is a list of basic precepts from the 1980 Tajo Functional Specification [Wallace80]:

- *Client programs shall not preempt the user (Swinehart's Law).* The user should never be forced into a situation where the only thing he can do is interact with only one tool.
- *Don't call us, we'll call you (Hollywood's Law).* A tool should arrange for Tajo to notify it when the user wishes to communicate some event to the tool, rather than adopt an "ask the user for a command and execute it" model.
- *The user owns the window layout.* Individual tools should make minimal assumptions about the size and position of any windows that they own. The procedural interface to writing on windows should make such concerns largely invisible to the client code.

There is another important point. By this time, the Mesa project had moved from corporate research into a development organization. While some of the same designers worked on it in both places, there was now a definite *engineering* flavor to the effort. Several times a 90% solution was implemented since it could be done *today*, rather than wait until we understood how to completely solve the problem.

3. User Illusion

For the use of multiple tools in multiple windows to be most effective (for changing activities quickly and easily), the user interface must be consistent across all tools. The perceptions, model, and conjectures that a user accumulates about a system are referred to as the *user illusion.* The intent underlying Tajo is to create a consistent user illusion that enables the user to predict instinctively how to use any tool, regardless of previous exposure to it. The principle is sometimes called the *Law of Least Astonishment.*

First a few words about the hardware. The Dandelion has a 17 inch CRT oriented with the larger dimension horizontal. The display image is divided into an array of pixels 1024 across by 808 down. The image is refreshed from a region of memory, where each pixel of the screen corresponds to a bit in memory. Such a display is often referred to as a *bitmap display.* A zero bit in memory causes the background color to be displayed, a one bit does the opposite. Most users run Tajo with a white background, although changing a parameter in the user profiie will allow white on black operation.

There is a keyboard and a mouse with two or three buttons. The software makes limited use of the middle mouse button, so it is "pushed" on a two button mouse by chording the other two buttons. The coordination of movement of the mouse on the desk and the movement of the cursor on the screen is done by software. There is a process running at high priority that notices changes in the location of the mouse and moves the cursor correspondingly, subject to keeping the cursor on the screen. It is possible for an application to replace this procedure so that, for example, a drawing program can make the cursor stay on a grid.

The Mesa language allows easy creation of multiple parallel processes in the same address space by means of

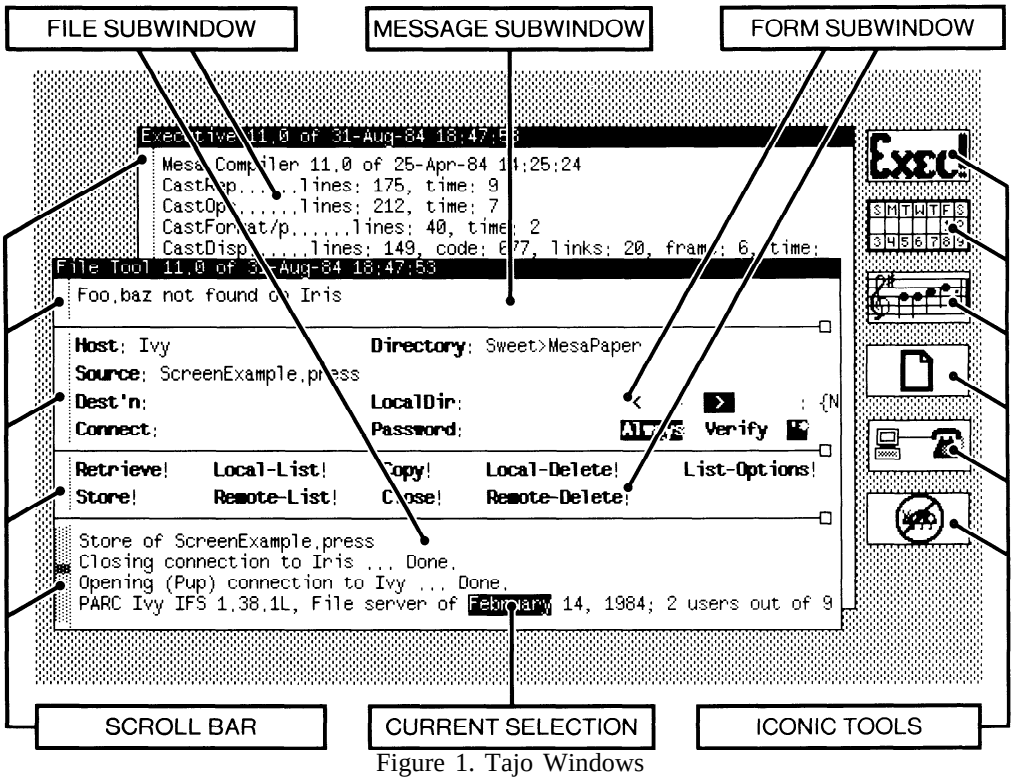


Figure 1. Tajo Windows

the FORK operator. This feature is heavily used in the XDE user interface paradigm. There is a process called the *notifier process* that is responsible for most user action processing. Client programs that have a non-trivial amount of work to do usually FORK a process to do the actual work and return to the notifier so that the user can do other things in parallel. There are three client process priorities: *background normal,* and *foreground.* The notifier runs in foreground, but client programs often reduce their priority to normal or background while running.

This section gives a brief description of the current (Mesa 11.0) version of Tajo. It has two somewhat independent goals: to describe the user interface features (as there is no public reference I can cite, and until recently XDE has not been widely available), and to give insights into the language features used to implement these features. Whenever possible, if a subsection tries to meet both goals, the descriptive material is given first.

3.1 Windows

Interaction with the user is through *windows* on the screen. A window is a rectangular area of the screen, and may be covered with *subwindows,* resulting in a *window tree.* Each window has a size and a position relative to its parent. Windows at the "top level" are children of a system supplied full-screen *root window.* The data structure used for representing this tree is a LISP-like scheme where each window points to its first child and to its next sibling. Windows can overlap in arbitrary ways, but the rules for visibility are easy to state in terms of the tree structure. Windows obscure their parents, but are clipped where they extend beyond the dimensions of their parent. If two overlapping windows are siblings, the one earlier in the sibling list is on top. Figure 1 shows several tool windows. The windows of the File Tool and the Executive are overlapping siblings, subwindows of the File Tool are non-overlapping siblings.

The system provides routines for adding *scrollbars* to subwindows. All of the subwindows in Figure 1 have vertical scrollbars. The scrollbar changes color when the cursor moves into it, giving feedback on that portion of the file currently visible. In the bottom subwindow of the File Window, the dark grey portion of the scrollbar indicates that the visible portion is about 40% into the file and is about 10% of the total file size. While the cursor is in the scrollbar region, the three mouse buttons are used for positioning. The left means "scroll up," the right means "scroll down," and the center means "thumb to this place in the file."

Each window is implemented by an associated *window object,* which in turn is accessed by a *window handle,* a pointer to this object. The object contains information about the tree structure, this window's location with respect to its parent, and several procedure variables, such as a *repaint procedure* that gets called whenever a portion of the window has invalid contents. The implementation makes use of Mesa's opaque type mechanism, so that client programs do not depend upon the actual representation of the window object. There is, however, a procedural interface whereby client programs can manipulate the information in the object.

Client programs can attach additional information to windows by means of named *contexts,* similar to property lists of LISP. Tools can be written to have *multiple instances,* obtaining their data from the context associated with the particular window that Tajo is notifying (passed as a parameter to the notify procedure). See §3.5 for further details.

3.2 Subwindow types

The user has control of the placement, overlap, and size of top level windows; maintaining the remainder of the window tree is the responsibility of client programs. Consider, for example, a tool that divides its window into two equal sized subwindows. Whenever the user changes the size of the enclosing window, the sizes of both subwindows and the relative origin of the second one must change. Tajo allows the client program to register an *adjust procedure* that gets called before and after the window changes dimensions. Since adjustment strategies are often shared by multiple window instances, a classing scheme called *window types* is used. Every window has a type, and associated with every window type is a collection of procedures. In addition to the adjust procedure, there is a *wakeup procedure* that is called when a window first becomes visible and a *sleep procedure* that is called when it will no longer be visible until awakened.

To simplify the task of the tool writer, there are several system-provided subwindow types, each with its own runtime management routines. Some tools have only one subwindow, like the file subwindow for the log of the Executive in figure 1. Some tools have multiple subwindows of the same type, like the two form subwindows of the File Tool. The three most popular types for incorporating in tools are the following:

- A *message subwindow* is one for giving small amounts of feedback to the user, such as error conditions. These windows typically have only a few lines, and as information scrolls off the top it is thrown away.
- *A file subwindow* is used by tools that want to create a log of their operations. The client program simply writes to the log by standard stream operations. As characters are written to the log, they appear on the screen and also go into the file. The built-in management routines take care of things like line breaks, scrolling, window splitting, selection, etc.
- A *form subwindow* is the major work horse for the *interactive* portion of a tool's user interface. These windows can contain form *fields* to be filled in (with either text or numbers), *choices* to be made (either by menus or one-of-many selection on the screen), and *command* buttons to be invoked with the mouse.

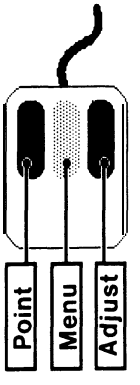
Client programs are welcome to define their own unique subwindow types, and there is a predefined type *vanilla* that works for many types of client managed windows that don't need fancy maintenance.

3.3 Tool state

When a tool has its full-sized window open on the screen, it is said to be in an *active* state. This is like having a woodworking tool out on the workbench, ready for use. Tools that will not be needed for a while can be shrunk to an iconic form on the screen; these tools are in *the tiny* state (see figure 1). A tiny window typically retains all window state, such as parameters that the user has given the tool, options selected, or messages posted by the program. In the physical metaphor, think of tools that are placed in a tool belt, convenient for use, but not cluttering the bench. The third state for tools is *inactive.* When tools are deactivated, they disappear from the screen entirely. By convention, they free up any system resources that they are using, including the previously described window state. They are, however, added to a menu of inactive tools so that they can be subsequently re-activated. These tools correspond to those put back into the tool box.

The tool data contains a client supplied *transition procedure* that gets called whenever the tool changes state. System-defined subwindow types such as form subwindows have system supplied transition strategies that "do the right thing." For example, file subwindows close the backing file when the tool is deactivated.

3.4 Selection



Two mouse buttons are used for selection. The left button, called **Point,** is used to begin the selection. When **Point** is depressed, the closest character is selected (video inverted), and the selection tracks the mouse until the button is released. In this sense, one should think of selection taking place on the button's *up* transition. Units larger than a single character are selected by *multiple clicking.* If the left button is pressed twice in rapid succession, the *selection mode* becomes word-selection. Three clicks select lines, and four clicks select the entire document. Multiple clicks cycle through the selection modes, so a 5-click is equivalent to a single click.

The right mouse button, called **Adjust,** is used to extend or contract the selection. Pressing **Adjust** causes the closest end of the current selection to move to the cursor position, subject to selection mode constraints (e.g., to the end of the current word if in word mode). As with **Point,** the selection tracks the cursor and "commits" on the up transition. There is one subtle, but quite useful addition. If the down transition of **Adjust** occurs when the cursor is over *the* last (or first) *character* of the selection, the selection mode reverts to character for extension purposes.

Other selection schemes have been tried and discarded. They include draw-through selection and separate character and word select buttons. Other indications of selection were tried as well, including underlining the selection and drawing a box around it.

There is a single global selection in the environment. Client programs that wish to deal with the current selection do so by calling a system interface **Selection.** Such clients fall into two different classes. The most common are those that wish to obtain the *value* of the current selection. This is done by calling the procedure **Convert.** This procedure takes a parameter describing what the client wants to know about the selection, e.g., its value as a string, its position within a file, the window containing it, or its value parsed as a number. The most common target is a string. For very long selections, copying the text to a string would be inefficient. In this case, **Convert** will refuse to return a string, and the client can ask for a *source,* a stream-like data structure that allows sequential access to the characters of the selection.

The other class of clients that deal with the selection are those who want to *manage* the current selection. This is done by calling the procedure Set, giving it two procedures: one that gets called when the global function **Convert** is called, and the other that gets called for various actions like un-highlighting the text when the selection changes. For most text selection, the calls to **Set** are at a level of abstraction far below that seen by typical client programs. An example of a client program where the programmer explicitly calls **Set** is a graphics editor where the client wants to make selected text in its window available to another tools that read the current selection.

One more subtle point on selection-when a client program is the manager of the selection, its conversion procedure need not support all of the potential target types defined in the interface. The conversion procedure is welcome to return NIL for any (or all) types. In the graphics editor example, the "position in the file" would have to be a two dimensional quantity rather than the scalar quantity it is for text files, so the editor would return NIL.

3.5 Keyboard

There is the concept of a global *input focus* in Tajo. This focus is associated with a particular subwindow. The input focus typically goes with the current selection, but the user interface provides a means for setting it independently. Within the subwindow there is an *insertion point* which falls between two characters. When the selection is moved, the insertion moves to the end of the new selection (character, word, line, document) closest to the position of the mouse when the mouse button **Point** is released. For example, clicking 3 times in the first half of a line selects the line and moves the insertion to the beginning of the line. The user profile can specify options that change the tracking of the insertion point, e.g., to always place the insertion at the end of the selection (as opposed to the beginning).

If a client program thinks of the keyboard as an ASCII input device, it can call a system procedure to associate a *string input procedure* with any of its subwindows. Whenever a character key is typed and that subwindow has the input focus, the procedure will be called. There are three keys that bear further discussion. The **Stuff** key simulates type-in of all of the characters of the current selection. The **Copy** key works as follows: when **Copy** goes down, the current selection is cleared. The user now makes a new selection-the tracking of the insertion point is disabled during the selection. When **Copy** goes up, this new selection is stuffed into the insertion point. The Move key works like **Copy** except it simulates a **Delete** after the copy has been done.

Being called with type-in is certainly consistent with the "Don't call us, we'll call you" philosophy described in §2.2, but some programs would rather ask for characters than be told about them. Such programs are usually transliterated from traditional timesharing environments where the program worked on the model that it owned the whole virtual machine (and the user as well). This style of interaction is supported by *teletype subwindows,* another system supplied subwindow type. These supply an abstraction similar to that of file subwindows, but with the requisite input/output model. The only subtlety is that when the client program is started, it must be careful not to read characters before it FORKS itself a process separate from the notifier.

Some client programs wish to give non-standard interpretations to the keyboard or mouse. The rest of this section describes the low level interfaces to user input. The keyboard of the Dandelion is non-encoded; there is a region of memory from which a program can read the current up/down state of each key. Very few applications deal with the keyboard at this level; they use a facility called TIP (for *terminal interface package).* At the heart of this package is a collection of *TIP tables,* descriptions of desired operations for specified user actions. These tables are associated with windows in a tree structure.

A high priority process watches the hardware for user actions. These actions are then enqueued along with their times of occurrence in a *user action queue.* A user level process, the *notifier,* removes these actions and matches them against the proper TIP tables.

If it finds the action, it passes the corresponding operations to that window's associated *notify procedure.* Some user events, such as most keystrokes, are for the window that currently has the input focus. Other events, such as most mouse button clicks, are directed to the window that contains the cursor.

A TIP table looks like a large SELECT statement with user actions as the selector arms. There are two classes of actions that can appear: TRIGGER actions refer to actions that are removed from the user action queue, such as a key going down or up, ENABLE actions interrogate the current state of the user interface, such as a shift key being down. If a left-side is matched, the notify procedure is called with the list of result items on the right-side. For example, the fragment

```
SELECT TRIGGER FROM
  . . .
  I Down WHILE COMMAND Down =>
        InvertScreen;
```

sends a single atom, **InvertScreen,** to the notify procedure. To get slightly ahead of ourselves, the line

```
Z Down WHILE COMMAND Down => Menu,
      "Window Mgr", Zoom;
```

sends two atoms, **Menu** and **Zoom,** and a string. This is part of a TIP table for a program that turns keystrokes into menu item activations. Menus are described in §3.6.

A more complicated example deals with the left mouse button going down. In this example, it could be part of a single or double click, or part of a chord simulating the middle button.

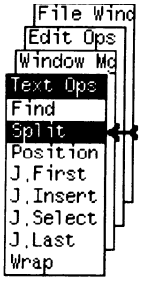
```
SELECT TRIGGER FROM
  . . .
  Point Down =>
    SELECT TRIGGER FROM
      Point Up BEFORE 200 AND Point Down
            BEFORE 200 =>
        SELECT ENABLE FROM
          LeftShift Down => COORDS,
                ShiftedDoubleclick;
          ENDCASE => COORDS,
                NormalDoubleclick;
      Adjust Down BEFORE 300 =>
            MenuDown;
      ENDCASE => COORDS, SimpleClick;
  . . .
```

An item in the result list (right-side) for an action is a variant record whose variant has one of the following types:

- A character corresponding to the key most recently pressed.
- The coordinates of the mouse **(COORDS** in the example).
- The full state of the keyboard.
- An atom.
- A number.
- A string.
- The time, in processor clock ticks, of the event.

3.6 Menus

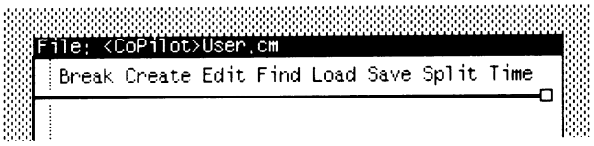
Another means for user interaction is via menus. Rather than have a single large menu, there are several menus available at any time, determined by the subwindow that contains the cursor. System supplied subwindow types have their own sets of menus, such as a text operations menu for a typescript subwindow. The set of available menus is obtained by depressing the middle button on



the mouse. Menus are stacked in such a way that the user can either invoke a menu item from the top menu or cause another menu to come on top.

From the programmer's standpoint, menus are easy to construct. There is a system procedure that is given a window handle and an array of <STRING, PROCEDURE> pairs. The strings become the menu items, and the corresponding procedure is called whenever the user selects that item. The system procedure returns a *menu handle* which can then be associated by other system procedures with one or more subwindows.

3.7 Symbiotes



Menus can provide many commands without requiring the user to remember the exact command names. However, for frequently used operations, it is cumbersome to remember which menu contains the desired command, bring it to the front, and select the item. One solution to this problem is *symbiote subwindows.* They appear at the top of other windows and contain a list of identifiers. When the user clicks the mouse button on one of these identifiers, the symbiote handler searches through all the item names from the collection of menus on the other subwindows of their window. If it finds a match, it calls the corresponding procedure. The TIP example in §3.5 shows how, with the. proper notify procedure, one could also turn keystrokes into menu activations.

4. Programming Tools

The previous section described the user illusion of XDE; this section describes some of the Mesa program development tools that made it possible to develop such an environment. The major thrust of this section is that the various tools are *highly interrelated;* each one either produces extra information for use by others or makes use of such information.

4.1 Runtime loader

The runtime loader is technically not a programming tool, but rather a part of the Pilot operating system kernel. Nevertheless, in order to understand the other programming tools, some knowledge of the runtime loader is needed.

A Mesa object program has, in addition to code, a header that specifies a list of *interfaces* that are *imported* or *exported.* These interfaces are the "glue" that holds modules together into larger programs. The binder specification language and operation are described in detail in the language manual [Mitchell79] and a paper by Lauer and Satterthwaite [Lauer79]. The runtime loader maintains a database of all interfaces imported and exported by programs now loaded (either part of the original boot file, or previously runtime-loaded). Loading a new module is a three step process:

1. Move the code from the object file into virtual memory and allocate space for the global frame of the module.
2. For all interfaces *imported* by this program, see if there is something already loaded that exports the desired items.
3. For all interfaces *exported* by this program, see if there is something already loaded that wishes to import any items exported.

Mesa does not restrict the programmer to have a single implementation module for a given interface, so it is possible for step 2 or 3 to only partially satisfy the import needs for an interface.

Note that the loader is capable of many of the tasks typically associated with the linkage editor of a more batch oriented system. Indeed, small multi-module systems are sometimes tested by simply loading the various modules. Large or publicly distributed systems are almost always bound together by the Binder described in §4.3.

Object programs also contain a list of *control modules.* Many Mesa modules are simply collections of procedures, but some contain mainline code as well. When a program is run, its control modules have their mainline code executed in the order specified when the object program was created. In keeping with the Tajo design principles, most programs either create windows for interaction or register commands with the Executive (or both) and then simply return.

Once a program is loaded, it usually stays around forever (or until the next boot, which may be a week or two away). There is, however, an *unloader* that goes through the three steps in the reverse order. Thus it is possible to load two highly interconnected modules, decide that one is buggy, unload it, make repairs, load the new one, and have the interconnections now properly made to the new module.

4.2 Compiler

The Mesa compiler is organized into multiple passes. This is partly to simplify dealing with a language that allows forward references, and partly because the initial implementation was on the Alto with a small memory. The first pass is a table driven LALR parser with semantic routines that construct an abstract syntax tree in virtual memory. Later passes decorate the tree, and eventually generate machine code as part of the object file. The code is for a stack architecture [Johnsson82] that is in turn implemented in microcode on the Dandelion. All jump instructions in this architecture are relative to the program counter, so the code requires no relocation by the loader. There is no separate assembler; all levels of the system are written in Mesa.

The unit of compilation is the Mesa MODULE. This is typically a collection of procedure declarations, together with global variable declarations, and sometimes

a small amount of mainline code. At the beginning of the source program, there is a declaration of those *interfaces* that are *imported* or *exported* by this module. As mentioned in §4.1, the compiler must produce a header on the object file that provides this import/export information to the loader.

The code and the binding information are all that are really needed for execution of a module. However, additional information is needed to support source-level debugging. The Mesa compiler writes out most of the compile-time symbol table for use by the debugger and performance tools. Unlike some published symbol table structures [Graham79], the Mesa symbol tables are organized in such a way that no information useful to debugging is destroyed in the compilation process.

The compiler also provides a *source-to-object* mapping facility. The symbol table contains a *body table,* which contains an entry for each procedure body or BEGIN...END block containing declarations. These entries are linked together into a tree structure showing the nesting relationship of the source program. Each entry in turn is associated with a piece of *the fine grain table*, a table that tells the program counter value associated with the first instruction of each *statement* of the source program. The exact representation of this table underwent significant changes in 1981 when allowances were made for the Packager (see §4.5).

What about optimization? In a nutshell, the Mesa compiler doesn't do a lot of optimization that rearranges the order of execution; when it does destroy the ability to do source/object mapping, it tries to arrange things so that the debugger can give the user feedback that this has happened. This is an example of a "90% solution." Being able to set source breakpoints is such a powerful capability, and works such a high portion of the time, that the users are willing to say "darn" in the few cases where it doesn't work, as long as the debugger is capable of determining these cases. (They have been known to say stronger things in the few cases where the debugger can't determine what's going on). The compiler/ debugger collaboration does an admirable job, but new ideas [Zellweger84] would allow it to do an even better one.

The Mesa language contains two classes of modules: PROGRAM and DEFINITIONS. The later class is used for defining interfaces and for sharing type definitions among several PROGRAM modules. When a DEFINITIONS module is compiled, its object file contains no executable code, but contains a symbol table like that produced for debugging a PROGRAM module. These tables are used to implement the *included symbol* capability of the language. For example, suppose module A obtains type definitions from interface B. When compiling A, the compiler opens the object file for B and copies information from the symbol table therein.

4.3 Binder

The output of the compiler is an object file containing the code for a single module. For many reasons, interesting programs are made up of more than one module. While the runtime loader is capable of linking together separately loaded modules, most medium to large systems are distributed as a single object file put together by a separate program called the *Binder*. This is done for several reasons, some more obvious than others.

- It is a lot easier to keep track of one file instead of 50, particularly given Mesa's strict type checking of interfaces.
- Resolving the linkage between modules is a time consuming process, so if all internal interconnections can be resolved by the Binder, the loader need only take time to resolve external linkage requirements.

  Packages can be shared among several applications more easily if bound together in convenient sized pieces.
- Interfaces can be *obscured* from other programs by binding the modules exporting them into a larger configuration that does not export that interface.
- The symbol table and debugging information produced by the compiler occupy a lot of space. The Binder need not copy this into the new object file, but need only change the header of the new object file to point back to the compiler output files. Optionally, all of this information can be copied into a new "symbols" file.

The input to the binder is a configuration description that contains a list of object files to be combined into a larger object file. It also contains a list of what is to be imported and exported by the new file, as well as optional information directing how the internal bindings are to be done. The language manual [Mitchell79] contains the syntax for the configuration description language.

Our original design (and each subsequent implementation) of the Binder provides complete control of binding, with several ways for the programmer to deal with multiple implementors of interface items. In the intervening six years, we have had very few configuration descriptions other than lists of object file names, which means "bind them together in the only way possible." Almost all of the complicated descriptions could be replaced by suitably staged sequential executions of the Binder on simple configuration descriptions.

4.4 Debugger

Mesa has a full-function source-level debugger that is used for debugging all levels of the system, including the operating system kernel. The debugger uses the symbol table and statement mapping information produced by the compiler. Its features include the following:

- The ability to set breakpoints. To set a breakpoint, the user loads the source file into a window, selects a place in the file, and executes a **SetBreak** menu command. As feedback, the debugger changes the selection to be the

first character of the statement where the breakpoint is set. If the source file is not available, but the symbols are, the user can still set a breakpoint at the entry or exit of any procedure by typing its name to the debugger command interpreter.

- The ability to interrupt a running program and find the context of any process running at the time of interruption. Equally important, it is possible to resume execution of a program interrupted in this fashion.
- The ability to walk up the links of the dynamic call chain, answering the question "Who called whom."
- The ability to examine and modify data in the client program. The debugger has an interpreter for the full expression syntax of the language. Since Mesa is strongly typed, the interpreter can use type information to print out the values in a more understandable fashion. Numeric types print as numbers, enumerated ones as the proper named value, and records print out as a list of field names, each with its value printed according to type.
- The ability to call procedures in the client program, type checking the actual arguments against the types of the formal parameters.

The Mesa debugger evolved from the Alto implementation and still uses the *world swap* principle [Lampson79] for insulating the debugger from the client code. There are several advantages to this scheme: one can set breakpoints down in the bowels of the operating system or in interrupt routines. There are also several disadvantages, of which the primary one is speed. When the Alto swapped its 128K-byte worlds, it took between 3 and 4 seconds. The Dandelion has a faster disk, but with the 768K bytes of a typical programmer workstation, it takes l0-15 seconds for the swap. Things get even worse with larger memories; the Dandelion will hold 1.5 megabytes on the standard memory cards.

One solution to this problem is *teledebugging.* The debugger's path to the client world is through a rather narrow interface. The routines that read the memory from the world "swapped" on the disk can be replaced with ones that talk to other machines on the Ethernet. Coupled with a small teledebug nub in the client code, these routines give essentially instantaneous turnaround. The client computer need not be geographically nearby, either; an implementor in Palo Alto can debug a program running on a machine in London.

4.5 Packager

The Pilot operating system supports a demand paged virtual memory. It also allows the programmer to specify swap units so that when a particular page must be swapped in, the operating system makes an effort to bring in its entire swap unit.

When the compiler generates code for a module, it puts the code for the various procedures into a contiguous set of pages in the output file (interspersed with readonly constants). When the program is loaded by the runtime loader, these file pages are mapped into virtual memory. The default swap unit for code is to swap the code for the entire module as a unit.

Procedures are often collected in the same module because they work on the same abstraction, or because they share common private definitions and data. It is often the case that some of the procedures of the module are used only for initialization, or that some procedure in module A is tightly coupled with another procedure in module B. The *Packager* is a tool that allows the programmer to associate procedures of a collection of modules into explicit swap units. It corresponds very much to the Chinese restaurant stereotype of "one from column A, two from column B."

The packager takes two files as input, one is Binder output, the other a packaging description. A packaging description is a sequence of swap unit descriptions. Each swap unit contains a list of modules and procedures from those modules. This is a considerable simplification, since an explicit list is only one of about nine ways supported for specifying the procedures in a swap unit. For example, one could specify "all procedures of M not already placed in swap units named A and B."

Recall that the source level debugging relies on compiler output tables that give a source/object mapping. These tables, together with the symbol tables take up lots of room; they sometimes account for 80% of the bulk of a compiler output object file. We did not want for the Packager to have to rewrite that information. Considerable care went into the redesign of the source/object mapping tables produced by the compiler to make them independent of the eventual entry point location of the procedures once they have been packaged.

4.6 Performance Tools

The packager discussed above is the *easier* half of an important problem, that of reducing working set size; the more difficult half is that of determining what the swap units should be. We have a number of tools in XDE to aid in answering this and other interesting questions. Indeed, a principal conclusion of Knuth's FORTRAN study [Knuth71] was that programmers should have and use more data about the runtime performance of their programs. The set of performance tools includes the following:

- Spy - a program that wakes up periodically (at interrupt level) and records the execution context of a given process (or set of processes) [McDaniel82].
- PerfPackage - a tool that allows the programmer to define a set of *nodes* in a running program, using the standard machinery for setting breakpoints. A set of legs, defined by pairs of nodes can also be defined. The PerfPackage replaces the breakpoint handler with one that counts the number of times that each node is visited, as well as the average time spent on each of the legs.

- Transfer counting tools-several tools to note all interesting control transfers (procedure calls and returns), and present various statistics about them.
- Page fault analysis tools-tools to record and present data about page fault behavior.

These tools all rely heavily on the source-level debugging information produced by the compiler to present their results in the most usable format. The Spy is an interesting example. One can first see in which modules a program is spending its time. Then for a set of modules, one can get statistics about the most time-consuming procedures. Finally, statistics can be collected at the statement level. Since the Spy works on a sampling principle, it is sometimes useful to switch to the PerfPackage to obtain exact statistics at the statement level.

4.7 Consistent Compilation Tools

The Mesa compiler chooses to paint RECORD types with the 48 bit version stamp of the object file of the module in which they are defined, typically an interface module. Thus if procedures in program modules X and Y both reference a record type from interface A, then X and Y need to be recompiled whenever A is. This leads to *compilation dependencies* determined by the include relationships of a collection of programs. A tool called the *IncludeChecker* tries to deal with these problems. Its original goal was to take a collection of modules and determine whether they were consistent with respect to versions of the various interfaces holding them together. It has grown to where it can be pointed at the current version of a program and told to create the sequence of Compiler and Binder command lines necessary to make a new one. It is similar to the UNIX™ program Make [Feldman79], but differs in at least two ways: it only cares about compilation and binding dependencies, and it determines these dependencies automatically from information in the object file.

An earlier, more ambitious, tool worked on the following premise: if, in the example above, the changes to A were in areas unused by X and Y, then they need not be actually recompiled, but merely updated to refer to the new version of A. Such a tool requires some care on the part of the Compiler to leave behind a suitable "audit trail" so that the later tool can determine exactly what was used from a given included module. This tool was not a success, quite possibly because it also tried to deal with the problem of an inadequate disk on the Alto workstation, and had the unfortunate side effect of sometimes deleting the only copy of the software that it was trying to make consistent.

5. Other useful tools

The tools described in §4 are a representative sample of those that are heavily integrated with the Mesa language. There are a number of other tools that run in the environment that are valuable for program development, but are not as language specific. This section gives a brief description of some of them with some motivation for why they came into existence.

5.1 Database tools

A system like XDE or Star involves a huge number of files, over 5000 for XDE alone. The version stamp system used for insuring consistency makes it critical that a programmer be able to find precisely the right version of each file used. A rather complete discussion of this problem can be found in Lewis's paper on software version control [Lewis83]. This section briefly describes two tools that aid in this process: **DFTool** and **Fetch.** It also describes two tools used for more general database applications: **Access** and **Adobe.**

Most programmer workstations have limited capacity local disks, typically 40 megabytes. For this reason, programmers spend a lot of time shipping files to and from file servers, where the "truth" resides. A program called **DFTool** is very helpful in minimizing the number of files that have to be actually transferred. The version for XDE is an outgrowth of Eric Schmidt's thesis work [Schmidt82] as modified by Brian Lewis [Lewis83]. This tool manipulates a collection of text files called *DF files,* which in turn describe collections of other files. It is best to think of a DF file as a *snapshot* of a given system or piece of a system. It lists the files that make up the system, complete with creation time and file server "home," and also lists those files that are needed from other DF files in order to recreate the system from the source. There are three major operations provided by **DFTool.** The operation *Bringover* retrieves the files from the server if they are not already on the local disk. The operation *SModel* stores any changed files onto the server and updates the DF file to reflect this. The operation *VerifyDF* analyzes the programs listed in the DF file and determines that they are consistent, and that all system files necessary for their recompilation are listed in the DF file.

In order to set source breakpoints and examine variables by name, the programmer must have the proper version of the symbolic information on his personal workstation. For the particular tool under development, this is easy, since the pieces were probably compiled there. For the files that are part of the released system, the problem is harder; the organization of the archive directory is designed to aid in the development process, not to simplify retrieval. For example, one might not necessarily think to look for the interface **BcdDefs** on the subdirectory **<Apilot>11.0>Mesa>Friends>.** There is a program called **Fetch** that greatly simplifies this process. As part of the release process, a text file is produced that gives the home of each file in the release, both its subdirectory and its DF file. A *fetch service is* run on a server machine with a slightly predigested version of this directory file. Individual programmers run **Fetch** which communicates over the network with the service. To obtain a copy of a file from the release. the programmer simply selects the name of the file and uses a menu to retrieve it (source, object, or both).

In a system with multiple implementors, it is important that no two persons be modifying the same file at once. To protect against this problem, XDE has a *program librarian.* The librarian runs as an Ethernet service that allows the programmer to check in and out tokens called *libjects* One or more files can be associated with a given libject, typically either DF files or Mesa source and object files. The programmer *checks out* a libject by running a program called **Access.** A reason for the checkout must be supplied. It is recorded (and optionally logged) in the librarian database. The default operation of **Access** is to also retrieve the associated file from whatever file server holds it. If the file is already checked out, the programmer is told the current holder's name and reason for having the file. There is a corresponding *check in* procedure that stores the file back on the server. Some libjects with no associated files are used by other programs for mutual exclusion.

Reasonably early in the lifetime of Mesa it became clear that something had to be done to keep track of bug reports, wishes, tasks, etc. A program called **Adobe** is the outgrowth of this need. While it started life as a bug-tracking tool, it now is a fairly general database. It is, in fact, a whole family of databases, called *Adobe systems,* all managed by a collection of tools that together make up **Adobe.** For a given system, there is a collection of typed fields that make up records. There are provisions for inverting the database on various fields to allow query operations, and user definable templates for generating reports. The records can be individually *checked out,* or *locked* for writing. In the case of bug reports, a maintainer will check out the bug report, fix the bug, and then check it back in.

5.2 Distributed computing tools

Although each programmer has a rather powerful workstation, XDE is very much a distributed environment. File servers are used for several good reasons: to share information, to gather together consistent snapshots, and to provide a backup in case (shudder) one's local disk crashes irretrievably. While the **DFTool** is useful for maintaining systems, casual access to remote file servers is done by a tool called the **FileTool.** Like many XDE tools, **FileTool** provides two user interfaces. One is a "two dimensional" one with forms to fill in and buttons to push (see Figure 1). The other is a "one dimensional" one that is accessed via the Executive window. This interface parses command lines into commands and parameters. The latter interface is the more useful with command files and batch processing (although most command files for building tools use the executive interface to **DFTool** instead).

A tool called **Chat** allows the user to create one or more windows on the screen that act as "glass teletypes" for interaction with other machine. There are options in **Chat** to emulate most of the popular smart terminals. A particularly useful application of **Chat** is in conjunction with a program called **RemoteExec.** This is a version of the **Executive** that replaces the input and output streams with ones that talk to a remotely connected computer. This allows a number of workstations, typically ones with more memory or larger disks, to be shared by several users. These *compute* servers can, for example, be used for large batch compilations without slowing down the programmer's personal workstation. Another use of **RemoteExec** is to allow the programmer to run programs on his personal workstation from any remote terminal, say one at home.

Electronic mail plays a major role in the development process at Xerox. Before programmers would give up their Alto workstations and move to XDE, it was necessary to provide access to mail, based on the Grapevine system [Birrell82]. The Alto based mail reading program was named Laurel, in keeping with PARC's affection for names from the *Sunset Western Garden Book.* The XDE version was whimsically named **Hardy.** Like its predecessor, **Hardy** provides multiple subwindows: a *table of contents* subwindow with the sender and subject of each message received, a *command* subwindow with buttons for the various operations, and a *text* subwindow where the messages are actually read. As the old 3 megabit research network is being phased out of operation, the PUP based Grapevine protocols are being replaced by more modern protocols on the 10 megabit network, and **Hardy** is being replaced by a similar program named **MailTool.**

5.3 Shortcuts

Mesa programmers tend to use rather long identifiers in their programs. The compiler doesn't limit the length, and with proper capitalization, this can help to make programs somewhat self documenting (at least that's one of the excuses I get from programmers that don't use comments in their programs). Also, some of the keywords are long and must be capitalized properly as well. XDE has a built-in abbreviation definition and expansion capability where the programmer can type a unambiguous prefix and hit the **Expand** key. The recently typed text is scanned backward to pick up a token which, if found in the dictionary, is replaced by the expansion. Hitting **Expand** with **Shift** down causes a window to open up where the user can define a new abbreviation.

6. User supplied "Hacks"

There is a definite positive feedback effect from having the developers of a system rely on it for their day-to-day existence, particularly in a system as open and extensible as XDE. In the early days, when an implementor wrote a small program that enhanced his productivity there was a natural tendency to share it with his friends. Such "neat hacks" often were included in the next official release of the system. The symbiote package described in §3.7 started life this way.

We reached the point where the producers and users of this ancillary software were no longer housed in the same comer of a single building, but were spread across the continent (even a few on other continents). In this situation, a degree of order was called for. The directory **<Hacks>** on one of the file servers was changed from a

private Mesa group directory into a public one, and a set of rules was established to govern the use of this directory. Here are some of the rules from the most recent set [Johnsson83]:

- A "hack" is a useful (or semi-useful) program that is made available to the general community as a public service. No warranty of suitability or responsibility for errors is implied by the author.
- A hack stored on the **<Hacks>** directory remains the "property" of the submitter. Others may not make modifications (except for their own private use) without negotiating with the owner (who may already be making similar or incompatible modifications).
- As the owner of a hack, you are not required to fix bugs, but you must be willing to transfer ownership (permanently!) to someone who volunteers to fix them.
- Hacks that are derived from released software should be announced to the public only after discussion with the organization or individual responsible for that released software.
- Every hack stored on the **<Hacks>** directory will also have documentation and all sources necessary to rebuild it stored on the **<Hacks>** directory.
- Testing is important. A hack is not shoddy software; it is software made available outside the regular release channels.

These rules have proved quite successful; at last count there were over 200 programs stored on the **<Hacks>** directory. The programs can be roughly divided into four categories:

- Modified versions of official tools, with added functionality. These new features are often incorporated into the official versions at the next release.
- Extensions to the environment. These include adding new keyboard commands (made easier by the TIP mechanism), changing the appearance of tiny windows, and changing the semantics of scrollbars.
- Generally useful tools. For example, there is a calendar program, several graphics editors, a spreadsheet, a tool for browsing files on remote directories, etc.
- Fun and games. This includes a program to play music on the tone generator of the keyboard, several arcade-like games, and MazeWar, a multi-player seek-and-destroy game played over the network.

7. Conclusions

Reflecting on the 5 year (or 10, depending on how you count) history of XDE, I am struck by a number of things that we seemed to get right. Of course, there have been ideas that didn't work out, but first consider the successes.

One major success is a consistent user interface across tools, particularly since this was done by seduction, not by fiat. The interface building support provided by the system, e.g., the form subwindow package, has been sufficiently powerful and easy to use that programmers seldom consider implementing their own. As one who has used four similar but different interfaces at once, I can attest that it is easier to use radically different interfaces than it is to use almost identical ones.

There is also a consistency at the program level. A tool can share information with another through the selection mechanism without any knowledge of how the other tool works. The same scrollbar mechanism can scroll text subwindows, form subwindows, or graphics editor windows. There are many other examples. The hierarchies of abstraction and information hiding ability of the Mesa language were invaluable for creating this support.

Another success is a balance of novice and expert features. While it is an admirable goal to make a system easy to learn, it should not be done at the expense of making it clumsy for the expert to get work done. There are several examples of this principle. For example, many tools have both TTY style interfaces and form subwindow interfaces. The former are quick, but require the user to know the proper command syntax and switches; the latter show all the options clearly, but are more clumsy for quick use. Another example is the window manager *accelerators.* There is a menu for changing the size and position of windows, but clicking the proper mouse buttons in the black stripe at the top of the window will also invoke these operations.

It is difficult to overstate the value of multiple processes. For many tasks such as compiling, retrieving files, or printing, the user is not in that much of a hurry; he has plenty of things to do, such as editing, that don't require much processing power. When we got to Mesa 8.0, we felt that we were moving back to *personal timesharing,* which differed from classical timesharing in that all the users were the same person. The user could control the "load average" by choosing to do more or fewer things at once. While parallelism at this macroscopic level was invaluable, so was the ability to have multiple processes within a single address space. This was particularly true for communications, or animated display applications like histograms of activity.

The environment has almost always had adequate performance. This is largely a result of the evolutionary way in which it has developed. We have gone through cycles where new features are added, slowing things down in the process, followed by periods of tuning, where little functionality is added, but performance is improved.

If I had to pick a language feature that was most critical to the implementation of XDE it would be the procedure variable. Virtually all abstractions are represented by objects (records) that contain several procedure variables that implement the operations on that object. Sometimes another level of indirection is used and whole classes of objects share a single record of procedures. Inserting a private procedure that does

something special before calling the standard procedure is a technique that has been exploited many times to try out new ideas or to extend the system.

It is ironic that some of our successes have produced some of our largest shortcomings. The Alto, on which we first implemented Mesa, was a very modest machine with only 16 bits of address space and no hardware support for virtual memory. We nonetheless implemented a compiler for a large language that ran on the Alto with respectable performance. This was done by carefully tailoring the data structures and algorithms to fit into the restricted space. As machines became larger, the data structures could not gracefully grow, so the current compiler still limits program size to about 1000 lines. The Mesa language itself has not been immune from this type of premature optimization. The two classes of pointers (short and long) in the language are largely an artifact of the original Alto implementation. The treatment of OPEN as substitution by name was partly motivated by the need to live with relocating storage management schemes on the Alto.

Another example of the inability to scale is in the world swap debugger. Programs run better with more memory, but world swaps take longer. Viewed in the development/tuning cycle described above, the added functionality (more memory) has necessitated performance work (changing the debugger paradigm). There are plans for a "same world" debugger.

Another class of shortcomings could be loosely called parochialism. Much benefit has been gained by having a single language system, and when we started, almost nothing similar was available from outside anyway. As the rest of the world gets bitmap displays and mice, many interesting applications will appear written in other languages. Work is underway to add a number of common programming languages to the environment. In addition, the current program of placing this environment in a number of major universities will likely lead to other extensions.

Acknowledgements

It is impossible to acknowledge everyone who contributed to the design and implementation of XDE. Since 1978, over two dozen persons have spent time working in the Mesa group of the Xerox Office System Division. Countless others from the remainder of OSD, from PARC, and from other parts of Xerox have made valuable contributions.

Bibliography

[Birrell82] Birrell, A.D., Levin, R., Needham, R. M., and Schroeder, M. D.. Grapevine: An Exercise in Distributed Computing, *Communications of the ACM* 25 4 (April 1982) 260-274

[Feldman79] Feldman, S. I., Make- A Program for Maintaining Computer Programs, *Software Practice & Experience 9 4 (April* 1979) 255-265

[Geschke77] Geschke, C. M., Morris, J. H., and Satterthwaite, E.H., Early experience with Mesa, *Communications of the ACM* 20 8 (August 1977) 540-553.

[Graham79] Graham, S. L., Joy, W. N., and Roubine, O., Hashed Symbol Tables for Languages with Explicit Scope Control, *Proceedings of the Symposium on Compiler Construction, SIGPLAN Notices* 14 8 (August 1979) 50-57.

[Johnsson82] Johnsson, R.K. and Wick, J.D., An Overview of the Mesa Processor Architecture, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems SIGPLAN Notices* 17 4 (March 1982) 20-29.

[Johnsson83] Johnsson, R. K., *HackRules.doc*, a text file on the <Hacks> directory.

[Kay69] Kay, A. C., *The Reactive Engine* (thesis), University of Utah, Department of Computer Science, Salt Lake City (August 1969).

[Kay76] Kay, A. C. and the Learning Research Group, *Personal Dynamic Media*, Xerox Palo Alto Research Center Report SSL-76- 1 (1976).

[Knuth71] Knuth, D. E., An Empirical Study of **FORTRAN** Programs, *Software- Practice and Experience* **1 2** (1971) 105-133.

[Lampson79] Lampson, B. W. and Sproull, R. F., An Open Operating System for a Single-User Machine, *Seventh Symposium on Operating Systems Principles, Operating Systems Review* 13 5 (December 1979) 98-105.

[Lampson80] Lampson, B. W. and Redell, D.D., Experience with Processes and Monitors in Mesa, *Communications of the ACM* 23 2 (February 1980) 105-117.

[Lauer79] Lauer, H. C., Satterthwaite, E. H., The Impact of Mesa on System Design, *Proceeding of the Fourth International Conference on Software Engineering,* Munich (September 1979) 174-182

[Lewis83] Lewis, Brian T., Experience with a System for Controlling Software Versions in a Distributed Environment, *Symposium on Application and Assessment of Automated Tools for Software Development,* San Francisco (November 1983) 210-219.

[McDaniel82] McDaniel, G., The Mesa Spy: An Interactive Tool for Performance Debugging, *Performance Evaluation Review* 11 4 (Winter 1982-83) 68-76.

[Mitchell79] Mitchell, J.G., Maybury W., and Sweet, R. E., Mesa *Language Manual,* Xerox Palo Alto Research Center Report CSL-79-3 (April 1979).

[Redell79] Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., Purcell, S. C., Pilot: An Operating System for a Personal Computer, *Communications of the ACM* 23 2 (February 1980) 81-92.

[Reid83] Reid, Loretta Guarino and Karlton, Philip L., A File System Supporting Cooperation between Programs, *Operating Systems Review* 17 5 (October 1983) 20-29

[Schmidt82] Schmidt., E. E., *Controlling Large Software Development in a Distributed Environment,* Xerox Palo Alto Research Center Report CSL-82-7 (December 1982).

[Sweet82] Sweet, R. E. and Sandman, J. G., Empirical Analysis of the Mesa Instruction Set, *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems, SigPLAN Notices* 17 4 (March 1982) 158-166.

[Swinehart74] Swinehart, D. C., *Copilot: A Multiple Process Approach to Interactive Programming Systems,* Stanford Artificial Intelligence Laboratory Memo AIM-230 (July 1974).

[Thacker79] Thacker, C. P., Alto: a personal computer, in *Computer Structures: Readings and Examples*, Second Edition, Sieworek, Bell, and Newell Eds., McGraw-Hill (1981).

[Wallace80] Wallace, Donald C., *Tajo Functional Specification, Version* 6.0, Xerox internal document (October 1980).

[Zellweger84] Zellweger, Polle T., *Interactive Source-Level Debugging of Optimized Programs*, Xerox Palo Alto Research Center Report CSL-84-5 (May 1984).