A PORTABLE USER INTERFACE FOR A SCIENTIFIC PROGRAMMING ENVIRONMENT

Vincent A. Guarna, Jr.
Yogesh Gaur

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract

The subject of integrated programming environments for scientific computing has become very popular over the last few years. Environments such as R^n [CCHK87] are being constructed to help coordinate the disjoint activities of editing, debugging, and performance tuning typically seen in the program development cycle. One key aspect of an integrated development setting is the library of user interface tools which are available to the environment builders. Projects such as Andrew [MSCH86] have begun to construct reusable user interface libraries for client applications. This paper describes the interface tool kit for the Faust project being conducted at the University of Illinois. Faust is targeted at building a coherent development environment for scientific applications through the use of a library of portable user interface utilities.

1. INTRODUCTION

This paper describes the user interface being constructed for the Faust project.[1] Faust is a workstation-based programming environment for scientific applications currently being developed at the Center for Supercomputing Research and Development at the University of Illinois at Urbana-Champaign.[2] The primary focus of Faust is to provide supercomputer users with a set of tools that make the task of programming parallel machines easier. However, many of the problems encountered in the construction of such an environment are not specific to parallel programming. In particular, issues such as functional tool integration and uniformity of user interface are very basic problems which any computing environment must solve in order to be effective.

[1] Faust is funded by the Air Force Office for Scientific Research under grant number AFOSR-F49620-86-C-0136.

[2] As with many projects, Faust was named with no underlying acronym or rationale. However, it has been suggested that completing the project will require a deal with the devil.

1.1. Faust Goals

The Faust project is directed toward several goals. The first goal is the design of a consistent user interface that can be used throughout all aspects of the program development cycle. This includes the development of a program abstraction mechanism that is useful not only during original program editing phases but through debugging and performance tuning phases as well. The model that was chosen is a combination of graphical and textual representations for application programs. The textual source code presentation serves as the user's detailed view of an application whereas hierarchical graphs serve as higher-level views that can be automatically generated to show inter-subroutine and inter-task relationships.

A second goal is portability. Although the Faust environment assumes the existence of a bitmapped workstation running Unix[3] as a basic platform, Faust is expected to run on a variety of hardware. In order to accomplish this goal, all user interface libraries have been layered on top of the X Windows System developed at MIT [GeND87, ScGe86]. X Windows is a network transparent windowing system which runs under 4.3 BSD Unix, Ultrix-32, VAX/VMS, as well as many other operating systems. By relying on X for primitive interactive I/O support, the Faust environment can migrate to additional machines as X implementations become available.

A third goal to be accomplished is non-intrusiveness of the environment; Faust should not interfere with "conventional" Unix operations. Faust is not considered to be a "sealed" system -- users may wish to do some operations within the context of Faust while doing other operations with traditional shell commands. One main objective of the Faust project is the development of a user interface strategy that allows both modes of operation to coexist.

1.2. Architecture

The organization of the Faust environment is shown in Figure 1. The lines between boxes indicate paths of communications between system elements. As can be seen, program development tools such as compilers and editors can directly access the Unix file system. However, additional paths to the Faust support libraries have been created in order to facilitate the addition of a unified user interface that

[3] Unix is a trademark of AT&T Bell Laboratories.

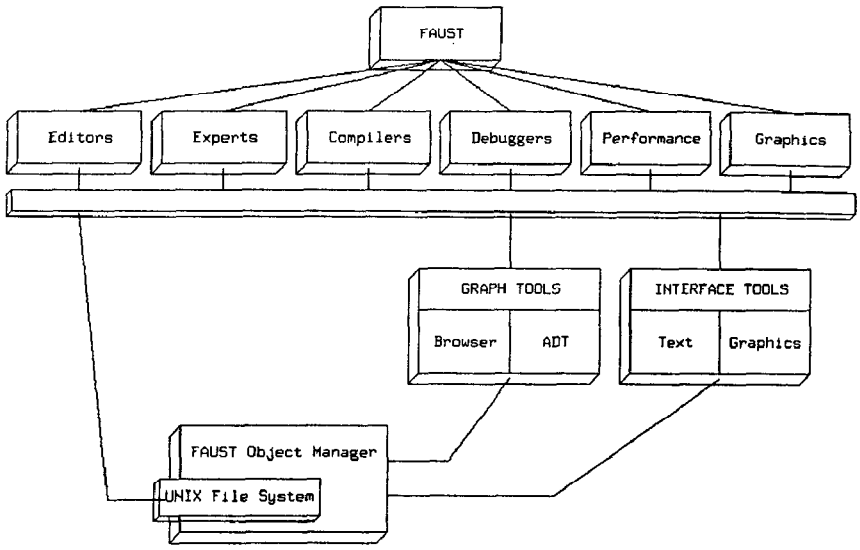

Figure 1. Architecture of the Faust environment

can be supported by all applications. The user interface functions comprise a library of routines for handling low-level functions such as displaying graphic and text primitives. However, it also contains the mechanics for the maintenance of a hierarchical program abstraction. This abstraction allows the user to view application programs in varying levels of detail, from a textual view of source code to a high-level graphical view of function and task relationships. The following sections describe the details of the user interface library and the hierarchical program abstraction as well as some examples of development tools that use them.

2. USER INTERFACE LIBRARY

The Faust user interface library comprises several components, including libraries for managing windows, user input, text objects, graphic objects, and application program objects. These libraries are collectively referred to as the Interface Manager (IM), and are described in the following sections.

2.1. Window Management

In any windowing environment, a user expects to have complete control of the images on his workstation screen. Faust makes minimal assumptions regarding the size and location of existing windows when new windows are created. Additionally, for simplicity, Faust windows can be manipulated by the window managers[4] supplied with X Windows so that users who have familiarity with X realize a smoother transition to the Faust environment.

A Faust window is composed of a maximum of three components: a title bar, a graphics subwindow, and a conversational subwindow (see Figure 2).[5] The title bar is used for identification and shows the window name and a list of pull-down menu options. The graphics subwindow is used to display graphics and combined text/graphics output. The text window may contain only textual information and is used primarily for conversational operations such as editing and command execution.

The proportion of the Faust window that is dedicated to the text subwindow can be defined by the client application during window creation. The percentage of the window height that is allocated for conversational (textual) operations can be specified as any integer in the range of 0 to 100 per cent. Windows specified with zero percent are opened as purely graphic; windows specified as 100 per cent are opened as purely textual. Figure 2 shows a Faust window opened with a height ratio of 50 per cent.

When opened, Faust subwindows are allocated as children of a single enclosing X window. This allows the Faust window handling subsystem to be compatible with existing window managers. Events related to the outer window are intercepted by the Interface Manager and translated to the appropriate resizing and refreshing operations needed within the constituent Faust windows.

[4] Such as **uwm**

[5] The terms "conversational subwindow" and "text subwindow" are used interchangeably throughout this paper.

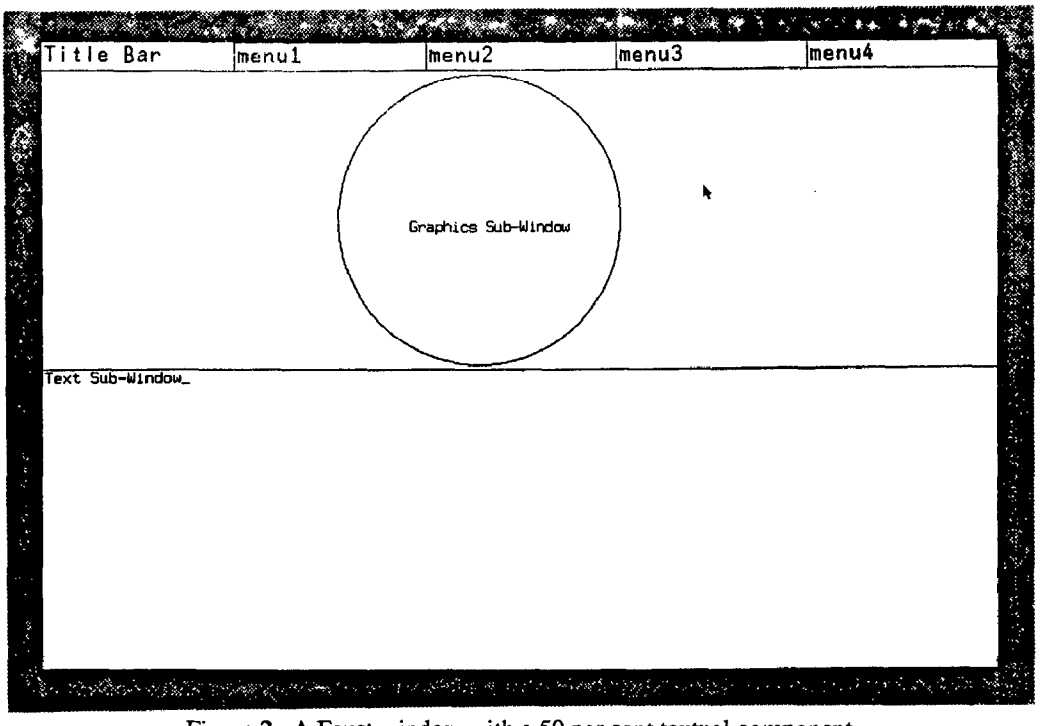

Figure 2. A Faust window with a 50 per cent textual component

It is sometimes desirable to treat a separate collection of windows as one collective unit. For example, Faust supports a performance analysis tool that shows the dynamic execution of eight processors in ten individual Faust windows [GJMY87] (see Figure 3). Since these windows represent a single user application, it is convenient for the user to treat them as a single entity. When one window is raised or brought up, all windows should be raised. When one window is moved, others should also be moved. One solution to this problem is to create a root window with ten children windows; however, this places the burden of window management on the application. To solve this problem, Faust allows applications to specify a group of windows that are to be treated as a single group for the purpose of window movement, raising, and lowering.

2.2. Text and Graph Operations

The Interface Manager contains a collection of routines for maintaining text objects in the system. The text handling subsystem provides an abstract data type for textual objects, including edit, search, and display functions. The rationale for including such a package in the Faust library is to promote reusability of basic functions to reduce development time of higher-level tools.

The Interface Manager also contains a collection of routines for maintaining graphic objects in the system. In addition to supporting a set of graphic primitives for drawing simple figures such as lines, circles, and boxes, the IM provides an extensive set of utilities for operating specific structures called *program graphs*. Program graphs and associated operations are described in detail in Sections 3 and 4.

2.3. Window Refreshing

The X Window server does not take responsibility for refreshing the contents of windows.[6] When all or part of a window is hidden and subsequently uncovered, the client program is notified by an "exposure event" that the window contents must be refreshed. Therefore, all client programs must be able to regenerate the contents of windows on demand. Although Version 11 of X does provide support for refreshing by storing "pixmaps" on disk, many environments could have limited amounts of off-screen memory, making it desirable to minimize the number of bit images that are stored in pixel format. Additionally, the quantity of data involved in storing and retrieving pixel images could make system response time too slow on diskless nodes in network configurations.

For most Faust applications, all text and graphic objects are stored in display lists that are automatically maintained by the Interface Manager. Maintaining a display list offers several advantages to the Faust client application.

First, the refreshing operation can be handled automatically by the IM without requiring interaction with the application.[7] Exposure events that are generated by X are trapped by the IM input handler and result in calls to the appropriate primitives to restore the window image.

Second, Faust client applications can generate window images in the larger X Window world coordinate space. By storing the entire image (both visible and invisible) on the

[6] For X Windows Version 10, Release 4.

[7] Although clients can optionally request to be notified of exposure events.

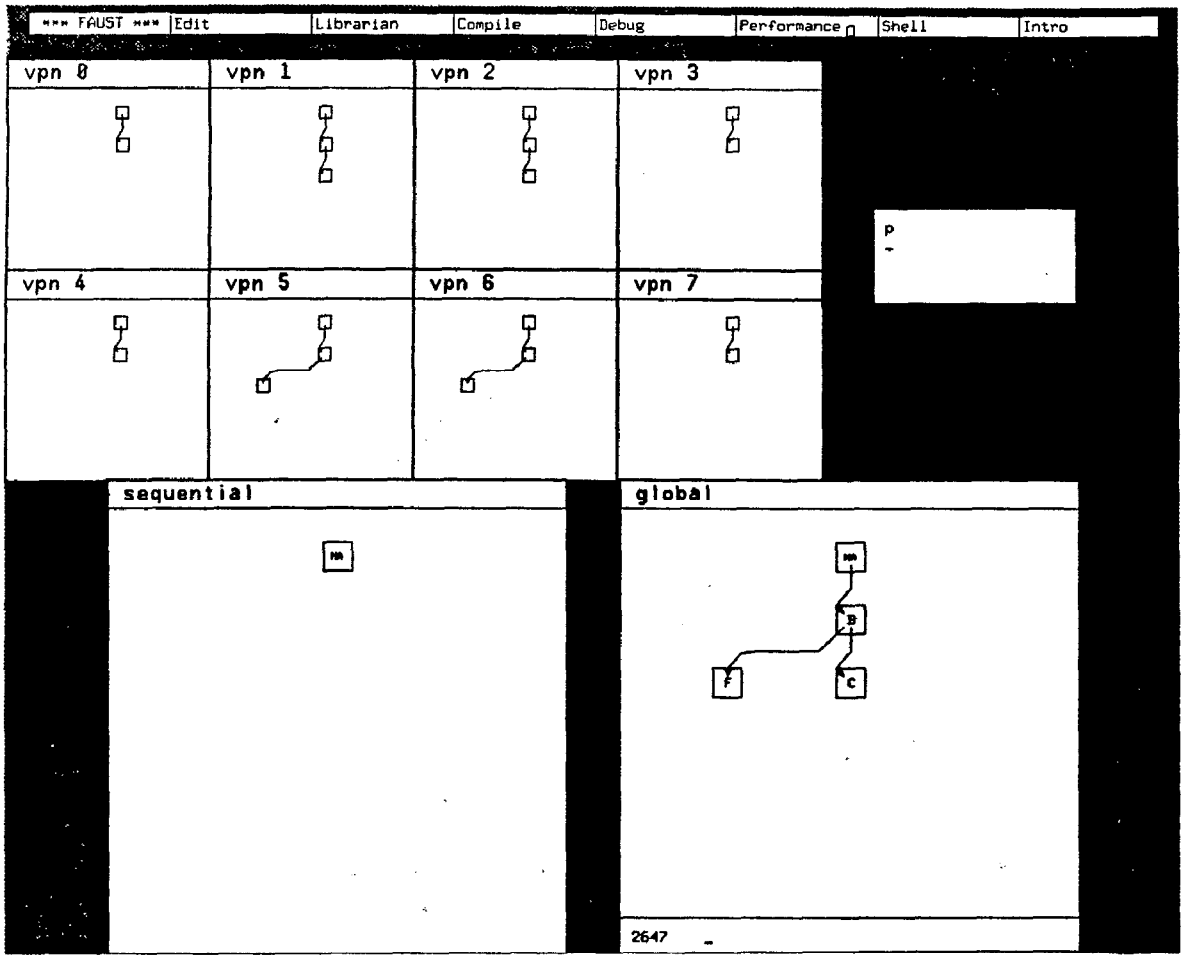

Figure 3. Dynamic Call Graph Monitoring Tool

display list, the Faust IM can support the automatic handling of panning operations as well as window refreshing. Additionally, since all objects are stored internally in terms of world coordinates, the IM can automatically support magnification and demagnification of drawn images.

All of these operations are handled internally by the Interface Manager without intervention from the client application by trapping a reserved set of inputs that are common across all windows.

2.4. Input Handling

The X server distributes user input to and accepts output requests from various client programs located either on the same host or remotely on the network. Input from X is presented to the client application in the form of "events." Input events may result either from direct asynchronous actions such as key presses or mouse clicks, or may happen as a side effect of another operation such as window resizing (which generates a window exposure event). The Interface manager handles input event processing on behalf of the application. A specific set of core input events are intercepted by the IM and processed automatically.

These input events include:

Magnify/demagnify window contents
Pan left/right/up/down
Exposure (refresh)
Zoom/Unzoom

The magnify/demagnify inputs are generated by the user and allow him to control the size of the contents in a window. In a graphics subwindow, all graphical objects are either enlarged or reduced, according to the user's input, and the window contents are redisplayed in the new configuration. In textual windows, the text font is automatically changed in order to accommodate the requested operation.[8]

The "pan window" option is also user-initiated and allows the user to move the window viewport through the X world coordinate space. This option operates similarly in the graphic and textual windows. Text that scrolls off the "top" of the window is saved by the IM for later review, similar to

[8] In textual windows, the value of this option is admittedly questionable because of the limited number of font sizes available through X. However, for the sake of orthogonality and text visibility for demonstrations, the feature is supported.

the operation of *pads* in the Apollo Display Manager [Apol85].

The refresh event is generated by the X server and instructs the IM to reconstruct any images that were located in the affected windows.

The zoom/unzoom inputs are generated when users peruse application programs. These operations are described in detail in Section 3.

Upon recognition of these events by the input handling subsystem, the appropriate function is automatically invoked without intervention by the client application. Only those inputs that are not understood as being "reserved" by the input handler are passed back to the caller.

The IM supports both blocking and non-blocking input calls from applications. The usual mode of operation is to have the application issue a blocking input call, returning only when unreserved input events are received by the IM. The primary purpose for non-blocking input is to support applications that perform dynamic or real-time output operations that continuously update displays while checking for input. For such applications, the client must poll the input service routine with sufficient frequency to ensure good user response time.

2.4.1. User-definable IDs and Input Selections

The Interface Manager provides a uniform manner for an application to receive user selections on textual and graphic objects. Every output primitive supported by the IM allows the application to associate an integer ID. Text strings as well as graphic primitives can be associated with an application-supplied ID that becomes permanently attached to the object on the display list.[9]

Selections in Faust are performed by clicking the left mouse button on a textual or graphic object. Upon recognizing an object selection, the user-defined ID is returned along with other information pertaining to the object such as type and window-relative and world coordinates for the selection. By doing ID associations with primitive elements, the IM is able to assist application programs by mapping window-relative x-y coordinates into specific objects. Attaching the ID to the objects in the display list also provides a simple means of maintaining the mapping information regardless of changes made to the view by the user.

2.4.2. The Sigma Editor

An example of a program development tool that uses the input selection features of the interface library is the Sigma editor developed at Indiana University [GALS87].[10] Sigma is a language-dependent interactive restructuring editor used by programmers to manually perform source-to-source transformations on programs for the purpose of optimization for parallel machines.

Sigma performs all of its operations by working from an internal parsed version of the user's program. The interface with the user operates by unparsing text into Faust conversational windows and taking inputs in the form of mouse clicks and pull-down menu choices. To support the remapping of unparsed text into internal parse tree locations, Sigma's unparser associates an integer ID with each output token string. This integer represents an index into a table of parse tree pointers.[11] By associating IDs with text strings in this manner, the Faust input handler can deliver specific parse tree references from a text window without requiring any interaction with Sigma.

3. HIERARCHICAL PROGRAM ABSTRACTIONS

The Faust environment supports a representation for programs that is graphical as well as textual. Although the conventional source text model is still available to the programmer, higher level abstractions such as static subroutine call graphs and task/process graphs are useful in understanding overall program structure more readily. Just as graphic images can be used to render a concise representation of large volumes of output data, graphic program structures can be a useful tool for the programmers wishing to elide much of the source-level implementation details in favor of perusing a terse representation of an application's architecture.

The Faust program abstraction supports three levels of detail. These are the process graph, the subroutine interconnection graph, and the program source code. The Faust Interface Manager provides the facility to traverse levels of the hierarchy with the "zoom" and "unzoom" commands. Zoom and unzoom are reserved input events trapped by the input handler in program application windows. These operations are initiated by the user and the response is handled by the IM, transparent to the client application program. Each level of the abstraction is described in more detail in the next three sections.

3.1. Process Graphs

The highest level of the program abstraction is the process graph. Each node in the process graph represents an individual process or task - a parallel running entity (see Figure 4). Nodes are created by issuing a "fork" style call in the target program. Directed arcs that connect these nodes represent the direction in which the fork was initiated and the data that was passed to the spawned task at creation.

3.2. Subroutine Interconnection Graphs

Performing a zoom operation on one of the process graph nodes results in the creation of a new window containing the subroutine interconnection graph (SIG) of that process.[12] Each node in the SIG represents a subroutine or hierarchy of subroutines in the target program. This structure is static and represents the inter-module binding.

---

[9] Although the IM supports a set of calls to change ID numbers.

[10] Sigma is described in [GALS87] as "Bled" for "Blaze Editor," after the programming language, Blaze, on which it originally operated. Since that time it has been extended for Fortran 8X [Metc87] and C, and has been renamed as Sigma.

[11] Integer indices are used rather than pointers in order to avoid problems resulting from discrepancies between pointer and integer types on some machines.

[12] The words "process" and "task" are used somewhat interchangeably in this section. They are meant to represent a single thread of processor execution, not necessarily the traditional Unix process.

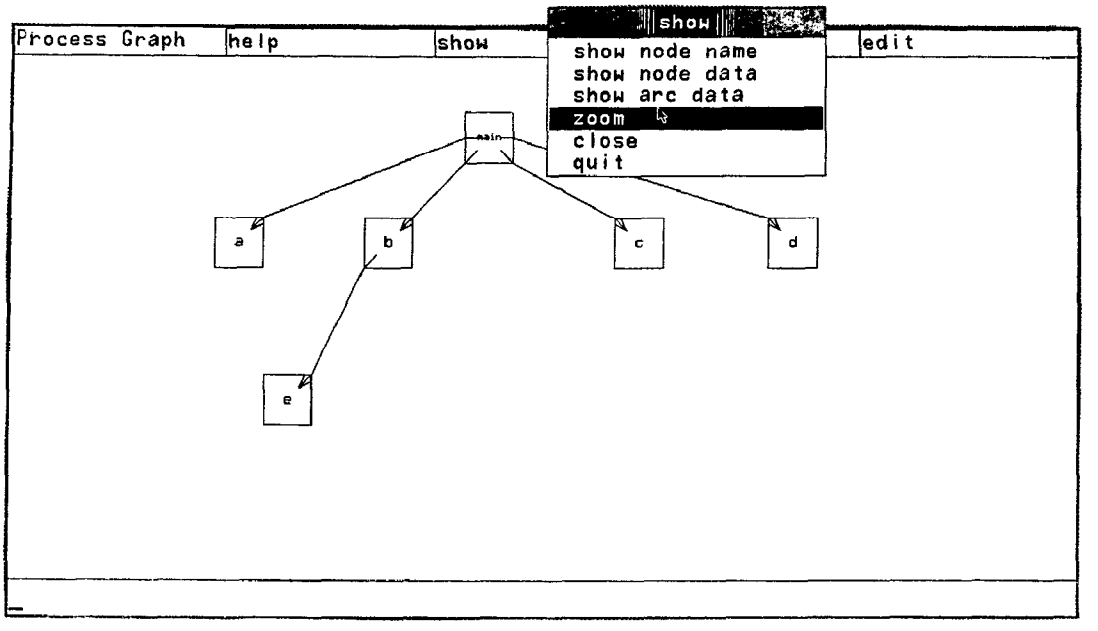

Figure 4. Process graph

The directed arcs in the graph represent function or subroutine calls that are issued and are satisfied by the end-point nodes. They also represent the flow of data that is passed between nodes when the control is transferred. Programs containing several system calls quickly become cluttered and difficult to view. For that reason, the SIG display functions reference a user-defined list of "uninteresting" functions that are not displayed. The user can also interactively specify portions of the tree to hide so that a small number of interesting nodes can be viewed.

3.3. Source Windows

Zooming on a SIG node results in the creation of a new window that contains the source statements that comprise that node. The program's sequential source code is the finest level of detail in the program abstraction. At this level the programmer may review and modify the constituent source code for a single subroutine using his favorite text editor. Changes made to the source that result in changes to the associated subroutine interconnection graph are "remembered" by the Interface Manager and cause the regeneration of the SIG the next time the graph is to be displayed. Figure 5 shows some example Fortran source code and the associate subroutine interconnection graph.

The "unzoom" function works exactly in reverse to zoom. Unzoom operations from a particular level of detail result in the creation of a new window containing the next higher level of detail. In order to support the zoom and unzoom functions, the Interface Manager maintains a list of associations between source files and graph structures. Consistency checks are made when zoom and unzoom operations are requested by the user in order to guarantee the display of consistent information. New graph structures are rebuilt only when time-stamp information indicates that it is necessary.

4. GRAPH RESTRUCTURING OPERATIONS

Many of the tools being built for the Faust environment require graph restructuring operations. The Interface Manager performs several graph editing and query operations on behalf of client applications. Processor time needed to do these internal operations is taken when applications pass control to IM through the input functions.

The Interface Manager bases many of its operations on facilities that are provided by a graph handling subsystem [JaGu88]. The graph handling subsystem comprises two tools, the Graph Manager and the Graph Browser, as shown in Figure 1. The Graph Manager is an abstract data type for graphs and manages all internal operations with respect to graph data structures without regard to mapping these data structures to the screen. The Graph browser can optionally be used to fetch graph data structures and compute the geometry placement of the nodes. The Interface Manager interacts with both tools in order to provide a graph edit/review facility to client applications.

When graph structures are displayed in a Faust window, the IM provides a skeleton structure for graph operations. Applications can access a set of standard operations that they can either use, redefine, or enhance to suit their needs.

Graph operations are divided into three categories:

(1) Functions that allow no changes to a graph. These operations treat graphs and their screen representations as read-only entities. The Faust user is allowed only to query the graph structure in order to view more detailed information that is contained within nodes and arcs.

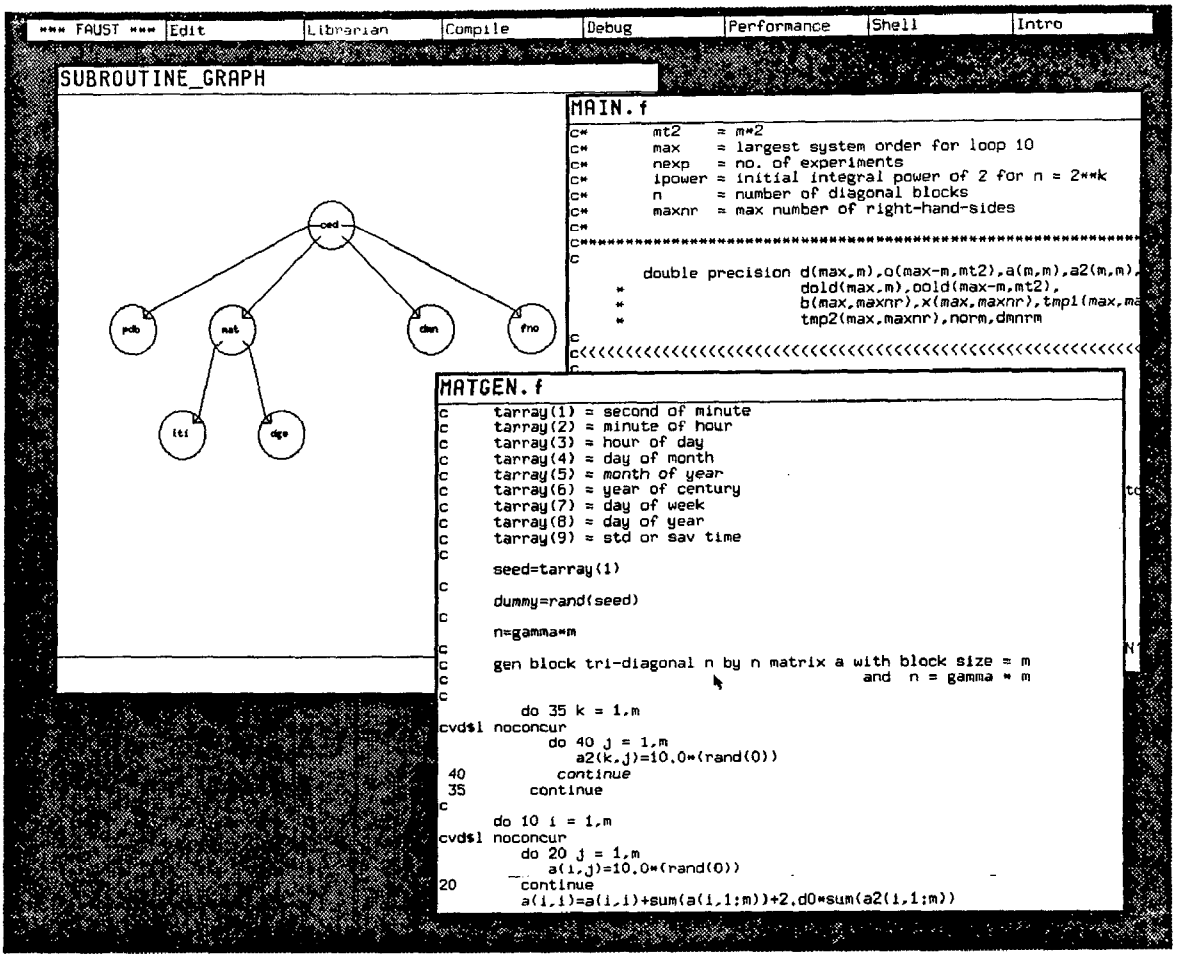

Figure 5. Fortran source code and associate SIG

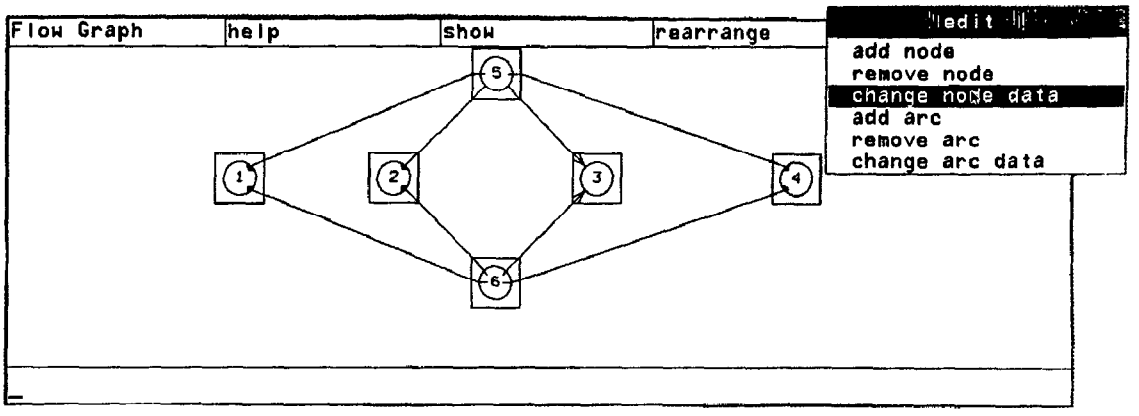

Figure 6. Sample graph editing session

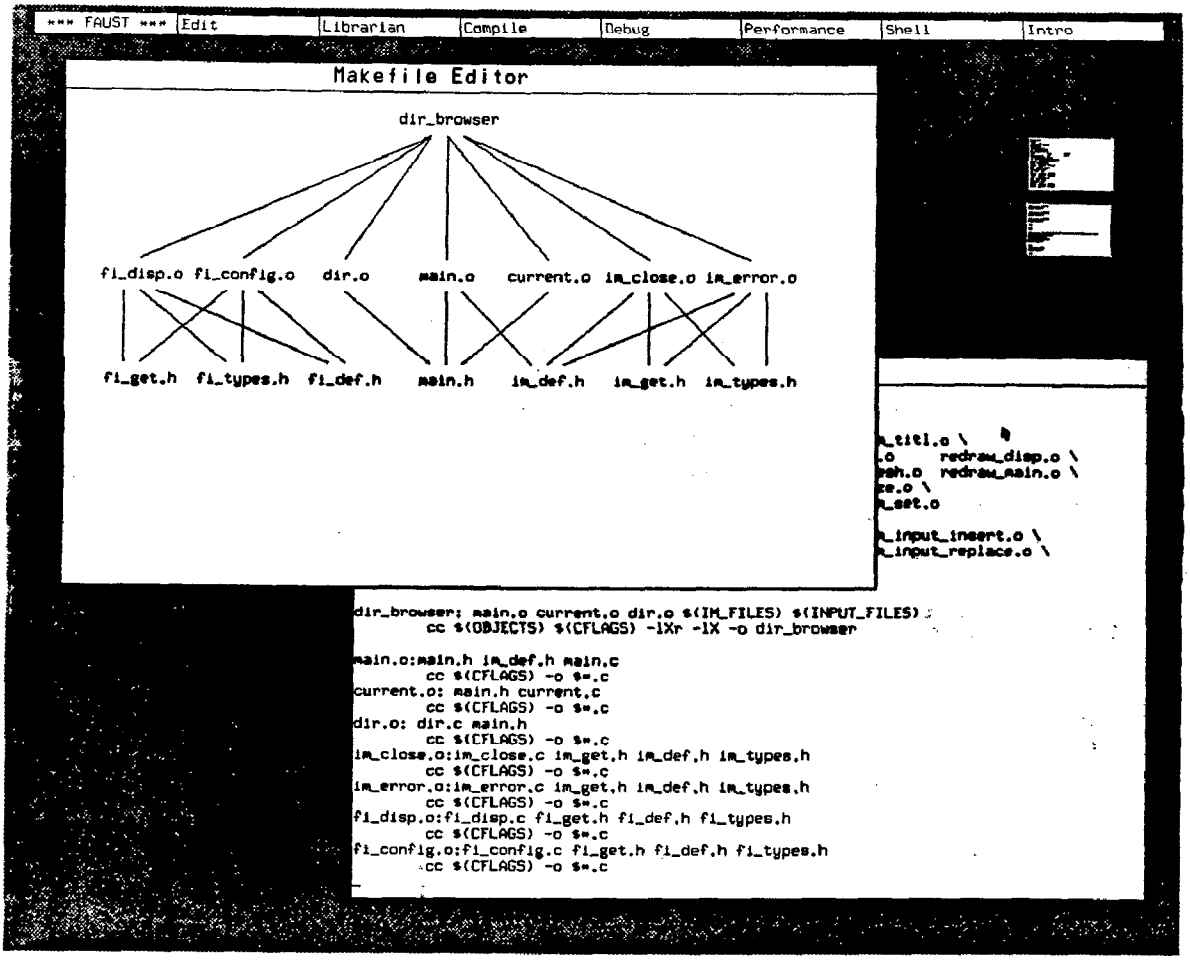

Figure 7. Graphical makefile editor

(2) Functions that allow *syntactic* editing. Syntactic changes are ones that do not affect the underlying graph structure, but allow the user to redefine the mapping of the graph object to the screen. An example of this type of editing is a subroutine interconnection graph. The geometry for a SIG is originally determined automatically by the graph browser when an unzoom operation is requested from a source window. The user may subsequently change the placement of nodes on the screen in order to make the graph more visually appealing since the geometric layout of the graph does not affect the underlying graph data structure. Such changes can be done through the Interface Manager without assistance by the application program.

(3) Functions that allow *semantic* editing. Semantic changes can modify the underlying program abstraction by changing the associated data. For example, the deletion of a subroutine node may involve commenting out the subroutine definition and all references to the subroutine. Usually such semantic changes are application dependent; consequently, applications are notified of the requests and are expected to provide functions to handle them.

Most operations in categories (1) and (2) can be handled directly by the Interface Manager without intervention by client applications. Some sub-operations in category (3) such as selections and highlighting can also be handled by the Interface Manager. Default functions are provided for all categories. Some default functions may just print error messages.

Every graph operation described above is composed of a maximum of four steps:

(1) The user selects the graph operation from a pull-down menu (see Figure 6).

(2) The user is allowed to select the desired graph objects to be affected by the chosen operation. Selected graph objects are highlighted as they are selected. For example, after selecting the DELETE NODE command the user may point to any number of nodes to be deleted followed by clicking on a "confirm" button.

(3) The requested editing operation is performed on the selected objects.

(4) The user is notified of any errors. The detection of an error results in the creation of an error window that waits for an acknowledgement.

It is the responsibility of the application writer to redefine any of the default graph editing functions that are not desired. Applications may override default functions by specifying a pointer to a function to be called for the particular operation of interest. In general, the Interface Manager identifies certain common graph manipulation operations and creates a standard default user interface through which all applications operate. Applications using the system therefore present a common user interface with appropriate context-specific additions.

4.1. Graphical Program Construction

Although subroutine interconnection graphs can be computed automatically by the Interface Manager from program source code, the graph editing feature may be used to develop applications graphically and later attach source code to subroutine nodes. By using the IM to build applications in this manner, Faust approximates a Structured Design environment [Page80].[13] Applications can be defined graphically in terms of function hierarchies, with source code added as implementation proceeds.

As source code is attached to SIG nodes, information pertaining to the calling parameters is attached to the arcs of the graph. In this way, the user may perform zoom operations on arcs in order to view details associated with function interfaces.

4.2. A Graphical Makefile Tool

One program development tool that is easily constructed with the Interface Manager is a graphical version of the Unix "Make" utility [Feld84]. The graphical makefile editor allows the user to specify program dependences by graphically creating a directed graph (see Figure 7). At the root of the graph (tree) is the executable object that is to be created. The next level of the tree consists of all of the object files required to generate the executable. Each object file serves as the root of a subtree that defines all files necessary to generate it.

The graphical makefile editor highlights those executable files that are out of date or inconsistent to remind the user which recompilations need to be performed. The user may specify specific subtrees to be recompiled or allow the system to perform all necessary build operations.

The rationale for developing the graphical makefile editor is twofold. First, it serves as a nice demonstration of the Interface Manager tools with relatively little development effort. Second, the nature of traditional makefiles was seen as tedious, making the investigation of a graphical approach attractive. Although building makefiles through the graphical editing tool requires as much (if not more) effort than standard makefiles, the final product appears to be more intuitive, giving users a better understanding of the relationships in the program. Additionally, the graphical representation provides a convenient method for inspecting the state of the application with respect to object file consistencies.

[13] Faust lacks many of the tools needed to be considered a true Structured Design environment such as those offered by Cadre [Cadr85] and IDE [WaPi87]. Faust is not intended to address the problems of Structured Analysis and Design in detail. However, the similarities are worth noting.

5. CONCLUSIONS AND FUTURE WORK

Several aspects of the Faust environment will be considered when evaluating the success or failure of the project. One aspect is the effectiveness of the program abstraction mechanism. The design of the hierarchical model and associated user interface operations was, without question, the single most challenging problem to face in the Faust project. Presenting a unified graphical/textual program interface to the user is fraught with technical and operational problems. Feedback from several users is needed to determine if the benefits derived from the hierarchical program model warrant the effort invested in the support tools. Much of the work described in this paper is nearing completion at the time of this writing and therefore, user feedback is not yet available. Although one stand-alone tool using the program graph model has been constructed and is available for use at this time, deployment of the first version of the environment is not expected until Summer of this year.

Another area of interest to the Faust project is the success of the support libraries. Of particular interest is a study of the level of complexity reduction that can be accomplished in higher level tools by using the Faust interface library. One data point already collected is the Sigma editor. Sigma was originally written to run on Apollo equipment using explicit text management code and Apollo Display Manager primitives [Apol85]. When converted to run with the Faust text handling utilities, SIGMA was reduced from 12,000 lines of code to 9,000 lines.

Finally, the question of portability must be answered. Preliminary experience has shown that the choice to use X Windows was a good one. Initial versions of the Faust user interface tools have been developed on Apollo workstations running BSD 4.2 Unix [Berk84]. Porting the libraries to a Sun environment required only recompilations with minor changes to Makefiles to account for pathname differences. The libraries will be ported to additional configurations as access becomes available.

Future efforts will diverge in two areas. First, the libraries and interfaces described in this paper will be revised as reactions are collected from applications programmers. Several areas in the Faust tool kit have already been recognized to be too inefficient. As frequently happens in system design, generality breeds complexity. The right balance between functionality and performance will be seriously considered in future versions. Second, high-level tools using the libraries must be constructed to build parallel programming tools. Tools such as performance evaluation and debugging systems will be constructed and integrated into Faust. At that time, the applicability of the program abstraction model will become more apparent as the integration effort for the now-disjoint applications is done.

6. ACKNOWLEDGEMENTS

This work was supported in part by the National Science Foundation under Grants No. US NSF MIP-8410110, the U. S. Department of Energy under Grant No. US DOE-DE-FG02-85ER25001, the U.S. Air Force Office of Scientific Research Grant No. AFOSR-F49620-86-C-0136, and the IBM Donation.

REFERENCES

[Apol85] Apollo Computer Incorporated. "Programming With DOMAIN Graphics Primitives", Software Release 9.0, Part No 005808, July, 1985.

[Cadr85] Cadre Technology. *Improved Quality and Productivity with Teamwork/SA.* **System Development**, October, 1985.

[CCHK87] Alan Carle, Keith D. Cooper, Robert T. Hood, Ken Kennedy, Linda Torczon and Scott K. Warren. *A Practical Environment for Scientific Programming.* **IEEE Computer**, Vol. 20, No. 11, pp. 75-89, Nov., 1987.

[Feld84] S. I. Feldman. *Make - A Program for Maintaining Computer Programs.* **UNIX Programmer's Manual, Reference Guide--4.2 Berkeley Software Distribution**, March, 1984.

[GJMY87] K. A. Gallivan, W. Jalby, A. D. Malony and P. C. Yew. *Performance Analysis on the CEDAR System.* In: **Performance Evaluation of Supercomputers**, J.L. Martin, ed. Elsevier Science Publ. (North-Hollan), Amsterdam, The Netherlands, 1987.

[GALS87] Dennis Gannon, Daya Atapattu, Mann Ho Lee and Bruce Shei. *A Software Tool For Building Supercomputing Applications.* **Parallel Computations and Their Impact on Mechanics**, Vol. 86, pp. 81-92, December, 1987.

[GeND87] Jim Gettys, Ron Newman and Tony Della Fera. *Xlib - C Language X Interface Protocol Version 11.* **Programming With the X Window System**, September, 1987.

[JaGu88] David Jablonowski and Vincent A. Guarna Jr. "A Dynamic Graph Tool and Its Use in an Integrated Programming Environment", CSRD Report No. 746, February, 1988.

[Metc87] M. Metcalf. *Fortran 8X - The Emerging Standard.* **ACM Fortran Forum**, Vol. 6, No. 1, April, 1987.

[MSCH86]James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, 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, pp. 184-201, Mar., 1986.

[PaGL87] David Padua, Vincent A. Guarna Jr. and Duncan Lawrie. *Supercomputer Programming Environments.* **Parallel Computations and Their Impact on Mechanics**, Vol. 86, pp. 55-79, December, 1987.

[Page80] Meilir Page-Jones. **The Practical Guide To Structured Systems Design.** Yourdon Press, New York, 1980.

[ScGe86] Robert W. Scheifler and Jim Gettys. *The X Window System.* **Programming With the X Window System**, November, 1986.

[Berk84] University of California. **UNIX User's Manual, Reference Guide--4.2 Berkeley Software Distribution.** Computer Science Division, University of California, Berkeley, California, 1984.

[WaPi87] A. I. Wasserman and P. A. Pircher. *A Graphical, Extensible Integrated Environment for Software Development.* **SIGPLAN Notices, Proceedings of the Second ACM SIGSOFT/SIGPLAN Software Engineering Symp. on Practical Software Development Environments**, pp. 131-142, January, 1987.