1995

DISplay - A Visualization and User Interaction Interface for Parallel Simulation Environments

Edward Mascarenhas

Vernon J. Rego
Purdue University, rego@cs.purdue.edu

Report Number:
95-016

DISPLAY - A VISUALIZATION AND USER INTERACTION INTERFACE FOR PARALLEL SIMULATION ENVIRONMENTS

Edward Mascarenhas
Vernon Rego

Computer Sciences Department
Purdue University
West Lafayette, IN 47907-1398

CSD-TR-95-016
March 1995

DISplay - A Visualization and User Interaction Interface for Parallel Simulation Environments *

Edward Mascarenhas
Vernon Rego

Department of Computer Sciences
Purdue University
West Lafayette, IN 47907

Abstract

An application-independent visualization interaction system is proposed, with potential for application-binding at any stage of the modeling process. Advantages of this approach include ease of use, flexibility, code reuse, and modularity. Our design ideas are manifest in the DISplay system, a graphical user interaction and display library which can be used with any parallel software. We outline its use in the the dynamic display of results from computations in queueing and distributed particle-physics simulations, exemplifying synchronization of multiple remote display requests, and the potential for enhanced parallel simulations. We also describe how it may be used to define customized user interaction dialogs for input from an application, and bi-directional interaction between a user and an application.

* Research supported in part by NATO-CRG900108, NSF CCR-9102331, ONR-9310233, and ARO-93G0045.

1 Introduction

The methodology of graphical user interaction is well accepted as a useful and constructive modeling aid in simulation applications. In application-specific user interactions with a computation, an interface is typically made part of an application. In contrast to this, we propose the use of an application-independent visualization interaction system with potential for application-binding at any stage of the modeling process. Advantages of this approach include *ease of use* – specialized display knowledge is encapsulated in the system, *flexibility* – the system can be used for a variety of applications, including parallel and distributed simulations, *code reuse* – graphical interaction code need not be reconstructed for different applications, and *modularity* – visualization and interaction portions of the application can be layered on top of the application, facilitating design changes and code modification. Our ideas are manifest in the DISplay system, a graphical user interaction and display library, and associated servers which can be used with any sequential or parallel computation. DISplay was motivated by our studies in Distributed Interactive Simulation on heterogeneous networks [9].

The construction of a user-interface for a given application is a nontrivial task, requiring specialized skills. A good interface enhances the ease of use of a software product, often improving a user's productivity significantly. Further, the success of a software product often depends on the interface it provides to potential users. In developing a user interface for a given application, a designer chooses one of two standard approaches. One approach is to bind the coding of the interface with the coding of the application; the interface becomes an integral part of the application. Though parts of the the interface may be common to other applications, these parts must be recoded when used elsewhere. To get around this, the entire functionality of an interface can be embedded in a high-level library which provides an application developer a tool for rapid interface generation. This is the second approach and is found in tools like Motif [8] and Tk [16]. This tack has several advantages. Application developers require no knowledge of the workings of low levels graphics functions. Distinct applications may link to the same library, using a common interface to make distinct application-specific interfaces. Display and interaction routines contained in the library may be shared by different applications, also known as code-reuse in the software engineering community.

In a typical modeling study, an analyst (user) initially interacts with a simulation to provide input, e.g. by providing parameters or the name of a file containing parameters. As the simulation proceeds, it may be necessary to interact with the simulation and provide input to change its computational trajectory, based on its current trajectory. Such interaction with an executing model may take place at specific or at arbitrary points in the model, depending on the type of interaction required. It is generally useful to an analyst to have a continually updated status report on the progress of an executing model, either via textual displays, graphical displays, or an abstract display representing meaningful progress so that a run may be aborted when a computation goes awry. When a simulation run is done, an analyst usually wants output in the form of tables or graphs. The complexity of such user-interactions is significantly larger in the parallel simulation, and more generally, parallel computing domains. Simple sequential displays now

require synchronization between multiple (distributed) processes, so that textual or graphical results are presented to a user in a consistent manner, obeying causality constraints. Simple user-interface tools for sequential interactions and displays cannot cater to multiple interactions for distributed applications.

1.1 Visual Interactive Simulation

Questions concerning the provision of facilities for user interaction with a running simulation, or for the graphical display of simulations results in real-time are not new. Indeed, O'Keefe [14] discusses basic ideas underlying and a methodology for Visual Interactive Simulation (VIS), and Rooks [19] gives a proposal for VIS, outlining general requirements and potentially unifying framework. Requirements delineated in the latter proposal include

- **Intervention:** allowing the initiation of interaction with the simulation model. Modes of interaction are inspection, specification and visualization.
- **Inspection:** allowing access to all simulation data for reading and/or writing.
- **Specification:** allowing specification of model parameters.
- **Visualization:** allowing diverse visualizations of model data, so that model dynamics and relationships of interest can be illustrated.

In early literature on this subject, the term VIS [14] was used to include activities which are now generally regarded as components of Visual Interactive Modelling (VIM), where a computer model of the system to be simulated is created. More recently, researchers have attempted to separate and refine functions associated with VIS and VIM [19]. Details on VIM can be found in [1, 19]. In the rest of this paper, we focus our attention on VIS, with special emphasis on parallel and distributed simulation.

Early software systems supporting VIS were generally restricted to animations of application components of simulation models. Later, these systems added user interaction capabilities in various forms. More recent systems supporting VIS include WITNESS [23] — which also allows interactive model building, Cinema [7] and Arena [3] — which are used with the SIMAN simulation language, providing real-time or post-processed animation and building of a model, SIMSCRIPT II.5 [20] — a language which provides an integrated graphical interface called SIMGRAPHICS, and TESS [17, 15] — a system associated with the SLAM simulation language, with facilities for graphical model building, analyzing, graphing and animating model results.

In this paper we describe the DISplay system, which exhibits all of the requirements of VIS to a large degree. Our terminology follows that provided by Rooks [19]. Facilities for *dynamic visualization* of the progress of a simulation in the form of *abstract displays* like histograms and *representative displays* like networks are provided. The result is a system which provides a simple scheme for animating general simulations. For example, consider a queueing network depicted as a graph where nodes represent

servers. Changing colors on nodes can represent server status: `busy` or `idle`. User interactions may be embedded in program code, with complex user supplied and model specific dialogs. The system integrates the functions of display (for simulation variables) and the functions of dialogs (for simulation interaction), so that an analyst seeking to examine and change values can do so from a process initiating the computation. The software provides for both *user prompted* interaction as well as *model prompted* interaction, with potential for involving the user, the model, or both.

DISplay is implemented in C and C++. The library with which the application program is linked is entirely C-based. An analyst may code an application in a variety of languages, provided these can be interfaced with a C library. We have tested the DISplay tool in a number of different environments, including distributed computing environments such as PVM [22] and Conch [24], the *EcliPSe* parallel simulation environment [18], and the process-oriented simulation tool CSIM [21]. Here we describe two examples of its use — one with CSIM, and the other with Conch, which is a message-passing environment for a network of workstations. Use of DISplay for the display of real-time performance measures in general *EcliPSe*-based parallel simulations is described in [9]. DISplay has a graphical component that was constructed using OSF/Motif [8], in conjunction with the PEX [6] library for 3D graphs. To use DISplay, all that is required is a workstation hosting an X server, with an optional PEX compatibility (for 3D graphs).

The rest of the paper is organized as follows. Section 2 contains a description of the DISplay system architecture, our design goals and software requirements. In Section 3 we describe design aspects in some detail, also motivating our design decisions. Programing and user interfaces are addressed in Section 4, and demonstrative examples of DISplay's use are given in Section 5. We conclude in Section 6, discussing the contributions of this work, aspects of its limitations and ideas for future work.

2 The DISplay Architecture

The DISplay software architecture is based on the client-server paradigm [4]. Here, the simulation application, which is created by an analyst or programmer, is treated as a client. Client calls are made by invoking functions resident in the DISplay client library. The server portion of the architecture consists of a **connection server (CS)** to which clients connect, and a **DISplay server (DS)** which handles all graphics and user interaction requests from a distributed application. Typically, the client connects to the **CS** during application initialization; the **CS** delivers a handle (i.e., socket) to the application for communication with the **DS**. The **DS** is created by the **CS** using a unix *fork* operation, and reads incoming messages from the application, creating the necessary display *tasks* and interaction *dialogs*, as specified by the application. Continuing display and user interaction is based on application specifics and objectives. Messages between the application and the **DS** form a well-defined set of primitives, part of the DISplay protocol. The **DS** connects to an X workstation to execute specified display and user interaction commands. The analyst can interact with an executing application from the workstation where the display is done, thus enabling a requisite interaction capability between analyst and model. A picture depicting the overall architecture

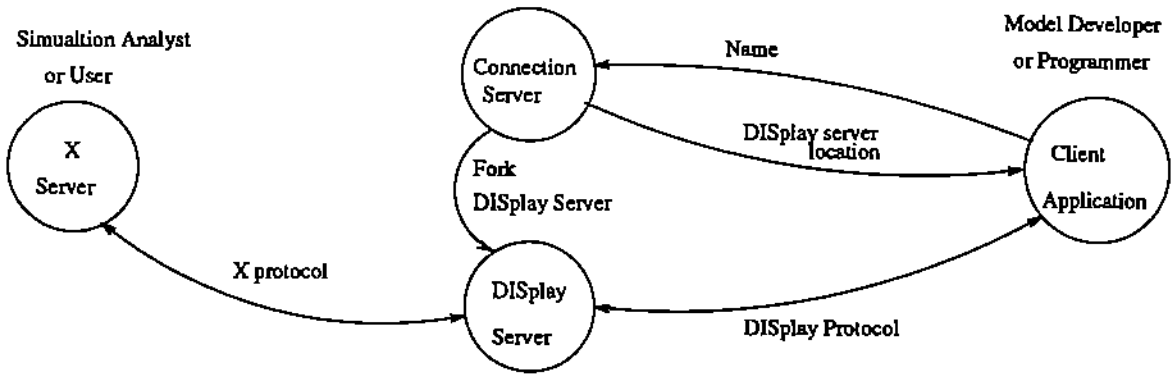


Figure 1: The Client Server Architecture

can be seen in Figure 1.

The CS performs an important function in the mechanism described above. It must be located on a well known machine and port, so that applications can readily connect to it. The connection is made using a unique Name, which identifies the application. The CS maintains a list containing each active DS, along with its associated Name. In a multiprocessor application, say using PVM, Conch, or some other system, several processes, possibly residing on distinct (distributed) processors, may request a single, cooperative graphical display and user interaction. In this case, the CS delivers the same DS handle to each of these processes (from the same application), so that they may all connect to the same DS. All executing processes may be on distinct and remote machines, including both the CS and the DS, which need not be located on the workstation where the display and user interaction is being done.

Once a DS has been created, an application interacts only with its DS, leaving the CS to continue to read connection requests from other applications. If the name specified in an incoming connection request is different from any of the current names in the CS database, it delivers the application a connection to a new DS. Otherwise, it delivers the application an existing connection. Such a scheme is the primary means for sharing a server among several cooperating processes, and is common in parallel computing environments.

2.1 Use of DISplay in Parallel Simulation Environments

There are several approaches to developing parallel discrete event simulations. Of these, the primary approaches are *conservative*, *optimistic* and *adaptive*. In the conservative approach events are processed strictly in order of occurrence, maintaining causality at all times. The approach is prone to deadlock, which may be avoided by resorting to the use of so-called *null messages*. In the optimistic approach, events are processed as they become available; potential causality conflicts are disregarded until a causality error is detected. Upon detection of such an error, invalidated simulation work is undone, the causality error corrected and the simulation re-executed from the point of correction. The approach requires some form of state-saving, so that simulation re-execution from a given point is possible. The adaptive approach is a

mixture of the optimistic and the conservative approaches. A summary of these approaches can be found in [5].

Without loss of generality, we assume that the simulation environment is made up of a cluster of workstations which work collectively to solve a problem. Each workstation may host a variable number of processes. The **DS** can be used in one of two ways in such a situation. One approach is to connect every simulation process to the **DS**. Doing this, however, may not be possible because the **DS** may run out of file descriptors if the number of processes is very large, since there is a limit on the number of open descriptors at any given time. An alternate approach is for one or few of the processes to collect messages from other processes and pass these on to the **DS**. This approach suffers the additional overhead incurred by the intermediary or intermediaries, if such message collection is not an inherent part of the executing parallel application. For each process in a given simulation that connects to the **DS**, the latter creates a virtual channel on which messages from the sending process are queued. It is assumed that messages along each channel are received in order of simulation time.

2.1.1 Synchronization

During a parallel simulation, distinct processors may be involved in computations associated with different virtual simulation times. Messages sent to the **DS** by different processors arrive at the **DS** with distinct virtual-time stamps. To provide the user with a consistent view of the simulated system, the **DS** buffers, sequences and processes messages only when it is certain that subsequent messages cannot induce causality violations. One way of achieving this is by adopting a conservative parallel simulation protocol in DISplay. Consider an example utilizing the conservative protocol. Each channel (source–destination link between processes, as viewed from the destination end) is associated with a channel time. This time is equal to the time-stamp of the first message queued on the channel buffer, if one exists, or the time-stamp of the last message retrieved from the channel buffer if the channel buffer is empty. At any given time, the first message from the buffer of the channel with the smallest channel time is selected for processing. The following description uses an example to simplify exposition.

In Figure 2 is shown a set of three processes, numbered 0,1, and 2, that connect to the same **DS**. The **DS** creates a virtual channel for each, using processor identifiers to distinguish between channels. Channel 0 has the smallest channel time, and so the message with time-stamp 25 in channel 0 is selected for processing. After this message is processed by the **DS**, further DISplay processing is temporarily halted because channel 0 becomes empty (with channel time still at 25), and a minimum time-stamp message cannot be identified. Now if, for example, another message arrives on channel 0 with a time-stamp of 27.5, channel time for channel 0 is updated to 27.5, and channel 2 wins as the channel with the smallest channel time. The message with time-stamp 27 on channel 2 is selected for processing, leaving channel 2 with a channel time of 28.5. As long as messages continue to arrive from processes, channel times can be updated, and a minimum time-stamp message selected for processing.

If a process has no messages to send, it avoids an indefinite delay of message selection at the DS

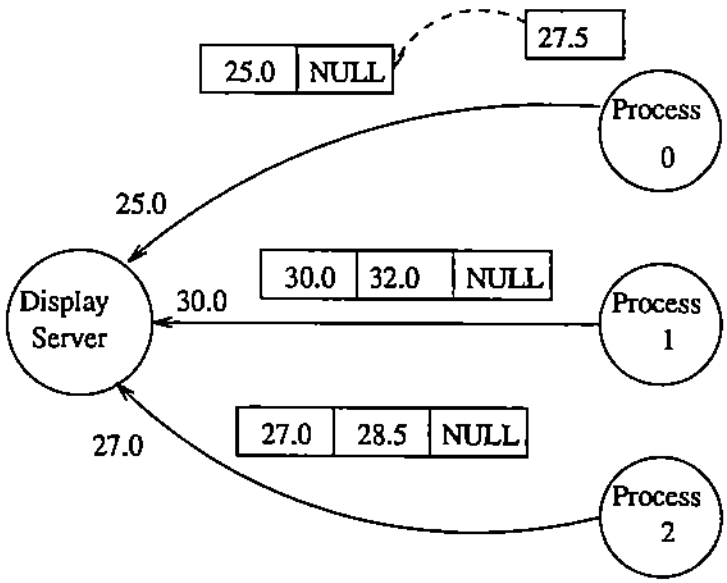


Figure 2: Synchronizing DISplay with the application

by forwarding a `Time Update` message, which is basically the equivalent of a null message in the conservative approach. In effect, the processor sending the message informs the **DS** that no message will arrive before a time specified in the `Time Update` message. Observe that the **DS** may also be used in an optimistic parallel simulation, in which case messages will not be sent to the **DS** until they are committed and cannot be rolled back. This occurs when the time-stamp of a message is smaller than a parallel simulation's global virtual time. In this case however, `Time Update` messages are not required but synchronization is necessary so that a consistent view of the simulation is provided to the user. An example of the use of interactive graphics in the optimistic simulation system SPEEDES is given in [25]. For applications that have no notion of time, it is feasible to send all messages with the same time-stamp of 0. However, requests must not be sent in wrong time order. If this occurs, there's a good chance that the application's execution logic is incorrect.

2.1.2 Resource Sharing

DISplay provides parallel applications with two basic resources, called *tasks* and *dialogs*. *Tasks* are used for displaying simulation output graphically, while *dialogs* enable user interaction. In a parallel execution environment, it is necessary for several processes to share a single resource at the server. DISplay permits *task* sharing between processes. For example, all computing processes may employ the same *task*, to perform displays in a single window. This is useful in domain-decomposition applications where processes work on different parts of a domain. Each process displays its contribution to an entire graphical result via a mapping to a relevant portion of the output screen. Such an approach can be useful when comparisons between processor results are required – distinct processes display results in the same window, and a

simple visual scan of the display allows for easy comparison.

2.2 Design Goals

The DISplay software system (servers and client library) is part of a larger development effort geared towards parallel computations and simulations. This effort is manifest in the *ACES* software architecture, a brief overview of which can be found in [9]. A first motivation was the need for a general mechanism to display data and perform user interaction in a software layer that made display and interaction functions independent of kernel and communication layers of the *ACES* system. A second motivation for the DISplay tool was the need to perform graphical debugging and performance monitoring of the system's nontrivial function-interactions, both within and between layers. Facilities for debugging and monitoring is currently absent in DISplay, but we expect to include this functionality as the project matures.

While additional goals may be added as the system evolves, the basic goals of DISplay at present are listed as:

1. **Simplicity:** provide the user with a simple means to collect and view results from parallel simulations. This should relieve the user of the responsibility of making and maintaining connections and creating displays.

2. **Extensibility:** provide an open library in that new functionality can be added with relative ease.

3. **Interaction:** provide complete interaction, so that the application can send data to the user, and the user can send data to the application. This reciprocal action allows the application to take in input, give the user simulation output and also allows the user to change simulation variables dynamically, during a run.

4. **Performance Monitoring:** provide a means to visually depict the status of a simulation at each process in a parallel simulation, so as to be able to identify critical sections or bottlenecks in the simulation.

5. **Generality:** provide as general an interface as possible, so that the system can be used with general parallel applications.

6. **Portability:** provide software that is minimally dependent on particulars of machines, that is inter-operable so that ports to a variety of platforms is possible.

The DISplay software system meets the goals listed above to some degree. For example, we have emphasized simplicity in the design of the interface. All complex functions related to graphics and data communication are hidden from the user – library function-calls enable various interactions with graphics support. The system is extensible – a simple interface allows for the addition of new visualization tools. Since software for the DS is based on the object-oriented paradigm, newer or more sophisticated graphical

displays may be built on top of already existing displays. Also, hooks provided in the software enable addition of new displays. Both user-initiated interaction as well as application-initiated interaction are allowed. A user may develop his/her own special interaction *dialog forms* without making changes to the DISplay software. Though motivated by distributed simulation, DISplay can be used with any parallel application. For example, it may be possible to use the tool with mathematical software to display the progress of a computation which solves a linear system of equations in parallel. Because the tool is built on the widely available and de-facto standard technology of the X protocol and C, it is highly portable.

3 The Display Server

The power of display server (DS) lies in its generality – because it does not store application-specific state. It is application-independent. The DS accepts information from an application for the purpose of display, and it is up to the user to make decisions on types of displays and types of interactions desired with an application. The DS presents information to the user in a well-defined manner, accepts input which it forwards to the application – if the application requires this input, and displays results – if the application requests such a display. All interpretations of presentation are left to the user. Functions that the DS provides execute independently of the application. For example, clicking on a button to display a graph will not affect the executing application. This function is handled at the DS itself. Since the DS is X-windows based, it provides the familiar windows, icons, menus, and pointing devices interface. It provides a point and click interface, for function invocation. These invocations may occur randomly, e.g. windows may be resized, uncovered, or iconified/deiconified, buttons that invoke user interactions or display windows may be clicked on.

The DISplay system provides *tasks* and *dialogs* as basic mechanisms for implementing graphical displays and user interactions, respectively. *Tasks* are of two types: `Single message tasks` require a single message to be delivered, and their action is taken based on that message alone. `Multiple message tasks` require a setup phase, where the task is created locally and then executed at the server. The creation mechanism returns a task identifier which is used as a handle in all subsequent messages related to the task. After task creation, multiple task related messages may be sent to the DS to carry out the intended function. These messages are interpreted in the context of the task identifier. Tasks are deleted by using an `EndTask` message. The use of a task identifier provides for some useful performance improvements. For example, it is not necessary to send information repeatedly saying that a plot is 2D or 3D, with every message that is associated with this plot. This information is stored at the DS at task creation time, and a request for plotting a point is executed based on whether the plot is setup for 2D or 3D.

User interactions are also handled in a similar manner. Each possible user-model interaction is assigned a unique dialog identifier, with a *form* based dialog created. A dialog setup phase creates the dialog locally and then sets it up at the DS. Queries and answers using these dialogs are interpreted based on the dialog identifier and the position of the data within a dialog. It is possible to link dialogs to specific tasks, and to

have tasks linked to specific dialogs. For example, the node or arc of a graph task can be associated with dialogs. Also, when a dialog is activated it may simply display an active task like a plot of points.

The DS repeatedly reads messages (including requests for new connections from clients) on incoming channels and acts on them. Messages that are used for setup and termination are identified as `control` messages, and others are referred to as `action` messages. Each message has a header which contains the type of message, virtual time, task identifier, a process identifier for the sending process, the simulation identifier (which depends on name of the simulation, and is assigned by the CS), and an operation code. If the message is of a `control` type, action is taken immediately provided the operation code in the message is *urgent* or specifies the creation of a new process. Otherwise, the message is queued and is acted on in correct time order. Queued messages are processed by a queued message handler which invokes the appropriate handling routine, based on message type and other relevant information contained in the message, e.g., task identifier, dialog identifier. Commands to display output or query the user are then executed. Details on these aspects of the system can be found in [10].

3.1 DISplay Classes

A central class, called the `controller` class, keeps track of all connected processes and their associated channels. It has a predefined maximum number of channels which are setup as each process connects to the DS. Each `channel` class records the socket over which its communication takes place, channel virtual-time, and a list of tasks and dialogs indexed by identifiers. Messages received on the channel are queued in FIFO mode. Other important classes in the system include the `task` and `dialog` classes.

3.1.1 Tasks

DISplay *Tasks* are used for any output requiring multiple messages. *Tasks* form an extensible set of textual, graphical and representational schemes for displaying the dynamics of a simulation and its end results. The C++ class hierarchy shown in Figure 3 describes the arrangement of tasks. The `Task` class provides control functions for maintaining the status of a task. A task may need to be displayed even after the simulation is complete. Thus, the display does not disappear when the simulation terminates, and may correspond to the final simulation result. When an `EndTask` control message is received from a client, the task is marked for deletion. It is deleted only when the user selects the *Cancel* button from the task window on the screen. When new task classes are added, these may utilize the functionality provided by the base task classes, such as the `Task` class. This facilitates extension of the interface with newer displays. The `ShellTask` class provides for the popping up and down of the window in which the display is drawn. `XYTask` is used for all 2D tasks that are drawn to scale. `PexTask` contains basic data structures for use with tasks that use the PEXlib 3D library. At the leaves are the actual classes that a user must be aware of, though it is never necessary to program directly with these.

The classes marked * in figure 3 are to be added in the near future. For example, a provision for the

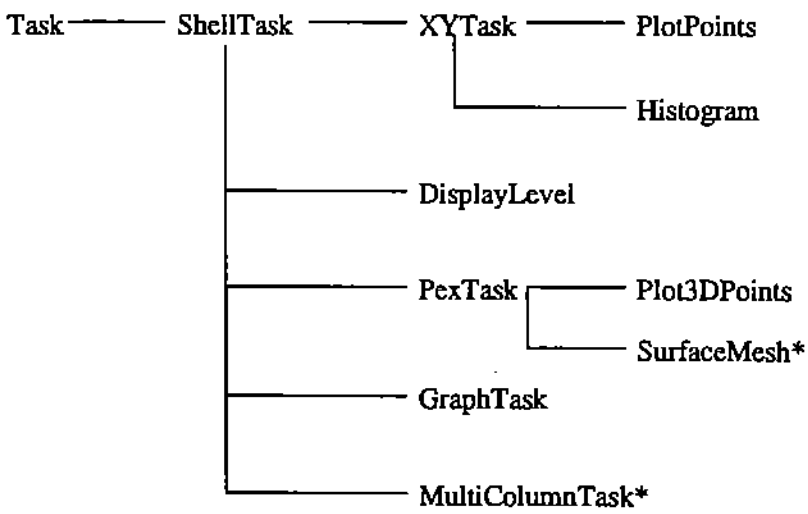


Figure 3: The C++ Task Hierarchy

display of multicolumn tables is planned. This may easily be sub-classed from the `ShellTask` class. Adding a new task means that the DISplay protocol must be expanded to include the new task message. Hooks must be provided in existing DS functions so that control may be passed to functions which will act upon these messages. A new set of functions pertaining to the new functionality must also be added to the client library so that the simulation analyst can make use of the new task.

3.1.2 Dialogs

DISplay *dialogs* are basic mechanisms for user interaction. A *dialog* is defined as a *form* consisting of a set of values. Associated with each value is a set of items which describe that value. In the current implementation, this set of items is limited to a textual description of the item and a specification describing the type of the value. The textual description can also be used as a query to the user to input a value of the required type. Other qualifiers may be added to the existing set. For example, in order to do validation of user input, a valid range for the values may be provided. Several related dialogs may be made part of a single *panel* from which the user can point and click on the label of the particular dialog that is being invoked. On being invoked, the current state of the dialog is displayed. Dialogs allow for bi-directional interaction between the application and the user.

Dialogs are of four types (see Table 1), depending on the direction in which interaction is allowed or restricted, and whether the dialog is invoked by the model or the user.

1. **User Query Dialog.** The request for value information is sent from the DS to the application. The application handles the query and returns the required values. Invocation is done by the user, who may fill in some of the values in the dialog. The application may use these in computation, returning computed values.

Interaction	Dialog Invocation	
	Model Prompted (Synchronous)	*User Prompted* (Asynchronous)
Two Way	Model Query Dialogs	User Query Dialogs
One Way	Model Display Dialogs	User Command Dialogs

Table 1: Classification of DISplay Dialogs

2. **User Command Dialog.** The user sends information to the application asynchronously. The application must be willing to handle these dialog messages through appropriate user written handlers which are registered with the dialog on creation.

3. **Model Query Dialog.** This functionality is similar to remote calls. The application pops up the dialog at the **DS** at predefined points in its execution, and waits for the user to reply. The application remains blocked until the user reply is received.

4. **Model Display Dialog.** Information is passed to the **DS** at predefined points in the application execution. The **DS** collects the information and displays it only when the user invokes the dialog (by clicking on the associated button). The information is received only when the application chooses to send. A special form of this dialog is one associated with a task. The associated task is popped up when the panel button is clicked.

It is the developer's responsibility to provide for functions called *dialog handlers* in the application. These must handle replies or queries from the user and are registered when the dialog is created. They are called automatically when a dialog message of the given type is received. If asynchronous dialogs are used, the developer must arrange to call a library-provided function from time to time, to read dialog messages from the **DS** and call application dialog handlers. An example given in Section 5 shows how this may be done. Since all messages are logged, it is possible to do a post-run determination of times of interaction between user and application. This may be important in performance-tuning and analysis of simulations.

4 Programming and User Interfaces

4.1 The Programming Interface

There are two simple approaches to using DISplay with an application. One approach is to first construct the simulation application, without regard to DISplay interaction. Calls to the **DS** can be added when the working simulation is available. With this approach, the user cannot avail of development support and debugging help (i.e., display of run-time status, values of simulation variables and simulation object

states) provided by DISplay. On the other hand, if parallel debuggers are available, they may compensate for or even offer services superior to DISplay's debugging services. Good parallel debuggers, however, are still years away. Given a working simulation, client calls may be added wherever necessary in the application code.

An alternate approach, which we recommend, is to create the simulation application *with* provision for client calls in the application design. In testing simulation logic, the **DS** may be disabled – either by not initiating a connection to the **CS** or by setting the value of the handle returned by the `sGetServer()` call to **SDERROR**. This constant is defined in an include file **sdconsts.h**, which must be included along with header file **sdclnt.h** in all programs that make calls to the **DS**. With the handle thus set, send-data calls to the server return immediately, with no effect on the simulation. The first example described in the following section was developed using the first approach simply because working simulation code was already available. The second example described was developed using the second approach; the **DS** was disabled until the application was tested.

To utilize **DS** functions the analyst must invoke functions in the DISplay Client Library (DCL) `libds.a`, which must be linked with the application. The DCL consists of C functions – so it can be used with any application that can be interfaced with the C language. All DISplay functions return appropriate error indicators, and functions that fail return SDERROR. For example, the availability of the **DS** is determined by examining the value returned by `sGetServer()`. Subsequent calls to the DCL may first ascertain the availability of the DS by testing this value. This allows the simulation to execute even when the **DS** is unavailable. If interactive use of the simulation is to be replaced by batch use, the analyst simply needs to set the window parameter in setup calls to FALSE. Of course, this will work only if synchronous dialogs (which require user replies) are not built into the application, or are replaced by calls to read from a file depending on the value of a switch. Logging is also allowed in batch-mode, with messages to the **DS** logged in DISplay message format, to be replayed at a later time using a DISplay utility program.

Client functions are of two types: functions which send messages to the **DS** and functions which perform local processing prior to sending a message. Functions of the former type begin with an `s` prefix, while functions of the latter type begin with an `sd` prefix. The DCL contains a host of basic functions, with several convenience functions that invoke basic functions. An analyst may call either basic or convenience functions. The latter are simpler to use and recommended, whenever possible. At present, the basic function set consists of the 34 functions listed in Table 2. These are classified according to functionality. The first two arguments to all message-sending functions consist of a socket identifier and simulation time. The third argument is usually a task identifier or a dialog identifier. In Figure 4 is shown an example with typical use of functions to connect to the **DS**. In setting up the connection, it is possible to specify whether logging is required, and whether a new main window is required. By specifying the number of processes in the simulation, all processes are guaranteed connection before the **DS** begins to act on messages from the simulation.

Once the connection is made, tasks may be created, initiated and terminated at any time. In Figure 5

Function	*Remote*	*Description*
Server Control Functions		
sGetServer()	Yes	Obtain a handle to the **DS**
sNewProcessMsg()	Yes	Identify itself as a process and request window setup and logging, if required.
sEndServer()	Yes	Close the connection to the **DS**
Single message Task Functions		
sNullMsg()	Yes	Send an empty message and update channel time
sStrPrintMsg()	Yes	Format and print the message on remote window
Multi message Task Control Functions		
sdCreateTask()	No	Set up a task and return a handle
sdSetTaskValues()	No	Set up specific parameters for the task
sdGetTaskType()	No	Get the type of task
sBeginTask()	Yes	Set up the task at the **DS**
sEndTask()	Yes	End the task at the **DS**
Multi message Task Functions		
sdCreateSdPoints()	No	Create a set of points to be filled in
sdFreeSdPoints()	No	Free the set of points created by sdCreateSdPoints()
sdAddSdPoint()	No	Add a point to the set created by sdCreateSdPoints()
sdFreqHistPoint()	No	Return the histogram frequency for a given X value
sColPoint()	Yes	Color a point with a particular color
sPltPoint()	Yes	Plot a line from the previous point to this point in a particular color
sHistPoint()	Yes	Draw the line bar in the Histogram
sRestartPlot()	Yes	Begin drawing a line plot from a different point
sDisplayLevel()	Yes	Display the level in a level-plot
Graph message Functions		
sGAddNode()	Yes	Add a node to the graph
sGAddArc()	Yes	Add an arc to the graph
sGDeleteNode()	Yes	Delete a existing node from the graph
sGDeleteArc()	Yes	Delete an existing arc from the graph
sGChangeVal()	Yes	Change some display parameter of a node or arc
Dialog Control Functions		
sdCreateDialog()	No	Create a dialog
sdDeleteDialog()	No	Delete a dialog when it is no longer required
sBeginDialog()	Yes	Set up a created dialog at the **DS**
sChangeDialog()	Yes	Change the parameters of the dialog
Dialog Handler Functions		
sdCreateReply()	No	Prepare a reply to send to the **DS**
sdAddToReply()	No	Add an answer in the reply created
sdFreeReply()	No	Free reply created by sdCreateReply()
sQueryReply()	Yes	Send a query to the User and get reply
sHandleReply()	Yes	Read user command and handle it in the application
sReplyDispMsg()	Yes	Send a reply to a User Query message

Table 2: Basic Functions used with the Display Server

```
main(int argc, char **argv)
 {
      char *simname;              /* the name of the simulation */
      char *host;                 /* the location of the connection server */
      char *service;              /* the port number of the connection server */
      int  sock;                  /* the handle to the DS */
      int  ret;                   /* return code from functions */
      int window=TRUE, log=TRUE; /* window up and logging on */
      int num_of_procs;           /* number or processes in the simulation */

      ...;

      sock = sGetServer(simname, host, service); /* make the connection */
      if (sock == SDERROR) {
       /* handle the error here */
      }
      /* register this process as one of num_of_procs */
      ret = sNewProcessMsg(sock, window, log, num_of_procs);

      /* rest of processing and messages to DS */

      ret = sEndServer(sock);         /* close the connection */
 }
```

Figure 4: Example of connecting to the Display Server

is shown the use of functions for creating a Histogram task (`histtask`) and two dialogs for a particle simulation. The first dialog (`dialog_task`) is associated with the newly created task `histtask`. The second dialog (`dialog_sync`) is a synchronous dialog which is invoked using the `sQueryReply()` call. The `global` argument to function `sdCreateTask()` ensures that different processes will cooperate to draw in the same window. For example, in a particle dynamics application (such as described in the following section) the domain is set up as a 2D grid, divided into equal-sized slices among processes. Though processors compute on distinct domains, it is possible to display the particle movement on the entire grid in a single window. All processes write into a globally known window, each writing onto only that portion of the window for which it is responsible, which may indeed be the entire window. This is preferable to the situation where processes create and display particle movement in distinct windows – this does not aid result interpretation and analysis, especially when the number of windows is large.

The functions which do the brunt of the drawing work in task windows are listed as `Multi Message Task Functions` and `Graph Message Functions` in Table 2. These functions take in a task identifier as a parameter, to identify the task to which they must refer on the **DS** side. Important functions for drawing in a window include `sColPoint()`, `sPltPoint()`, `sHistPoint()`, and `sDisplayLevel()` – capable of 2D and 3D drawings. Function `sColPoint()` colors a point in the window with a specified color. The analyst programs with world coordinates and color names. The DS converts world coordinates into corresponding window coordinates, and maps colors to pixel values that can be displayed on the specified X workstation. Function `sPltPoint()` plots a line between two points. The corresponding convenience function for performing 2D plots is `sPltPoint2D()`. Function

```
#include "sdconsts.h"
 #include "sdclnt.h"
 #include "sdutil.h"
  {
       /* dialog */
       static SdQtnAns qtnans_sync[] = {
            {"What is your name?", SDSTRING},
            {"What is your key?", SDINT},
       };
       char *reply[2];
       double timereply=0.0;
       int i;

       int histtask;
       int dialog_task, dialog_sync;
       int global = 1;           /* is a global task */

       /* create and initiate a Histogram task */
       histtask = sdCreateTask(HISTTASK, window, log, global);
       sdSetTaskValues(histtask,
                       SdXaxisName, "X- axis",
                       SdYaxisName, "Y- axis",
                       SdTitle, "Particle simulation",
                       SdXmin, 0.0,
                       SdXmax, 20.0
                       SdYmin, 0.0,
                       SdYmax, 20.0
                       SdNoPoints, 5.0,
                       SdDim, 2,
                       SdXaxisInterval, 1,
                       NULL);
       sBeginTask(sock, simtime, histtask);

       /* create the task dialog */
       dialog_task = sdCreateDialogTask("Histogram", histtask);
       /* create the Synchronous dialog */
       dialog_sync = sdCreateDialogNotask("PassKey", SDMODEL_QUERY,
                                          Number(qtnans_sync), qtnans_sync,
                                          NULL);

       /* begin the dialogs grouping the two into a single pane*/
       sBeginDialog(sock, simtime, dialog_task, dialog_sync, "Some Dialogs");

       /* invoke the synchronous dialog */
       ret = sQueryReply(sock, &timereply, dialog_sync, reply);

       /* print the replies */
       for (i = 0; i < Number(qtnans_sync); i++) {
            printf("Qtn:%s Ans:%s\n",qtnans_sync[i].question,
                   reply[i]);
       }

  }
```

Figure 5: Use of Task and Dialog Functions

`sHistPoint()` places a histogram line on the task window, and function `sDisplayLevel()` displays the specified level of a meter or variable. With the aid of such functions, diverse abstract displays like scatter plots, line plots, histograms, x-y-z plots, and level indicators may be shown. Some of these displays are *instantaneous* displays (the display shows the status of the simulation at specific points of time), whereas other displays are *cumulative*, in that users are presented with a history of information (for example, a x-y plot task may display a continuous change of the value of a simulation variable plotted against time).

DISplay allows representational displays to be created using networks. Networks can be used to represent displays for a variety of physical systems. In queueing systems graph nodes may represent servers, with arcs representing possible customer routing between servers. In most cases some form of visual representation of the physical object is used for a graph node. The default is to use a rectangular box with a label. It is possible to associate dialogs with nodes and arcs. Clicking on a node or arc with which a dialog is associated, will cause the associated dialog to execute at the DS. For example, in an *EcliPSe* simulation, processors arrange themselves in a virtual tree-topology [13] to perform parallel sampling. This underlying tree configuration of the machines can be displayed by DISplay's Graph Task(see Figure 6). With each node is associated a dialog that displays performance data for that node, causing it to pop up when that particular node is clicked. Moreover, the graph can be modified dynamically. Nodes and arcs may be added or deleted. The colors of the nodes and arcs may be changed to signify a change in the value associated with the node (e.g., CPU load level) or arc (e.g., amount of traffic).

4.2 The User Interface

The DS provides the analyst with a user interface that can be tailored to suit the application. This graphical interface is common to all DISplay applications. For example, a sample screen dump of an *EcliPSe* [13, 9] Performance Monitoring display is shown in Figure 6. This is a particular use of DISplay for *EcliPSe* applications. In the figure is shown an example of a network (GRAPHTASK), a histogram task and several asynchronous dialogs. The interface consists of the *Interaction Push-buttons* which, when clicked, cause *Panels* to pop up. Each panel encapsulates a group of associated dialogs or displays which may be invoked through use of the panel buttons. Button labels clearly indicate button functions. Each interaction push-button displayed on the main menu bar of the user interface is application specific. To modify a user-presentation (for example, to add another panel option) it is necessary to incorporate the option in the program code and recompile the user's application. Thus creating application specific interfaces does not require the client library or the server software to be modified. Task windows open up whenever a Task is initiated in the application. The X window system allows the user to arrange multiple windows on the screen, as required. The DS also handles redraw and resize commands independently of the application. Scale sizes and limits may also be changed interactively and the complete window is redrawn with the new sizes and limits. Windows can be iconified and deiconified as necessary, and dialogs can be invoked to display data and closed when not required.

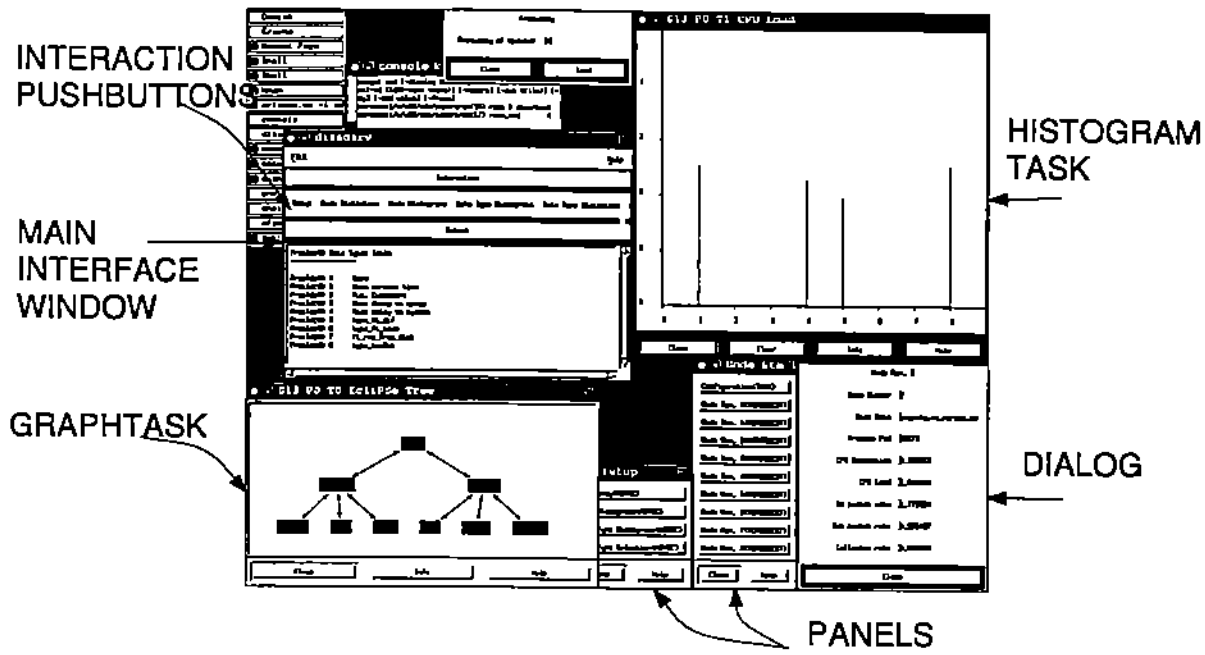


Figure 6: The DisplA User Interface

The DISplay system may be used online as well as off-line. A typical online use of the system proceeds as follows. On Unix based machines the CS is started using the command `consdsrv port server-port`. The CS accepts connections from port `port` and doles out DS connections to clients, starting with port `server-port`. To bring up a display on a machine distinct from the one hosting the application, the target (i.e., X server host) machine's name must be appended to the simulation name. For example, suppose that the application is hosted by machine `ariadne.cs.purdue.edu`, and a display for an `M/M/n` application is required on machine `carcassi.cs.purdue.edu`. The required specification is `M/M/n -display carcassi:0`. This suggests that it is preferable for the application to read the *simulation name* off a file, or accept it from a terminal, so that the name of the workstation where the user interface is displayed may be changed easily.

To run the DISplay software off-line, the logging feature must be turned on, and the display of windows turned off, via the parameters provided in the `sNewProcessMsg()` function. Later, logged messages may be input to the `dissdlog` utility, which accepts the name of the log file as a command-line argument. This program reads in logged messages and replays the simulation run using the log. All synchronization is done exactly as in the actual run. This feature is useful as a demonstration tool when the dynamics of some application must be described to a learning audience.

5 Examples of DISplay Use

The DISplay system software is useful in a variety of situations including product demonstration, gaming, learning, modeling, performance monitoring, parallel debugging, etc. The use of the DS for monitoring of parallel applications in *EcliPSe* is described in [9]. Here we focus on simple examples demonstrating the use of DISplay in general simulations. The first example is that of an `M/M/n` queueing system. This example shows how user interaction facilities and basic displays are used. Results are displayed using line plots and histograms.

It is instructive to note that the user can dynamically alter simulation variables such as customer arrival rate and/or number of servers, to examine system behavior under such changes. The second example involves a simple parallel simulation in particle physics, where particles move about randomly within a 2D grid. What is of interest to the analyst is the dynamics of particle movement and final particle positions. In this example we employ the *Ariadne* light-weight process library [11] and the *Conch* distributed computing environment [24]. All computing processes work on distinct portions (slices) of the 2D grid, cooperatively sending data to the DS in order to compose a single window display. Each process displays on a specific part of the screen window. Three different synchronization schemes have been implemented, with the performance of each measured and displayed using DISplay.

5.1 M/M/n Queueing System Simulation

An M/M/n queue is an n-server queue with Markovian arrivals and services. Customers queue in FIFO mode in a single queue, awaiting service. In this example we fix the mean service time μ and examine the effect of arrival rate λ and number of servers n on the measured responses of mean queue size, actual queue size and server utilization. The M/M/n application was developed using the process-oriented simulation language CSIM [21]. The multi-server facility is implemented using the `facility` type in CSIM. This simulation tool also provides functions for computing queue size, mean queue size and utilization, all of which were used to collect data that was subsequently sent to the DS.

In this example application, a user interface was rapidly generated using three dialogs. One was a synchronous, model-prompted dialog, where the application blocks and queries the user to enter input parameters (i.e., λ, μ, n and total number of customers to be simulated). Once these parameters are initialized, the application continues to execute. The other two dialogs are asynchronous, user-command dialogs. These allow the user to dynamically alter λ and n. New values for these parameters are sent to the application, which echoes back both new and old values to the user interface. Appropriate handlers must be provided in the application to act on these parameter changes.

To display the state of the system three displays are used. The first display depicts the utilization level of the multi-server facility. The second display shows both the mean queue size and the actual queue size on a single graph. The third display shows a queue size histogram, made by updating samples on each customer departure from the facility.

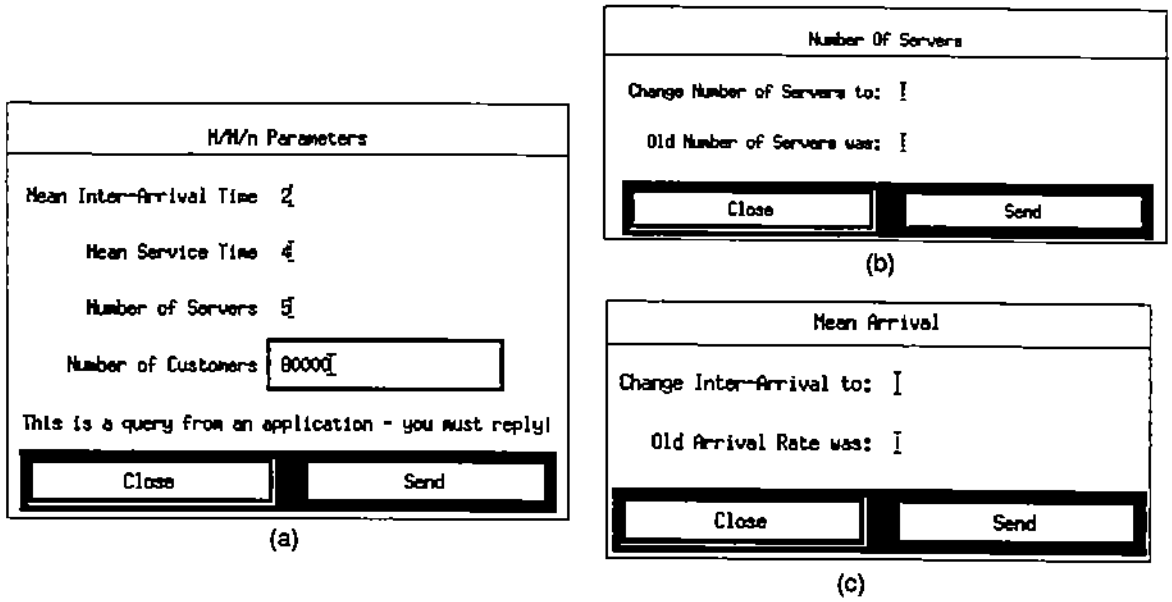


Figure 7: Dialogs for the M/M/n Server System

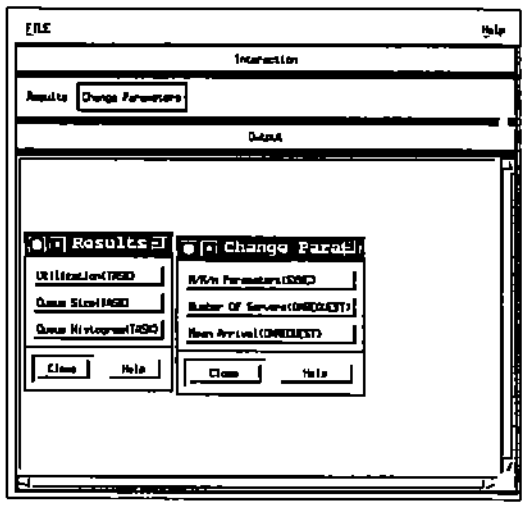
The different dialogs just described are shown in Figure 7, with the synchronous dialog displayed in Figure 7(a). The code segment shown in Figure 8 shows how this dialog is set up and used. Each entry in the dialog form has a corresponding entry in the `SdQtnAns` structure. The dialog is set up locally using `sdCreateDialog()`, with `SdQtnAns` passed to the latter as a parameter. Dialog initiation at the **DS** is accomplished with `sBeginDialog()`, and invocation via the function `sQueryReply()`.

Asynchronous dialogs are set up locally and initiated at the **DS** in the same manner. But in contrast to synchronous dialogs, these are initiated when the user clicks on specific buttons that appear in the user interface window. Appropriate handlers and calls to read messages from the **DS** must be provided. In the application, the analyst must call the DCL function `sHandleReply()` periodically, to look for queries or commands from the user, with appropriate handlers invoked in response. An example of a handler (`handle_arrival_change()`) whose function is to change mean inter-arrival time on user input is shown in Figure 8. This example shows how application-specific dialogs are constructed and used; dialogs can also be changed with ease, if such a change is required. Dialog handling code is easily separated from the main part of the application through use of handlers that are automatically called when a dialog is invoked by the user.

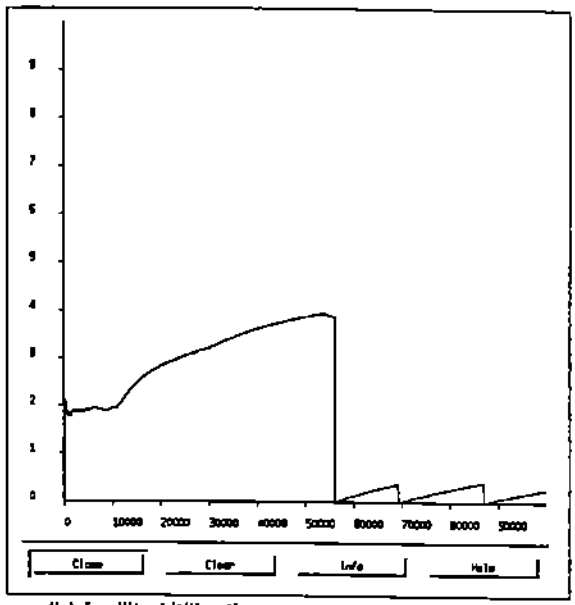
Figure 9 contains a user interface window for the queueing application, and also windows which display simulation results dynamically during model execution. In Figure 9(a) is shown the main user interface window, with two interface push-buttons corresponding to `Results` and `Change Parameters`. By clicking on these buttons, the user obtains panes with dialog-activating push-buttons. These panes are shown in the `Output` portion of the window in Figure 9. Results are displayed in abstract form, often with plots and histograms. For example, in the queueing application, model parameters are initially set as

```
int sock;                                        /* socket for communication */
int iatm;                                       /* the mean inter-arrival time */
static SdQtnAns dialog_parm[] = {              /* parameters of the simulation */
     {"Mean Inter-Arrival Time", SDFLOAT},
     {"Mean Service Time", SDFLOAT},
     {"Number of Servers", SDINT},
     {"Number of Customers", SDINT},
};

static int parm_dialog_id, arr_dialog_id;              /* dialog identifiers */

int ds_setup()                                         /* demonstrates setup */
{
     ...;
     parm_dialog_id = sdCreateDialogNotask("M/M/n Parameters", SDMODEL_QUERY,
                                           Number(dialog_parm), dialog_parm,
                                           NULL);
     ...;
     sBeginDialog(sock, clock, parm_dialog_id, arr_dialog_id,
                  "Change Parameters");
     return(sock);
}

get_parameters(int sock)          /* demonstrates use of Synchronous dialog */
{
     char *reply[4];                                  /* replies placed here */
     double dbl, timereply;
     int ivl;

     timereply = clock;
     sQueryReply(sock, &timereply, parm_dialog_id, reply);     /* query user */
     if ((dbl = atof(reply[0])) > 0.0)
          iatm = dbl;                                      /* Inter-arrival Time */
     if ((dbl = atof(reply[1])) > 0.0)
          svtm = dbl;                                           /* service time */
     if ((ivl = atoi(reply[2])) > 0)
          ns = ivl;                                         /* number of servers */
     if ((ivl = atoi(reply[3])) > 0)
          nars = ivl;                                      /* number of customers */
     sStrPrintMsg(sock, clock, "M/M/n parameters");
     sStrPrintMsg(sock, clock, "Service Time %f Inter-Arrival %f Servers %d",
                  svtm, iatm, ns);
}

int handle_arrival_change(char *reply[])       /* handle mean inter-arrival  */
{                                               /* reply contains user input */
     float new_iatm;
     char **send_reply;

     if (((new_iatm = (float)atof(reply[0])) == 0) || (new_iatm == iatm))
          return(0);
     wait(event_list_empty);
     sStrPrintMsg(sock, clock,
                  "Changing inter-arrival time at time %f\n", clock);
     send_reply = sdCreateReply(arr_dialog_id);           /* create a reply */
     sdAddToReply(arr_dialog_id, 0, new_iatm);          /* place reply here */
     sdAddToReply(arr_dialog_id, 1, iatm);
     iatm = new_iatm;
     sReplyDispMsg(sock, clock, arr_dialog_id, send_reply);/* update screen */
     return(1);
}
```

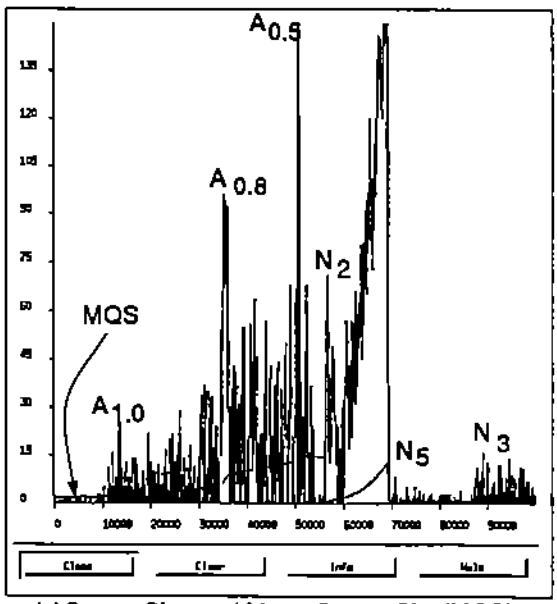
Figure 8: Handler and Code for Dialogs in a CSIM program



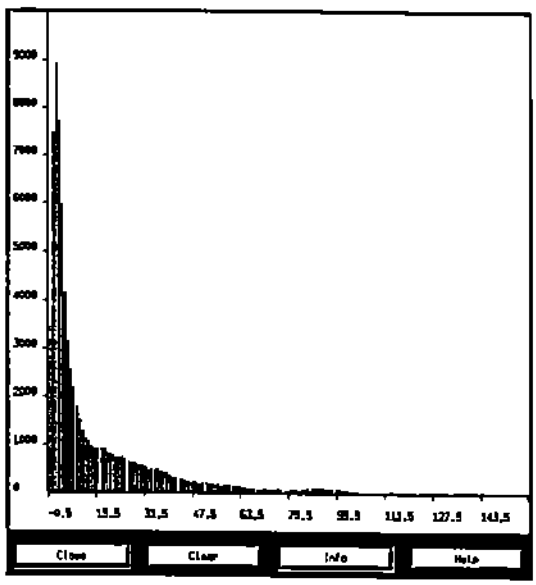
(a) Main Interface Window



(b) facility Utilization



(c)Queue Size and Mean Queue Size(MQS)



(d) Queue Size Histogram

Figure 9: User Interface Window and Results

depicted in the dialog given in Figure 7(a). In Figure 9(b) can be seen a simple plot of facility utilization versus time. In Figure 9(c) is shown a realization of queue size as well as mean queue size, with both graphed against time.

The graphs show that utilization can be increased, during a simulation run, by interactively changing the arrival rate λ; this also results in a larger mean queue size. Facility utilization is seen to fall to zero at certain times. These times coincide with the times at which the number of servers in the CSIM facility was changed. For example the points labelled A in Figure 9(c) correspond to points where λ was changed, and the points labelled N correspond to points at which the number of servers was changed. Subscripts on each symbol give the new value, after the change.

In Figure 9 is shown a histogram of queue size. This is a visual representation of a `qhistogram` facility provided by CSIM, the difference being that the histogram was computed dynamically using DISplay's support functions. Points at which input parameters were changed are logged and displayed on the user interface window via calls to the DISplay function `sStrPrintMsg()`. This example demonstrates how simulation parameters can be tweaked interactively to obtain changes in the behavior of the simulation.

5.2 Particle Dynamics on a 2D Lattice

Particle dynamics studies are of considerable interest in the fields of physics and materials science. Examples where their study is useful to name just a couple, are fluid flows in porous media and electrical conduction. These phenomena are often modeled as *random walks* on disordered clusters. The model described below is in essence the one known as the *ant in the labyrinth* and is due to deGennes [12]. The *random walker* (ant) moves at random only on certain accessible sites of a lattice, where the *fraction* of accessible sites on the lattice is q, $0 < q < 1$. We simulate the movement of the ant for T time-steps and then compute the mean square displacement of the ant. The goal is to find the relation between this displacement and the values of q and T. Details on this model can be found in [12].

This example shows how the DISplay system can be used with distributed or parallel applications. Here, DISplay was valuable in verifying program correctness through visual examination of program results, and also in monitoring the performance of different algorithms. The purpose of the application is to study particle movement on a 2D lattice. Initially, a small percentage ($(1 - q) = 2\%$) of all grid points are labeled as inaccessible. Particles are then assigned to a maximum of 10% of the remaining lattice, with positions assigned randomly. The application proceeds in a sequence of *time-steps*. All particles are visited in some sequence, and given a chance to move to a randomly chosen neighboring lattice point in a single time-step. For such a move to be successful, two conditions must hold. First, the destination point should be accessible to the particle (i.e., it should not be labeled inaccessible). Second, the point should not already be host to another particle. If either condition is false, the particle remains where it is until it can make another attempt to move in the next phase. Wrap-around is used to handle particles that move across a boundary. Of interest is the relation of the mean square displacement of the particle (averaged

over several runs) to q and T.

As described above, the application is simple to code on a uniprocessor. After creating a grid to represent the lattice and marking sites as inaccessible, a sequence of time-steps ensues. At each time-step, each particle is visited in turn, with random numbers used to obtain direction of movement. Depending on the satisfiability of the two conditions described above, a particle may or may not change its position. In any case, all necessary information is available to the single host processor locally.

Porting the application to a multiprocessor or distributed system is not difficult, but requires some care. A good way to parallelize the application is to place portions of the lattice on distinct processors, so that work can be shared. The lattice is divided into slices, with each slice (and thus all particles on the slice) assigned to a distinct processor. Processors work on their slices independently, marking sites as inaccessible and generating particles. Each processor need only be aware of the status of the slice that it has been assigned. Because each processor does not have all the information it needs for independent simulation (since events in a time-step may involve particles at other processors, and particle movement between slices or across boundaries requires between-processor information), some form of processor synchronization is required. This synchronization must occur after every time-step so that particles that migrate from one processor to another (across slices or across the lattice boundary) do not arrive at a destination processor *late* in simulation time. Note that particles which migrate only to find destination sites already occupied must migrate back to their original positions on source processors. In this application, we test three different synchronization mechanisms, obtaining performance displays for each on a single graph.

5.2.1 Methods of Synchronization

The first two methods of synchronization involve simple *centralized* mechanisms. A master process synchronizes computing done by slave processes. The master process also sends timing information to the DS at each time-step. In the first method, each particle (on a slave) sends a message to the master process after it has been considered for movement. Particles that must migrate send a message only after they have arrived at a final destination location. When all particles have been considered, each slave awaits a signal from the master. The master sends each slave a signal only when it has received completion messages from all particles. If the number of slave processes is p and the total number of particles is n, a total of $(n + p)$ synchronization messages are sent at the end of each time-step. For a simulation with T time-steps, the total cost in terms of messages is $M = T(n + p)$.

The second synchronization mechanism is a small variant of the first. Instead of forcing each particle to send a message to the master process after the particle has been considered for movement, a single message is sent by each slave process after it has finished processing all its particles in a time-step. This message, containing a count of the number of particles processed at the slave (excluding particles which migrate), is sent by the last particle processed at the slave. Migrating particles send messages to the master process independently. Each migrating particle sends a completion message to the master process, when

it arrives at a destination site. When all particles have been accounted for, the master sends each slave a message initiating the next time-step. If we denote the number of migrations at time step i by m_i, the total number of synchronization messages required in this scheme is

$$M = T(2n) + \sum_{i=0}^{i=t-1} m_i$$

Observe that with both schemes just described, migrations are handled with care. That is, the master processor should not instruct slaves to begin a time-step until all migrant particles have already arrived at their destination sites. Initiating a time-step while particles are in transit (which can occur if communication time between processors is large) will result in a logic error.

The third synchronization mechanism is *decentralized*, based on a conservative parallel simulation protocol [2]. The master process does not play any role in synchronization; instead, time-stamps on migrating particles are used for synchronization. If no particles migrate from a processor, the processor sends null-messages [2] to other processors, thus giving them the necessary time-stamp information. In this application, a process interacts with only two other processors– those assigned to neighboring slices of the lattice. Because of this, it is sufficient for a processor to synchronize itself with its neighbors. Processors in receipt of migrating particles must send acknowledgements to source processors because the underlying communication software does not guarantee delivery within a fixed time. A source process can continue execution only after such an acknowledgement is received. The total number of synchronization messages required in this case is

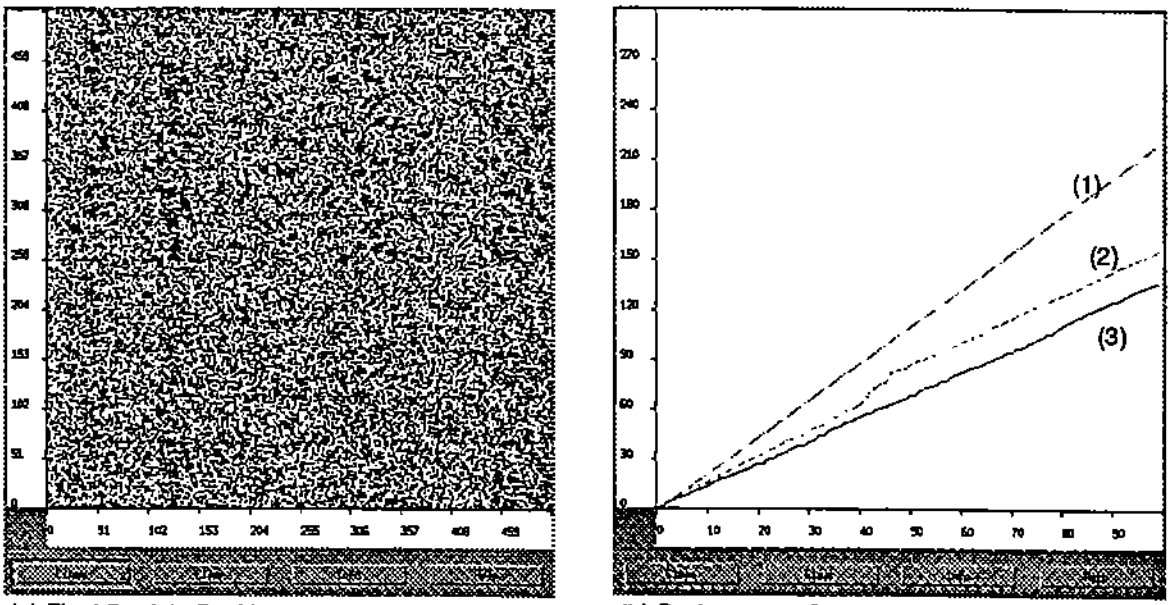
$$M_{max} = T(2n) + \sum_{i=0}^{i=t-1} m_i$$

The *max* subscript denotes the fact that an explicit synchronization message may not be sent if the last particle at any time-step in a processor migrates. The particle migration message can also carry the necessary synchronization information thus reducing the number of synchronization messages.

From our simple message-cost analysis, it would appear that synchronization mechanisms two and three are superior to the first mechanism. The graphical results obtained via DISplay indeed show this to be the case. By simple calculation, the first method performs the best when

$$p < n + (\sum_{i=0}^{i=t-1} m_i)/t$$

The distributed application was implemented in C using the *Ariadne* lightweight process library and the Conch communications library. *Ariadne* supports thread-migration in homogeneous environments. In this example we use a network of five Sun-4 workstations. One of the workstation was made to house a master process (with id 0) which controlled the execution of four slave processes, each on distinct workstations. Thus, each workstation hosted a single process. Lattice dimensions were set at 512×512, and a total of $p = 256$ particles were used.



(a) Final Particle Position on 512x512 Grid (b) Performance Comparison

Figure 10: Particle simulation Grid and Performance

Each particle was implemented as a lightweight process in *Ariadne*. When particles migrate from one processor to another the thread that represents the particle is migrated from one processor to another. This is possible since the *Ariadne* library supports thread migration among homogeneous architectures. The simulation was run for a total of 100 time-steps, with particle-movement graphed dynamically by DISplay. In Figure 10(a) is shown such a DISplay plot of particles, with the dark streaks representing particle paths. The display was implemented using a `Global Plot Points` task. By declaring the task as global when initiated, all processes are able to share this task at the server. At the end of the simulation the starting and the ending positions of each particle, the root mean square displacement and the number of wrap-arounds performed by each particle is printed in the user interface window.

It was possible to observe that all three algorithms functioned correctly by observing paths taken by particles in actual runs. Also, the display verified that particles did not move into inaccessible locations (black dots in display). DISplay allowed particles originating at distinct processors to be assigned different colors, thus aiding in tracking particle migrations across processors. In Figure 10(b) is shown a performance graph for each of the three synchronization mechanisms. Clearly, the first method does not perform as well as the other two. Methods two and three have performed equally well, as predicted by our simple analysis. But since method 3 requires only local synchronization (though with almost as many messages as method 2), it has a slight performance advantage. Also, using DISplay's performance graphs, synchronization method 1 was observed to be best for $p <= 8$.

6 Conclusion

The DISplay software is an application-independent tool for performing Visual Interactive Simulations. It may be used in parallel simulations or computations. It provides the model developer with tools for parametric description of a large set of graphical outputs and capabilities for data visualization, inspection and specification. Abstract as well as representational displays are implemented and these can be dynamic or static in nature. Provision is also made to capture the instantaneous state or the cumulative state of an application. User interaction allows for intervention in model execution, with two way interaction permissible. Model specific handlers can be invoked automatically when a particular type of interaction is invoked by the user. Specific support for parallel computations is provided by allowing several processes to cooperate and draw in a common shared window, and to synchronize with the display. DISplay provides the model developer with a flexible yet powerful tool to create application specific displays and user interactions without any need to know the underlying complexities of graphics and data communications in a distributed environment.

The system described can be enhanced in several ways. Use of a specification file for task and dialog descriptions would be useful in setting up these resources, instead of coding these directly into the program. Interface modification would thus only entail change in the specification file. The specification file could be built interactively. Several tasks, including support for three dimensional surface meshes, and display of multicolumn tables can be added; display of results at more than one workstation — to provide distributed displays — so that multiple users can collaborate in simulation analysis also would undoubtedly be useful. These enhancements are currently under way, as part of a larger effort in heterogeneous distributed simulation. This effort is based on the ACES system [9].

Support for ACES is planned in the form of debugging, and display of arbitrary simulation objects at different points in time. Visual model building and interface specification for ACES-applications is a long term goal. A critical issue we plan to investigate is performance. DISplay was not designed with efficiency as a major goal, though it has proved to be reasonably efficient in our experiments. Future versions will improve on communication overheads with the server. Applications currently suffer at least a 30% increase in cpu time (based on rough measurements), depending on the nature of the displays in the application. The DISplay client library has been ported to HP and RS6000 hardware platforms. Ports to other environments are in progress. We expect DISplay to grow, with support for modeling of ACES simulations that is being developed here.

The basic types of simulation output described here are common to most simulations. This realization motivated us to develop a generalized and extensible display capability. DISplay is now a full fledged VIS tool, enabling easy user interaction and dynamic visualization of output in nontrivial applications. We have used DISplay to visualize parallel simulated annealing, numerical analysis algorithms, particle-physics, and computer network simulations.

References

[1] P. C. Bell and R. M. O'Keefe. Visual interactive simulation – History, recent develeopments, and major issues. *Simulation*, 49(3):109–116, September 1987.

[2] K. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. on Softw. Eng.*, SE-5(5):440–452, September 1979.

[3] N. Collins and C. Watson. Introduction to Arena. In G. Evans, M. Mollaghasemi, E. Russell, and W. Biles, editors, *Winter Simulation Conference*, pages 205–212, December 1993.

[4] D. Comer and D. L. Stevens. *Internetworking with TCP/IP*, volume III:Client–Server Programming and Applications. Englewood Cliffs, N.J. : Prentice Hall, 1993.

[5] R. Fujimoto. Parallel Discrete Event Simulation. *CACM*, 33(10):30–53, 1990.

[6] T. Gaskins. *PEXlib Programming Manual.* The Definitive Guides to the X Window System. O'Reilly & Associates, Inc., 1992.

[7] M. Glavach and D. Sturrock. Introduction to SIMAN/Cinema. In G. Evans, M. Mollaghasemi, E. Russell, and W. Biles, editors, *Winter Simulation Conference*, pages 190–192, December 1993.

[8] D. Heller. *Motif Programming Manual. For OSF/Motif Version 1.1*, volume Six of *The Definitive Guides to the X Window System.* O'Reilley & Associates, Inc., motif edition, 1991.

[9] F. Knop, E. Mascarenhas, V. Rego, and V. Sunderam. Fail-Safe Concurrent Simulation with EcliPSe: An introduction. *Simulation Practice & Theory (to appear)*, 1994.

[10] E. Mascarenhas and V. Rego. DISplay: A technical reference manual. Technical report, Purdue University, Department of Computer Sciences, 1994. In preparation.

[11] E. Mascarenhas, V. Rego, and V. Sunderam. Ariadne user manual. Technical report, Computer Sciences Department, Purdue University, 1994.

[12] H. Nakanishi. Anomalous diffusion in disordered clusters. In P. J. Reynolds, editor, *On clusters and Clustering, From Atoms to Fractals*, chapter 27, pages 373–382. Elsevier Science Publishers B.V., 1993.

[13] H. Nakanishi, V. Rego, and V. S. Sunderam. On the effectiveness of superconcurrent computations on heterogeneous networks. *Parallel and Distributed Computing (to appear)*, 1994.

[14] R. M. O'Keefe. What is Visual Interactive Simualtion? (And is there a methodology for doing it right?). In *Proceedings of the Winter Simulation Conference*, pages 461–464, 1987.

[15] J. J. O'Reilly. Introduction to SLAM II and SLAMSYSTEM. In G. Evans, M. Mollaghasemi, E. Russell, and W. Biles, editors, *Winter Simulation Conference*, pages 179–183, December 1993.

[16] J. K. Ousterhout. An X11 Toolkit based on the Tcl language. In *Winter USENIX Conference*, pages 105–115, 1991.

[17] Pritsker and Associates, Inc., West lafayette, IN 47906. *TESS and SLAMII*.

[18] V. J. Rego and V. S. Sunderam. Experiments in Concurrent Stochastic Simulation: The EcliPSe Paradigm. *Journal of Parallel and Distributed Computing*, 14(1):66–84, January 1992.

[19] M. Rooks. A Unified Framework for Visual Interactive Simulation. In *Proceedings of the Winter Simulation Conference*, pages 1146–1154, 1991.

[20] E. C. Russell. Building simulation models with SIMSCRIPT II.5. CACI Products Company, La Jolla, CA, 1989.

[21] H. Schwetman. *CSIM Users' Guide*. Microelectronics and Computer Technology Corporation, June 1992.

[22] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4), December 1990.

[23] W. Thompson. A tutorial for modelling with the WITNESS visual interactive simulator. In G. Evans, M. Mollaghasemi, E. Russell, and W. Biles, editors, *Winter Simulation Conference*, pages 228–232, December 1993.

[24] B. Topol. Conch: Second generation heterogeneous computing. Master's thesis, Department of Math and Computer Science, Emory University, 1992.

[25] Y.-W. Tung and J. Steinman. Interactive graphics for the parallel and distributed computing simulation. *Proceedings of the Winter Simulation Conference*, pages 695–700, 1992.