CSRD Rpt. No. 829

AN ENVIRONMENT ARCHITECTURE AND ITS USE IN PERFORMANCE DATA ANALYSIS

Allen D. Malony and
Joseph R. Pickert

October 1988

Center for Supercomputing Research and Development
University of Illinois
305 Talbot – 104 South Wright Street
Urbana, IL 61801–2932
Phone: (217) 333–6223

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

An Environment Architecture and Its Use in Performance Data Analysis and Visualization

Allen D. Malony* Joseph R. Pickert*

November 8, 1988

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

1 Introduction

An environment architecture supporting performance data analysis and visualization should allow tools to be designed and developed based on their individual functionality while providing a common framework for the use of the tools in the environment. Instead of statically binding operational aspects of the environment into the tool at compile time, the architecture should promote the separation of tool interaction from tool application by providing services for tool communication. This encourages the development of simple tools (performance tools, user interface managers, etc.) with well-define external interfaces and less restricted internal implementations that can more easily be combined by users and environment application developers.

Because of the variety and combinations of performance data, tools for performance data analysis and visualization cannot be designed as monolithic applications. The set of performance displays alone indicates that several tools must be provided if a reasonable set of graphical rendering techniques is to be supported. More importantly, the way a user combines the analysis and visualization functions cannot be anticipated and it is impossible to statically code all interactions into a single program.

This paper describes an environment architecture that supports the requirements of performance data analysis and visualization as well as offers a framework for designing other environment tools and services. The ideas given are simple and can be found in many other contexts. Examples of performance tools and user interface management tools that could be built based on the environment architecture are given.

*Supported in part by NSF Grants NSF MIP-8410110 and NSF DCR 84-06916, DOE Grant DOE DE-FG02-85ER25001, Air Force Office of Scientific Research Grant AFOSR-F49620-86-C-0136, and a donation from IBM.

2 An Environment Architecture

The proposed environment architecture combines the notion of a tool with a data-flow approach to tool interaction. A tool embodies the object-oriented paradigm of a high-level data type (or object) abstraction whose implementation is hidden from the user but whose behavior is described through a well-defined public interface. For a tool to exist as part of an environment, however, an underlying communication mechanism must be provided that allows other environment components to access the tool's interface. A goal of the environment architecture is to provide a framework by which tools can be designed and developed independent of their instantiation in an environment and isolated from the communications mechanisms for tool interaction.

In object-oriented parlance, each tool object is considered to have two parts: private and public. The tool private part is composed of data structures and code fragments that implement the tool's function. The tool public part defines a set of messages for invoking the tool's methods and a protocol for tool interaction. The tool interface assumes that facilities exists for passing messages between the tool and the rest of the environment.

In general, the architecture places no requirements on how a tool is to be instantiated in the environment. Tools have been described elsewhere as independent processes [2,5] as well as independent tasks [13] and threads of execution [9]. As long as the logical message passing tool interface is maintained, the tool can be instantiated in whatever manner is most appropriate for its use, even as a dynamically linked object library. The low-level data communication support will be dependent upon tool instantiation.

In addition to the architecture defining how a tool must operate in the environment, it must also specify how message passing tool communication will be implemented. Here we distinguish two layers analogous to the datalink and network layers of the ISO OSI network model [11] called the tool *data interchange* (DI) layer and the tool *message interchange* (MI) layer, respectively. The DI layer is responsible for moving message data between tools and other components within the environment. The MI layer supports a virtual circuit messaging interface for the tool object relying on the DI layer for message transfer.

Figure diagrams the environment architecture. The rest of the paper describes ideas for implementing this environment architecture. The DI and MI layers for message passing support are discussed in Sections 3 and 4, respectively. Approaches to tool construction is the topic of Section 5. Some examples of performance tools and manager tools are considered in Sections 6 and 7 to better understand how tools might be constructed and how they might operate in the environment.

Background

The environment architecture described here is an almalgamation of ideas from languages, operating systems, communications, and programming environments. The tool-oriented model is clearly not new. Tools have long been part of the Unix world as has the notion of connecting tools together using the pipe and socket inter-process communication facilities [6]. Many systems dealing with aspects of windowing environments user interfaces, interactive graphics, algorithm visualization, and programming environments have also promoted the concept of toolkits.

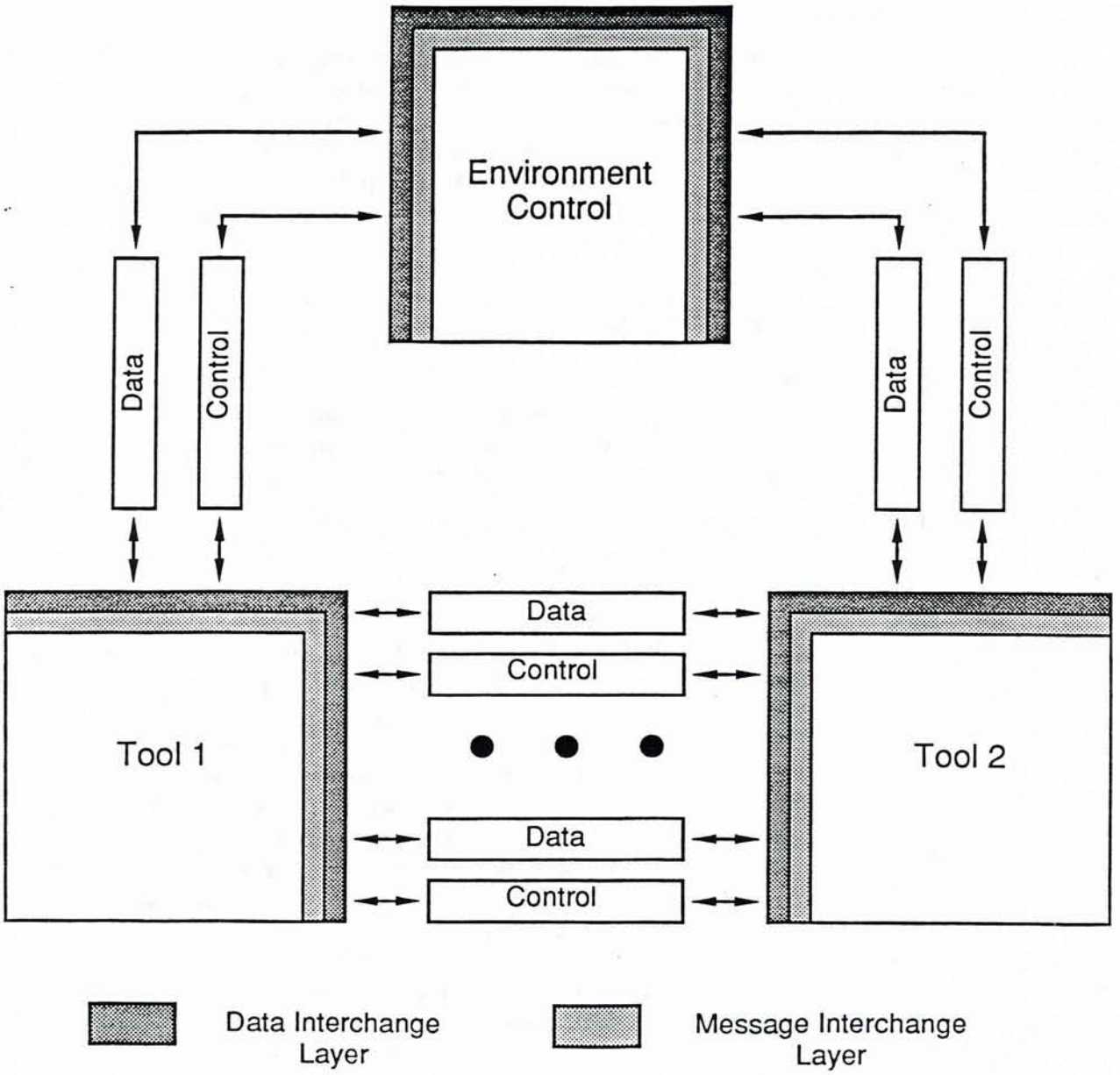


Figure 2: Environment Architecture

Defining interfaces and protocols for communication between tools and processes is an important subject in computer networks [11], distributed operating systems [12], language design [5], and computational environments [7,8]. Message passing as a mechanism for data interchange has had many applications including its implementation in graphics and in programming environments as actual run-time systems supporting the message metaphor of invoking methods of different environment objects. Gentleman's administrators [2]; Cardelli's fragments of behavior [1]; Tanner's servers, clients and switchboard [13]; Purtilo's message handler [8]; and Haeberli's ConMan [4,3] either make use of or provide mechanisms for message handling within an environment.

3 The Data Interchange Layer

The DI layer implements the mechanism for transferring data to and from tools. The techniques used for any particular instantiation of a tool depends on the association between the tool and the environment entity that is using the tool. Three such associations are identified: process, task, and routine. The data interchange mechanisms are described below for each of these associations.

3.1 Process Data Interchange

The DI layer for the use of a tool instantiated as a process must be based on some form of inter-process communication (IPC) [1]. Berkeley UNIX supports IPC with sockets and the use of datagram or stream communication. Because the DI layer must support virtual circuit communication, socket streams will be the implementation of choice.

The process DI layer (PDIL) supports connection establishment and control for the use of a tool through sockets as well as the sending and receiving data across the current set current socket connections. The basic implementation approach is shown in Figure 3.1.

The PDIL manages a certain number of sockets through a socket table. Each low-level socket is implemented as a socket pair for full-duplex communication. As will be discussed in the MI layer section, each message-level connection consists of control and data messages. We will represent this at the PDIL by a data socket and a control socket for each connection [2].

Associated with each socket is a port that interfaces the MI layer with the process DI layer; see Figure 3.1. A port is essentially a large data buffer that provides some flow control support by smoothing out the transfer of data between the system-maintained socket buffers and the tool-internal message buffers. Ports also perform data-to-message and message-to-data transformations using a simple PDIL message format:

# message data bytes	message data

If the tool process does not exist at the time of a connection request, the tool process is started. The PDIL always maintain a known *environment management* data and control socket;

[1] We are making a simplifying assumption here that processes do not share memory. Memory sharing between processes is possible in some operating systems. In this case, techniques described for tasks might also be appropriate.

[2] Separate control and data sockets for each connection are assumed for sake of discussion. In some cases, it might be appropriate to have data and control multiplexed across the same socket.

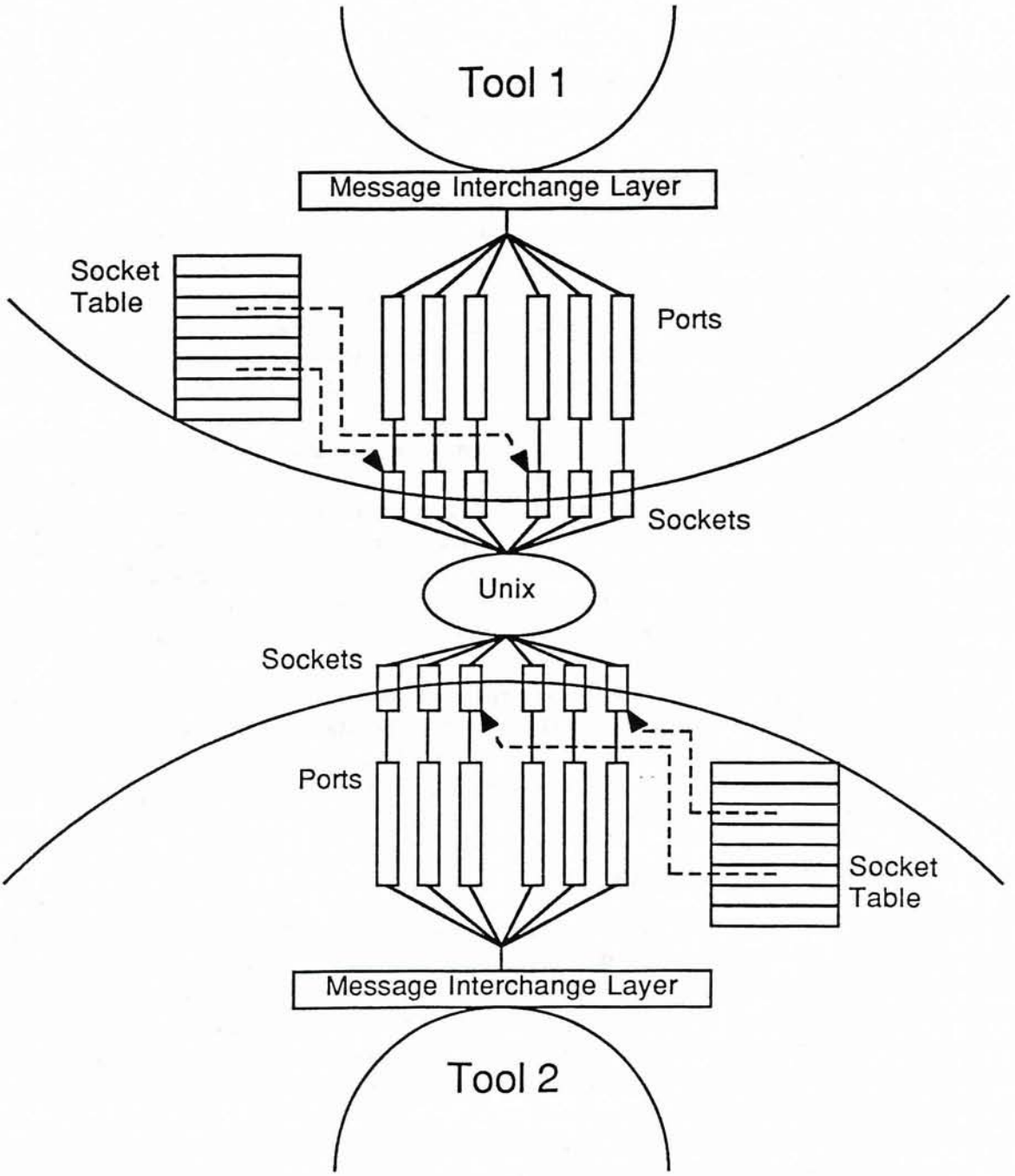


Figure 3.1: Socket Implementation

see Figure 2. It is with these sockets that the socket control component of the PDIL receives and processes messages to create, report on, and terminate process connections.

3.2 Task Data Interchange

The process DI layer is the hardest to describe because of the need to define the low-level socket data transfer and to use environment and system management facilities to establish tool connections. The assumptions that we will allow in the case of a tool instantiation as a task is that there already exists standard messaging mechanisms between tasks or that they can be easily constructed. The tool itself is created as a task of the program wanting to establish a connection to the tool. Therefore, the task DI layer (TDIL) should be able to make use of any inter-task communication mechanisms that are available.

As an example, assume a multitasking environment where tasks can share memory. The tool connection establishment procedure involves the creation of the tool task, if it has not already been created, and the setting up of data and control message buffers equivalent to socket buffers in the PDIL case. The low level message-to-data transformations are not necessary here as the unit of communication can be the message itself. The copying of messages from the TDIL buffers to message buffers of the MI layer may or may not be done depending on the degree of autonomy desired between the two layers and the efficiencies of accessing task-local MI layer message buffers versus global TDIL buffers.

It is still reasonable to think of an environment management component of the TDIL. Since the DI layer control procedures for managing connections interact with the environment manager through messages, most of the code used in the PDIL case could be used here. Although certain aspects of the protocol might be superfluous and thus could be simplified, it would promote consistency and reuse of the DI layer control software.

3.3 Routine Data Interchange

The distinction between a task DI layer and a routine DI layer (RDIL) is that the TDIL layer supports the tool as a separate thread of execution. This is not the case in the RDIL. Here the tool exists as a piece of the program code that is using the tool at the same level as a program routine. The program and the tool use the same thread of execution. However, the interaction between the program and the tool is still in the form of messages. It is equivalent to thinking of the program and the tool as multiplexing the processing resource and communicating through message passing mechanisms[3].

Certain efficiencies can be gained in the RDIL because the tool is so tightly coupled with the program. Connection management functions are simplified. Also, RDIL buffers and MI layer buffers can be made the same, thereby removing a level of message handling.

However, there are a few issues that arise. The most important concerns the transfer of execution control between the program and the tool. There must be policies adopted for the passing of control to avoid problems such as starvation and deadlock. The problems that

[3]It should be kept in mind that we are always supporting a message style connection with a tool. Although the tool could be built completely as a library of routines that can be directly called by a program, we are intentionally requiring the tool interaction to be in the form of messages. This forces the tool designer to think of a tool as an object and leads to an implementation whereby the tool can use a routine, task or process DI layer with the tool "object" code remaining unchanged.

can exist are program and tool dependent. Limited or no execution control mechanisms are necessary in some cases, such as when the tool guarantees an acceptable response time back to the program. In other cases, tighter control policies might have to be enforced. This whole subject requires further study.

3.4 Dynamic Linking

One subject that was glossed over in the discussion of the task and routine DI layers was that of how the tool code was linked in with the program wanting to use the tool. There are two possibilities. First, a tool library could be linked in with the program at compile time. The second alternative is to linked in the tool dynamically when the tool is requested during program execution. This service would be provided through a dynamic object code linking facility that uses a tool name passed by the program as the identifier for the tool code plus a DI layer type field indicating the type of DI layer desired, task or routine.

3.5 Observations

It is interesting to see how dynamic linking together with message-based communication supports the concept of a reconfigurable tool environment. Although particular tool protocols must be adhered to, the different alternatives for tool instantiation can be experimented with without modifying the tool message interface. It would also be possible to modify the tool itself without recompiling programs that use that tool, as long as the protocol remains the same.

4 The Message Interchange Layer

The message interchange layer supports the message communication mechanisms for a tool. It provides a simple interface to the tool for message buffer allocation, message creation, and message sending and receiving. Internally, it manages outgoing and incoming message queues for each DI layer port. Free message buffers are also maintained by the MI layer for use in constructing new messages to send and in separating messages being received.

A simple set of MI layer interface routines are listed in Appendix A. Notice the *vc* field in the MsgSend() and MsgRecv() routines representing a virtual circuit connection. An additional function of the MI layer is to support a basic mapping between logical tool-level connections (referred to here as *virtual circuits*) and the DI layer ports; it is assumed, for sake of simplicity, that this mapping is one-to-one. A data and control vc is defined for each DI layer port.

The implementation of the MI layer is logically independent of a tool's instantiation. However, it is possible that performance efficiencies can be gained in certain situations that would make it advantageous to have alternative MI layer mechanisms. As mentioned earlier, the ability to effect the DI layer through shared memory, in the cases of a tool as a task or as a routine, allows message buffers to be shared also. In this case, the MI layer could merely pass message buffer pointers instead of physically copying the message data between sending and receiving buffers. An even tighter coupling is possible with the tool is a routine. Here the MI layer could be a simple synchronous message interchange implemented through indirect MsgSend() and MsgRecv() argument passing. This, however, is a restriction of the general message passing paradigm and it is probably the case that message buffer pointer passing offers enough

performance efficiency to make message transfer through argument passing unnecessary.

5 Tool Design Methology

The design of a tool for the environment architecture described above involves two steps. First is the tool function design. That is, the data structures and routines that implement the tool's intended function must be specified and developed. The designer should think of the tool in an object-oriented manner that separates the tool's function from its use. This will make for a clean tool interface.

The second step is the specification of the tool interface in the form of tool message types and a tool protocol. To maintain some consistency in tool design, an attempt should be made to adopt common message format conventions, especially in the case of similar tools. We will explain each of these steps in more detail below using as example a simple tool for 2-D point and line plotting.

5.1 Tool Function Design

Suppose we wanted to build a tool for 2-D point and line plotting that would eventually be available as part of our environment. We begin by defining the plotting functions that the tool must support. The set of features for our simple 2-D plotting tool are:

- individual point plotting
- line plotting with and without point marks
- linear x and y axis
  - setting of lower and upper values
  - setting of number of divisions
- automatic plot clipping
- plot title labeling
- x and y axis labeling

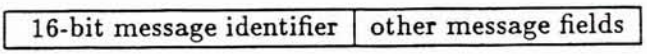
Based on this feature set we can develop the data structures and code needed to implement the desired plotting functionality. However, during this process it is necessary to differentiate between the drawing of the plot and the specification of the plot as well as what interactions the tool will provide directly to the user and those that will be supported through the message passing interface to the environment.

Here it is reasonable to focus the plotting tool's internal implementation on the actual drawing of a plot. The specification of the individual components of the plot will be done through the tool's user interface. We will, however, allow a user to interact directly with the plotting tool to select different icons to serve as point marks in the plot.

5.2 Tool Interface Specification

Once the internal functionality of the tool has been decided upon and implemented, it is necessary to specify the tool's interface to the environment. The tool interface is composed of a set of messages that the tool receives and sends, and a message protocol. Because users of the tool only see its interface, the messages and protocol must be clearly defined.

The format of the messages is completely determined by the tool. However, a simple high-level format, such as the one shown below, will help maintain consistency in message structure across tools.



The messages that a tool receives should be thought of as invoking some tool action or registering some information with the tool. Those functions and services the tool wants to make accessible to the environment at the tool interface should be cast in different message types, which define the other message fields, and should be given separate message identifiers. Messages sent by a tool are either responses to messages it receives, reports of its internal operation, or requests for services performed by other tools. Again, those send messages the tool defines should be clearly specified.

In addition to the set of messages, a tool might require certain protocols to be adhered to for its use. It is more difficult to explicitly specify the protocol in some physical form, like a message, because it depends more on the sequence of messages. However, the protocol can be stated in the form of interface restrictions and expected message sequences.

The interface specification for the plotting tool is shown in Appendix B. To keep it simple, there is only one send message, *Error*. The receive messages either pass plot data or request that certain plot display actions be performed. Most of the messages are self-explanatory. Two protocol restrictions are given.

5.2.1 A Tool Interface Generator

One possible approach to the design of a tool interface is to think of the set of messages and the message protocol as forming some simple tool language. As such, the message formats as well as some aspects of the tool message protocol might be described through a tool interface grammar. Each production in the grammar would represent a message type. Protocol sematics could be specified as attributes in the grammar allowing protocol violations to be detected and properly handled. A tool interface generator could then input the grammar and automatically build the tool interface. Ideas similar to the specification and generation of tool interfaces can be found in language-based environments [10], environments for software interconnection [7,8], and user and graphic interface design [1,3].

Depending on the complexity of the tool, it might appear in some cases where message types and protocol are trivial, that a tool interface grammar is an unnecessary level of design. The benefit, however, is the tool interface definition flexibilities it provides for selecting message formats and message designators, and for extending the language if and when additional message types are added to the set. Furthermore, advantages can be gained through the tool interface generation capabilities where standard message identification and parsing functions can be used.

6 Performance Tools

Various types of performance analysis and visualization tools will exist as part of a performance evaluation environment. Many of these tools can conceptually be viewed as performance data transformers. Performance data can come from several sources: simulation systems, program software instrumentation, operating system measurements, machine hardware monitors. Furthermore, the data can be post-mortem or real-time. Analysis tools filter and condense the performance data while the visualization tools are concerned with representing the data in meaningful textual and graphical ways. The following briefly describes anticipated performance analysis and visualization tools that will be developed.

6.1 Analysis Tools

Performance data analysis can be as simple as basic counting and timing, or as complex as determining processor/task/system level parallelism or potential parallelism using synchronization and dependency information. The analysis tools input performance data from files or from sources generating data in real-time, and output analysis results to other tools that will present them.

Counting and timing tools can be combined into the general class of profiling tools. Profiling functions to be supported include counting program and system level events, determining time spent in code sections, and keeping use statistics of system and machine level resources.

Higher-level analysis tools perform more complex and potentially more time-consuming operations on the performance data. These operations could require larger and more global data to be analyzed or keep complex state information about a parallel execution. A tool to determine task concurrency statistics, for instance, must monitor the simultaneous state of all tasks participating in a parallel execution to detect concurrency state transitions. To determine virtual parallelism, such a tool must be extended to take into account the inter-task synchronizations and dependencies.

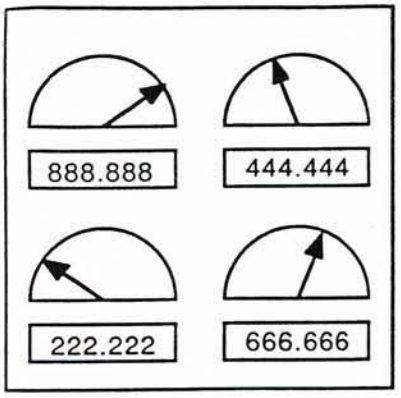
6.2 Graphics Tools

A variety of tools can be used to visualize performance results. In addition to the tools shown in Figure 6.2, there are 2-D surface/contour/scatter plots, 3-D plots, strip charts, process/task/subroutine graphs, and so on.

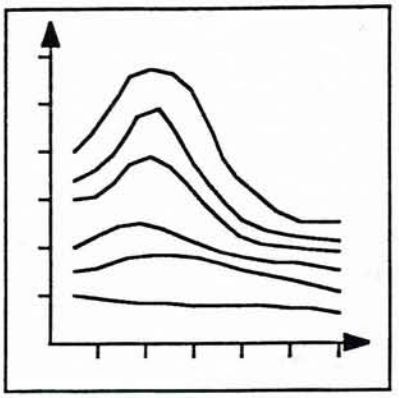
Each of these tools is essentially an output display tool. Performance data is input at the tool's interface, in the form of raw data as well as analysis results, and is presented textually and/or graphically to the user. The tools might provide user-level interactions for selecting variants of the display type or focusing on certain aspects of the performance data.

6.3 Performance Tool Integration

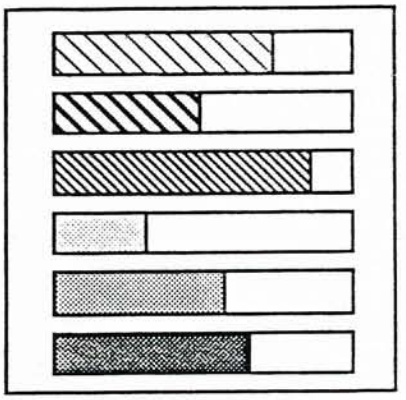
It is hoped that the performance analysis and visualization tools can be integrated with the performance data collection facilities to the extent that the user has a consistent framework for specifying performance experiments, analyzing the performance data, and visualizing the performance results. Analysis and display tools should be as easily selected and enabled as the tools for performance measurement. For real-time operation in particular, support in the



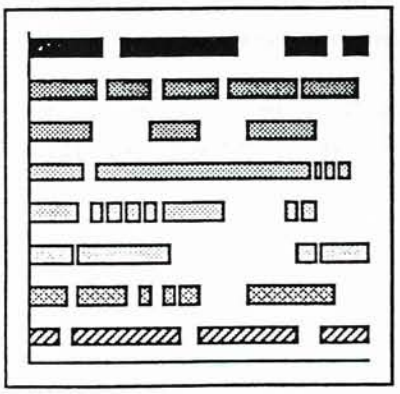
Analog and Digital Meters



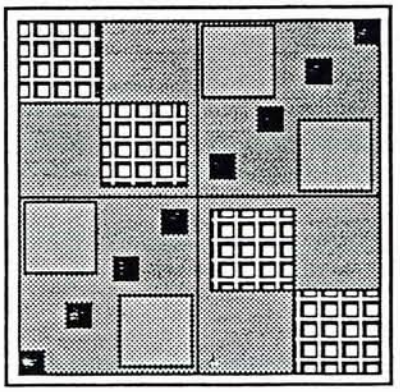
2-D Plots



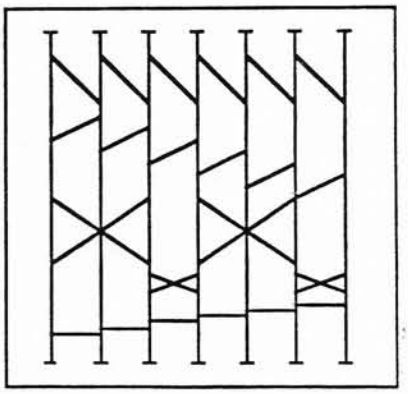
Histograms



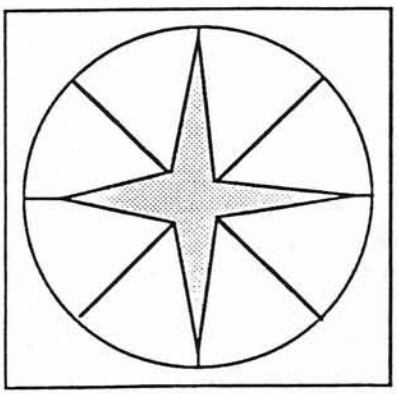
Event Graphs



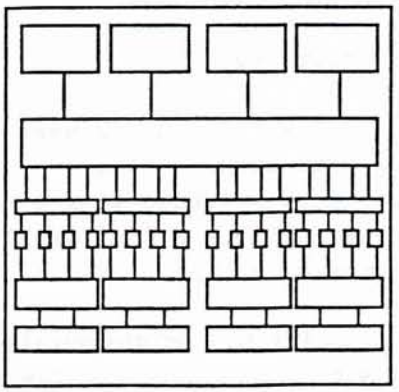
Array Displays



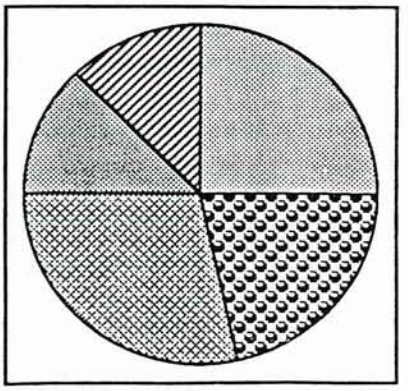
Concurrency and Synchronization Graphs



Kiviat Diagrams



Topology Displays



Pie Charts

Figure 6.2: Performance Displays

environment must exist for establishing the connections between the different tools and in handling the message data transfer.

7 User Interface Managers

A number of standard tools should be supported by the environment. These standard tools work together to enforce a user-interface that is consistent across the range of tools. For instance, a menu manager can be provided to manipulate menus. A tool simply "registers" its menus with the menu manager, and the tool is notified when a particular menu item is selected by the user. Note that the menu manager may be a library, another task, another process, or on another machine. A tool is not concerned with the particular implementation of the menu manager, only its interface.

The interface to the menu manager should be robust enough to allow a tool to create, destroy, and dynamically change various aspects of its menus. The menu manager interface might be specified as follows:

```
ERROR Menu(id, message, data)
int32 id;
int32 message;
int32 data;
```

where *id* is a unique identifier which specifies the menu instance, *message* is an indicator of the function to be performed, and *data* is something that makes sense in the context of the function being performed.

```
A non-exhaustive list of messages includes:

  NEW: create a new menu (or submenu),
  FREE: destroy a menu,
  ACTIVATE:     activate a deactivated menu or menu item,
  DEACTIVATE:   deactivate a menu or menu item,
  SETFUNCTION:  set function to be executed when a menu item is selected,
  SETITEM: a bitmap to be used in place of (or with) the text,
  MODIFY: modify the appearance of a menu item, as specified by the data.
```

In addition to a menu manager, a list of other obvious choices for user-interface managers would include a dialog manager, a color manager, a window manager, a list manager, and a text manager.

8 Conclusion

The use of performance analysis and visualizaton tools in a supercomputer programming environment necessitates an environment architecture in which the tools can be developed individually to meet their desired internal functionality yet provide a well-defined interface through

which other environment components can access the features of the tool. The environment architecture described above supports this approach to tool design and interaction with a simple message passing model for tool communication.

Development of an environment based on this architecture must proceed along two fronts: tool implementation, and the implementation of the data interchange and message interchange layers. Tools other than performance tools can take advantage of the logical separation of tool interaction from tool function. Managers of various types as well as more graphics oriented tools are obvious candidates. The support for message passing communication in the environment must take into account the physical computer system configurations on which the environment and the tools will be running. For instance, performance data collection might be done on one machine with the data analysis done on a workstation which in turn sends results to a performance visualization tool running on yet another workstation that drives a graphics display. In addition to allowing tools to execute in a distributed system, the environment's message communication should also be able to take advantage of the multiprocessing and multitasking features of the machines themselves.

Components of the described environment architecture are already under development. A first implementation of underlying IPC socket mechanisms for the DI layer is almost complete. This includes a simple environment control for the instantiation of tools as processes as well as the interconnecting of tools together using sockets. A few basic performance visualization tools (meters, histograms, and a basic strip chart) are being designed and will be implemented within the next month. We will then use these tools together with the environment support for the DI and MI layers to learn about the efficiencies of the architecture for different system configurations and tool instantiatons.

References

[1] L. Cardelli and R. Pike. Squeak: a language for communicating with mice. In *Computer Graphics*, SIGGRAPH, 1985.

[2] W.M. Gentleman. Message passing between sequential processes: the reply primitive and the administrator concept. *Software Practice and Experience*, 11(5):436–66, May 1981.

[3] P.E. Haeberli. Conman: a visual programming language for interactive computer graphics. *Computer Graphics*, 22(4):103–11, August 1988.

[4] P.E. Haeberli. A data-flow manager for an interactive programming environment. In *Usenix Summer Conference*, 1986.

[5] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), August 1978.

[6] B.W. Kernighan and R. Pike. *The UNIX Programming Environment.* Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[7] J.M. Purtilo. Polylith: an environment to support management of tool interfaces. In *ACM SIGPLAN Symp. on Programmig Issues in Programming Environments*, pages 12–18, June 1985.

[8] J.M. Purtilo. *A Software Interconnection Technology to Support Specification of Computational Environments.* PhD thesis, University of Illinois, 1986. Department of Computer Science, Report UIUCDCS-R-86-1269.

[9] J.M. Purtilo, D.A. Reed, and D.C. Grunwald. Environments for prototyping parallel algorithms. In *1987 Int'l. Conf. on Parallel Processing*, pages 431–38, August 1987.

[10] T.W. Reps. *Generating Language-Based Environments.* MIT Press, Cambridge, Mass., 1983.

[11] A.S. Tanenbaum. *Computer Networks.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[12] A.S. Tanenbaum and R. Van Renesse. Distributed operating systems. *Computing Surveys*, 17, December 1985.

[13] Peter B. Tanner, Stephen A. MacKay, Darlene A. Stewart, and Marceli Wein. A multitasking switchboard approach to user interface management. In *Computer Graphics*, 1986.

A MI Layer Interface Routines

```
typedef char    *MSGP;

/*************************************************************************/
/* MsgAlloc() allocates a free message buffer from the free list.  NULL */
/* is returned if none is available.  Otherwise, a pointer to the free  */
/* message is returned.                                                  */
/*************************************************************************/
MSGP MsgAlloc()

/*************************************************************************/
/* MsgFree() returns a message buffer to the free list.                  */
/*************************************************************************/
void MsgFree(msg)
MSGP    msg;

/*************************************************************************/
/* MsgBuild() builds a message from multiple pieces of data.  At this    */
/* level, the message format is just a length (in bytes) followed by     */
/* the data.  Each data item is identified by a pointer to the data and */
/* a byte count.  A NULL pointer terminates the argument list.  The      */
/* total number of bytes should not be greater than the maximum message */
/* size.  If it is, multiple messages will be sent.  A maximum of 10     */
/* messages are allowed.                                                 */
/*                                                                       */
/* Messages buffers are gotten from the free list.  If no buffers are    */
/* available at any time during the building process, the entire         */
/* operation is terminated.  None of the data will be sent and 0 is      */
/* returned.  Otherwise, 1 is returned.                                  */
/*************************************************************************/
int MsgBuild(data)
struct {char *p; l} data;

/*************************************************************************/
/* MsgSend() sends a message out on a virtual circuit.                   */
/*************************************************************************/
int MsgSend(vc, msg)
int     vc;
MSGP    msg;

/*************************************************************************/
/* MsgRecv() receives the next message in on a virtual circuit.          */
/*************************************************************************/
MSGP MsgRecv(vc, msg)
int     vc;
MSGP    msg;
```

B Plot Interface Specification

B.1 Messages Received

Point	x value	y value
Line Begin		
Line End		
X Axis	low	high
X Divisions	number	
X Label	label	
Y Axis	low	high
Y Divisions	number	
Y Label	label	
Draw Plot		
Clear Plot		

B.2 Messages Sent

Error	message

B.3 Protocol Restrictions

1. Default values are chosen in case the *Draw Plot* message is received before all plot components have been specified.
2. A line is defined by a *Line Begin* message followed by a group os *Point* messages followed by a *Line End* message.