Design and Implementation of the RWM Window Manager

Paul H. Eissen

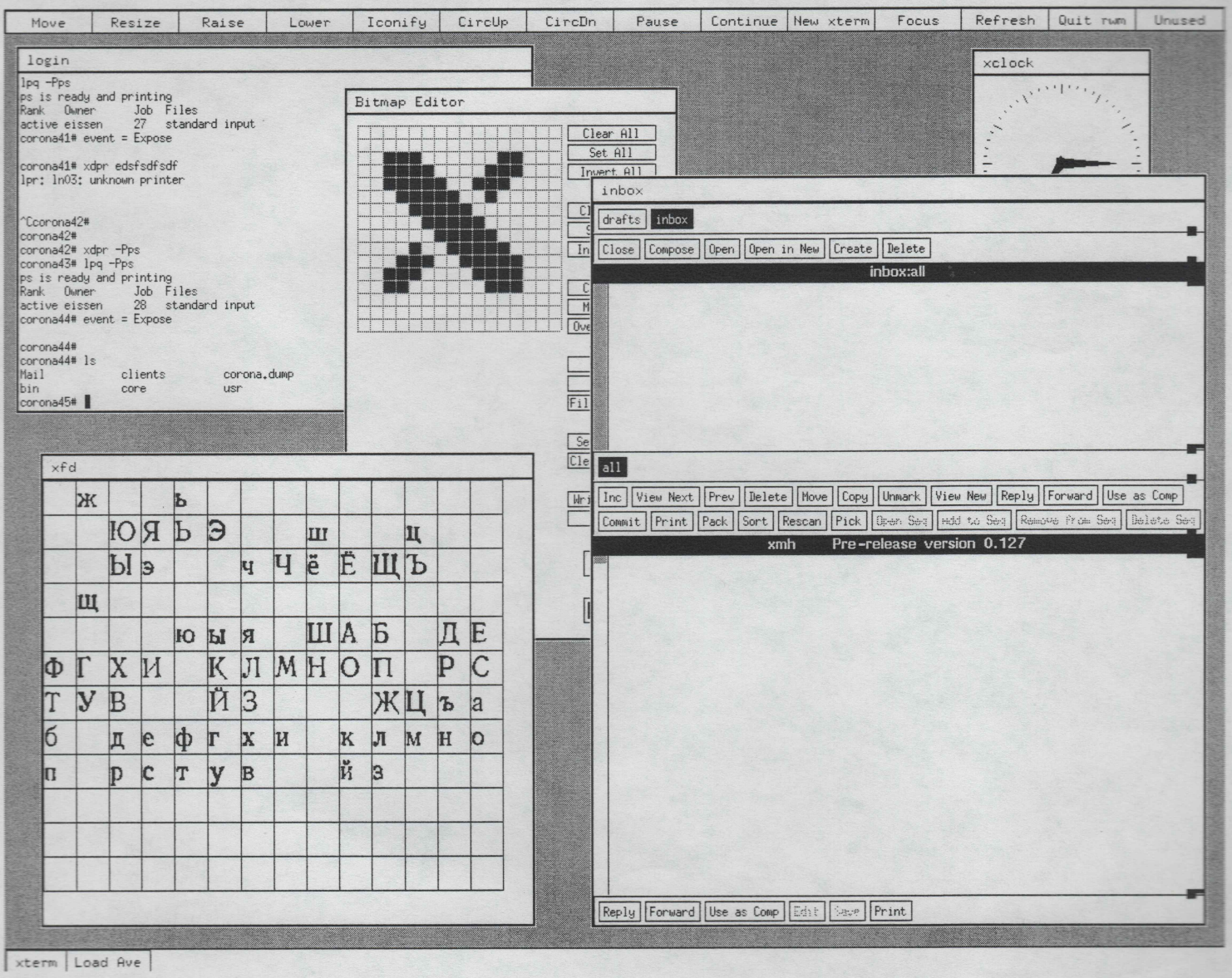
[redacted]

ABSTRACT

RWM is a menu-driven window manager for Version 11 of the X Window System. RWM manages the size and placement of overlapping client windows, provides consistent client window decoration in the form of borders and titlebars, controls the redirection of mouse and keyboard input, and handles the placement and content of icons. This paper describes the design and implementation of RWM in light of: influences from other X window managers and other windowing systems; problems implementing X window managers in the C programming language; immature client-to-window manager communication conventions; the lack of symmetry in the X Protocol; and the policy-free nature of the X Window System.

[redacted] Category: Applications of Computer Systems

September 23, 1988


Move
Resize
Raise
Lower
Iconify
CircUp
CircDn
Pause
Continue
New xterm
Focus
Refresh
Quit rwm
Unused
login
lpq -Pps
ps is ready and printing
Rank Owner Job Files
active eissen 27 standard input
corona41# event = Expose
corona41# xdpr edsfsdfsdf
lpr: ln03: unknown printer
^Ccorona42#
corona42#
corona42# xdpr -Pps
corona43# lpq -Pps
ps is ready and printing
Rank Owner Job Files
active eissen 28 standard input
corona44# event = Expose
corona44#
corona44# ls
Mail clients corona.dump
bin core usr
corona45#
Bitmap Editor
Clear All
Set All
Invert All
xclock
inbox
drafts
inbox
Close
Compose
Open
Open in New
Create
Delete
inbox:all
all
Inc
View Next
Prev
Delete
Move
Copy
Unmark
View New
Reply
Forward
Use as Comp
Commit
Print
Pack
Sort
Rescan
Pick
Open Seq
Add to Seq
Remove From Seq
Delete Seq
xmh Pre-release version 0.127
Reply
Forward
Use as Comp
Edit
Save
Print
xfd
xterm
Load Ave

Design and Implementation of the RWM Window Manager

Paul H. Eissen

[redacted]

ABSTRACT

RWM is a menu-driven window manager for Version 11 of the X Window System. RWM manages the size and placement of overlapping client windows, provides consistent client window decoration in the form of borders and titlebars, controls the redirection of mouse and keyboard input, and handles the placement and content of icons. This paper describes the design and implementation of RWM in light of: influences from other X window managers and other windowing systems; problems implementing X window managers in the C programming language; immature client-to-window manager communication conventions; the lack of symmetry in the X Protocol; and the policy-free nature of the X Window System.

1. Introduction

Windowing systems come in many flavors. Some environments, including Andrew [], Blit [], Cedar [], Macintosh [], SunView [], and Windows [], merge low-level graphics and device operations with higher-level window and input management. In contrast, the X Window System[1] leaves the construction of the user and management interfaces to the (adventurous) programmer.

This paper describes the design and implementation of RWM[2], a menu-driven window manager for Version 11 of the X Window System. The remainder of the paper is divided as follows: Section Two presents a brief overview of the X Window System; Section Three discusses characteristics of window managers common to both X and rival windowing systems; Section Four presents RWM design decisions and a tour through the implementation; and Section Five discusses of problems encountered during the design and implementation of RWM.

2. The X Window System

The architecture of X Window System ("X" for short) is based on the *client-server* model. [XWS] A *server* is a user-level process that controls a single bit-mapped display and accepts requests from, and sends input to, *client* applications. The server communicates with clients via a reliable, 8-bit duplex stream in which a simple block stream protocol, called the X Protocol [XP], is layered on top of the byte stream [XWS]. The X Protocol not only defines the base window system the server is expected to implement, but also describes how clients and servers communicate with each other. (Throughout the rest of this paper, "X Protocol" and "X" will be used interchangeably.)

---

1. The X Window System is a trademark of the Massachusetts Institute of Technology.
2. RWM does *not* stand for *R*eal *W*indow *M*anager.

Clients and servers communicate via requests, replies, events, and errors [XP]. A *request* is a command for the server to perform an operation on behalf of a client. Some requests generate one or more *replies*, in which the server synchronously sends information back to the client. A server sends information generated asynchronously to clients via *events*. An event can be generated from devices (e.g., mouse movement, key press) or as a side-effect of a client's request. An *error* is a special event generated when the server detects a problem(s) in a client's request.

Servers create and maintain *resources* on behalf of clients. X resources include windows, pixmaps, cursors, fonts, graphics contexts, and color maps. Resources are destroyed by client requests or automatically at connection close. [Xlib]

The design of X explicitly separates high-level management functions from the low-level graphics operations in the base window system. By stressing *mechanism* rather than *policy* [XWS], X allows many different application and window management interfaces to be built on top of the base window system.

The X Protocol defines a "window manager" to be a client that manipulates windows, implements most of the user interface policy, and controls the keyboard input. The X Protocol limits the number of window manager clients communicating with a server at any point in time to *one*.

X defines an arbitrarily branching hierarchy of overlapping rectangular windows [XWS]. Every window, with the exception of the *root* window, has exactly one parent. X allows windows to be *reparented* to other windows. A "top-level" or "client" window is a window whose parent is the root window. Client applications create and maintain the contents of client windows; window manager clients restack, resize, and reposition them [XWS].

A *property* is a collection of named typed data. [Xlib] A client can send "hints" to a window manager by storing properties on its top-level window(s). A window manager in turn may process selected properties and ignore the rest. The X Protocol defines a set of properties but does not impose semantics.

3. Characteristics of Window Managers

The majority of window managers for bitmapped displays, regardless of method of construction, functional description, or position in the system architecture, share certain characteristics. These include window decoration, window manager functions, function selection styles, window layout policies, icon support, and keyboard input management.

Window Decoration

Many window managers add "decoration" to application windows. The WM [], DXWM [], and Andrew [] window managers, for instance, place a *title bar* above every client window. (Title bars normally run the entire length of the window). Title bars are used by window managers to display temporal and permanent information, including names and/or symbols for applications, files, hosts, menus, actions, etc.

Window managers may also add decoration in the form of *borders*. Macintosh [] and Windows [] incorporate *scrollbars* into their window borders. Scrollbars allow a user to view the hidden regions of a window's contents. Other window managers simply fill in borders with solid colors.

A few window managers, like UWM [], avoid window decoration completely, leaving it up to the application.

Window Manager Functions

Window manager functions fall into two categories: those which manipulate application windows, and those that do not. Functions exist to move, resize, select, raise, lower, circulate, iconify, de-iconify, shrink, grow, refresh, assign the keyboard to, and destroy an application window. Non-window operations include refreshing and retiling the screen, and exiting from

the window manager. User interface policy dictates the set of operations supported by window managers.

Function Selection Styles

Users communicate their orders to window managers through a variety of interfaces. Function selection styles include static menus, popup menus, title bars, and special mouse-key combinations. The XNWM [] window manager draws a static menu across the top of the screen and waits for the user to "click" the mouse in the menu window containing the name of the operation to be selected. UWM, in contrast, displays a popup menu containing a list of function names only when the user clicks the mouse in the root window: the menu is invisible otherwise.

Many window managers represent operations as symbols in title bars: to move an application window with WM, for instance, a user clicks on the "move" symbol in the window's title bar and drags a "bounding box" of the window until satisfied with the new location. Window managers may also include menus in their title bars.

Special mouse-key combinations in certain contexts can be used to communicate choices to window managers in lieu of more cumbersome methods. A unique combination of mouse clicks and key presses, for instance, may substitute for the selection of a certain menu operation. This method can provide for a range of interfaces in the same window manager to satisfy both "novice" and "expert" users. [XWS] [Tiet]

Window Layout Policies

Window managers are either *manual* or *automatic*. [XWS] Manual window managers override very few user actions concerning the placement, sizing, and stacking of client windows. Automatic window managers, on the other hand, operate with little or no human interaction.

Window layout algorithms generally fall into one of two categories: *overlapping* or *tiling*. [Teit] Overlapping window managers implement the so-called "desktop" metaphor, in which windows can stack on top of and obscure other windows. Tiling window managers allow no overlapping by placing windows adjacent to other windows. Some tiling window managers fill the entire screen with windows while others allow blank space to exist. Examples of tiling managers include RTL [], Andrew, Cedar, and Windows.

Generally, overlapping window managers are manual, while tiling window managers are automatic. It is possible, however, for an overlapping window manager to incorporate automatic operations and for a tiling window manager to accept user input.

Icon Support

Icons are small graphical symbols representing actions, states, windows, and other objects of interest in the user interface. The Macintosh, for example, uses icons to represent such concepts as "the system is busy and you must wait" [HFactor]. A user of the SunView environment can reclaim screen real-estate by "closing" a window and replacing it with a smaller icon. [SSI] The Andrew window manager simulates the use of icons by "shrinking" an application window to its title bar [].

Many window managers restrict icon placement. Both Windows and Cedar, for instance, reserve the bottom of the screen for icons. An interesting variation on this theme is the "icon box" window used by DXWM to store icons []. At the opposite extreme, UWM maintains a *laissez faire* attitute toward icon placement.

Keyboard Input Management

Many window managers are responsible for the management of keyboard input. (in X, this is referred to as "setting the input focus".) The two basic models for keyboard management are *real-estate* and *listener*. [XWS] A real-estate window manager directs keystrokes to the application whose window currently holds the mouse. A listener window manager forces a user

to "select" a window (via a mouse click, perhaps) before directing keystokes to the window. A handful of window managers support both models. UWM is capable of switching between real-estate and listener modes at the direction of the user.

4. RWM Design and Implementation

To the software engineer, design is the bridge between software requirements and an implementation that satisfies those requirements. [RF] Ideally, the design process should specify the structure of a software system independent of the programming language. In theory, it is *possible* to design an X client using only the X Protocol as a guide; in practice, programming interfaces to the X Protocol permit a programmer to "intertwine" the design and implementation phases to facilitate the rapid creation of prototypes.

Although an initial requirements list [] was generated at the beginning of the development cycle, RWM is actually the result of countless prototypes. For this reason, both the design decisions and the implementation details are presented together in this section.

A List of Requirements

Several of these requirements are documented in []; many were formulated as a result of prototype evaluation.

- A static "Menu" window shall be painted at the top of the screen. The Menu shall be completely unobscured at all times. Menu operations shall be selected with the mouse.
- RWM shall operate on the set of client windows.
- Client windows shall be decorated with "Frame" windows, "TitleBar" windows, and borders.
- RWM shall associate an "Icon" window with every client window and maintain the set of Icons.
- All Icons shall be kept in the "IconBox" window. RWM shall unmap the IconBox when the last Icon is unmapped and map it along with the first mapped Icon. When mapped, the IconBox shall be painted at the bottom of the screen and kept completely unobscured and static. Icons shall be selected with the mouse.
- The selection of an Icon with the mouse shall unmap the Icon and map its client window.
- For consistency of appearance, the Menu, the IconBox, and TitleBars shall use the same font.
- RWM shall support both real-estate and listener models of keyboard management (input focus).
- RWM shall support overlapping client windows.
- The user shall be allowed to manipulate client windows at will, subject to constraints placed on these windows by the applications themselves. In other words, RWM shall emulate manual window management with respect to the set of client windows.
- RWM shall emulate automatic window management with respect to the Menu, the IconBox, and the Icons.
- RWM shall honor the following client "hints": client window name; client window size and location; icon name; transient state; maximum, minimum, and incremental client window sizes; client window state; and input focus model.
- RWM shall implement the following "selection" Menu operations:
  - *Move* - select a client window and move it to a new location.
  - *Resize* - select a client window and resize it.
  - *Raise* - select a client window and raise it above all other siblings.
  - *Lower* - select a client window and lower it below all other siblings.

- *Iconify* - select a client window, unmap it, and map its Icon into the IconBox.
- *Focus* - select a client window and direct all subsequent keyboard input to it (listener mode), or select the root window or Menu and reset RWM to real-estate mode.

- RWM will implement the following "non-selection" Menu operations:
  - *Circulate Up* - raise all client windows obscured by siblings.
  - *Circulate Down* - lower all client windows that obscure siblings.
  - *Pause* - stop all output to the screen; used with *Continue.*
  - *Continue* - release the screen; used with *Pause.*
  - *Refresh* - send exposure events to all client applications.
  - *Quit* - perform a graceful exit.

Programming Details

Xlib [], a C [] subroutine library, is the best-known and most-robust X Protocol programming interface. For ease of binding with Xlib, RWM was implemented in C.

Abstraction Models

The use of procedure and data abstraction [RF] simplified the design and subsequent debugging of RWM. Procedure abstraction was used to divide the high-level structure of RWM into three "packages"[3]: Menu and functions, Frames and TitleBars, and IconBox and Icons. Although numerous prototypes were created, RWM's basic three-package structure never changed.

As the design progressed, it became apparent that some kind of "list" was needed by RWM to keep track of three sets of state information: client windows, Frames, and TitleBars; client windows and Icons; and currently-mapped Icons. A utility package containing a generic C-language doubly-linked-list *abstract data type* [SRC] was created.[4] The Frame and Icon packages made use of this package so as to separate information processing from storage and retrieval details and in turn simplified their own implementations.

RWM As Client

In X, a window manager, however specialized, is just another client. An Xlib client is divided into *initialization* and *event handling* [HW] [Jones].

5. Problems

The design and implementation of RWM was not without problems. While some of RWM's flaws can no doubt be attributed to the author's inexperience, asymmetry in the X Protocol, the policy-free nature of X, and programming with C and Xlib complicated the development of RWM.

---

3. A C package is nothing more than a pair of files: the ".h" file contains function declarations, and the ".c" file contains function and static variable definitions.

4. This C-based abstract data type is a reseach topic unto itself and any discussion of it is regrettably beyond the scope of this paper.

Asymmetry in the X Protocol

In the "Commentary on X Version 11 Design" [], Gettys noted that

> A window manager... interested in controlling placement of subwindows... can select structure redirect control. Whenever there is an attempt to map, unmap, destroy, reposition, resize, [or] alter [the] border of a (sub)window the selecting client will be notified, and the operation ignored. A window manager can then perform the operation on behalf of the clients.

In reality, the X Protocol lets a window manager "steal" MapWindow, ConfigureWindow, and CirculateWindow requests from other clients but does not provide for the redirection of UnmapWindow and DestroyWindow requests. This asymmetry in the design of the X Protocol forces a window manager to react to the unmapping and destruction of client windows *after the fact.* While a well-behaved window manager, if given the chance to capture UnmapWindow and DestroyWindow requests, will simply pass the requests unchanged, the ability to capture all window "change of state" requests from other clients makes for a cleaner and more flexible window manager design.

Since it intercepts MapWindow requests, RWM does not have to distinguish between MapNotify events generated from its own MapWindow requests and those generated by requests from other clients; in fact, RWM ignores all MapNotify events.

On the other hand, when a client issues an UnmapWindow request, RWM has to wait for the server to send the UnmapNotify event. An additional complication is that several of RWM's operations issue UnmapWindow requests. Therefore, not only must RWM react to the unmapping of client windows after the requests have been processed, it has to decide which UnmapNotify events to ignore (its own) and which to process (clients'). When RWM calls mapFrame(), the is_mapped field associated with the newly-mapped client window is set to True; when the window manager calls unmapFrame(), this field is set to False. Therefore, to determine the originator of an UnmapNotify event, RWM checks the current value of the is_mapped field:

```
case UnmapNotify:
   if (clientInFrameList(event-> xunmap.window))
   {
       if (frameIsMapped(event-> xunmap.window))
       {
          unmapFrame(dpy, event-> xunmap.window);
          ...
       }
   }
   break;
```

Unfortunately, this solution contains a flaw: if RWM were modified to receive SubstructureNotify events on the root window, RWM would inadvertently *unmap* every mapped client window reparented during the initialization phase. A side effect of reparenting is the implicit unmapping by the server of currently-mapped client windows: when it receives an UnmapNotify event, RWM will mistakenly conclude *another* client generated the event and unmap the window.

The inability to intercept UnmapWindow requests resulted in unbalanced (and flawed) UnmapNotify event handling in RWM.

The Policy-Free Nature Of X

policy-free - too many choices, few restrictions, almost no restrictions on clients (since they should be written w/out ANY window manager in mind), (WM can't implement true user interface because clients have control over their contents); anti-social behavior hard to defend

against (See XWS, section 3.2 re guessing resource IDs), immature client-window manager comm. conventions;

The Xlib Programming Interface

C and Xlib, window managers could use object-oriented and message-passing concepts effectively, C+ + , event-driven (changes in software propagates to other parts, no matter how hard one tries to separate frame, icon, event handling, etc. code)

6. Conclusions

References

[1] Paul Asente, "xnwm - X window system manager process", X Window System, Version 10, 1986.

[2] Ellis Cohen, "The Siemens RTL Tiled Window Manager", *2nd MIT X Conference Notes*, Jan. 1988.

[3] *DECWindows User Interface Style Guide*, Digital Equipment Corporation, 1988.

[4] Paul Eissen, *Requirements for Raoul's Window Manager*, Internal Memorandum, 1988.

[5] Richard Fairley, *Software Engineering Concepts*, McGraw-Hill, 1985.

[6] Hania Gajewska, "Window Managers in X11", *2nd MIT X Conference Notes*, Jan. 1988.

[7] Hania Gajewska and David S. H. Rosenthal, "wm - a simple real-estate-driven window manager", X Window System, Version 11, 1988.

[8] Michael Gancarz, "uwm - a window manager for X", X Window System, Version 11, 1988.

[9] Jim Gettys, "Commentary on X Version 11 Design", X Window System, Version 11, 1986.

[10] Jim Gettys, Ron Newman, and Robert W. Scheifler, "Xlib - C Language X Interface", X Window System, Version 11, 1988.

[11] *Inside Macintosh*, Addison-Wesley, 1985.

[12] Oliver Jones, "Introduction to Programming the X Window System", Apollo Computer Inc., 1987.

[13] Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[14] Ed Lee, "Window of Opportunity", *UNIX Review*, Vol. 6, No. 6, June 1988, pp. 47-61.

[15] *Microsoft Windows User's Guide*, Microsoft Corporation, 1985.

[16] James H. Morris, Mahadev Satyanarayanan, Michael H. Connor, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith, "Andrew: A Distributed Personal Computing Environment", *Communications of the ACM*, Vol. 29, No. 3, Mar. 1986, pp. 184-201.

[17] Rob Pike, "The Blit: A Multiplexed Graphics Terminal", *AT&T Bell Laboratories Technical Journal*, Vol. 63, No. 8, Part 2, Oct. 1984, pp.1607-1631.

[18] David S. H. Rosenthal, "A Simple X11 Client Program, or, How hard can it really be to Write 'Hello, World'?", *USENIX Conference Procceedings*, Winter 1988, pp.229-235.

[19] David S. H. Rosenthal, "Inter-Client Communications Conventions Manual", X Window System, Version 11, Draft, Feb. 25, 1988.

[20] Richard Rubinstein and Harry Hersh, *The Human Factor*, Digital Press, 1984.

[21] Robert W. Scheifler and Jim Gettys, "The X Window System", *ACM Transactions on Graphics*, Vol. 5, No. 2, Apr. 1986, pp. 79-109.

[22] Robert W. Scheifler, "X Window System Protocol, Version 11", X Window System, Version 11, Sept. 1987.

[23] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

[24] *Sun System Introduction*, Sun Microsystems, Inc., 1987.

[25] *SunView 1 Programmer's Guide*, Sun Microsystems, Inc., 1987.

[26] Warren Teitelman, "A Tour Through Cedar", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, Mar. 1985, pp. 285-302.

[27] Robert A. Zimmermann, "The C-Programmer's Toolbox A Set of Packages for C Programs", SRC Technical Report T86081, Semiconductor Research Corporation, June 1986.