NORTH-HOLLAND

A Graphical User Interface Design for Network Simulation

Yi-Bing Lin

Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan, Republic of China

Joe Geigel

Pittsburgh Supercomputing Center, 4400 Fifth Avenue, Pittsburgh, Pennsylvania

This article describes a prototyping effort of a flexible graphical user interface (GUI) for a simulation tool called COPS. The GUI is designed to allow parameter setup for all modules in simulation model, and can be easily replaced by new GUIs implemented in different languages/graphical tools. This article provides design guidelines and implementation details of the flexible GUI.

1. INTRODUCTION

Computer Operations Performance System (COPS) is a simulation tool for network modeling. COPS consists of five interactive components: the *command interpreter*, the *graphical user interface*, the *library*, the *simulation engine*, and the *output analysis*. The organization of COPS is illustrated in Figure 1. The user constructs models by using the graphical user interface (GUI). The command interpreter interacts with the GUI and the model library to construct the executable code for the simulated model. The simulated model is then executed by the simulation engine. The results are displayed by the output analysis component.

A significant distinction between COPS and other simulation packages is the interactions among the components. In most simulation packages (Funka-Lea et al., 1991; MIL, 1991; Northern Telecom Ltd., 1992; Vaughan and Newton, 1993), the components are tightly coupled. On the other hand, COPS components are loosely coupled. Every COPS component is executed by a process, and the interactions among components are done by sending messages between processes. Our loosely-coupled structure provides the following advantages.

- Because the COPS components are isolated in the loosely coupled structure, a component can easily be replaced. For example, we have built three user interfaces; one implemented in HP InterViews (Hewlett Packard, 1992) on in Unidraw (Vlissides and Linton, 1990), and one in Tcl (Ousterhout, 1990). We can integrate either implementation to other COPS components without (or with little) modifications.

  Also, we can easily upgrade the simulation engine to a faster one (e.g., a parallel simulator (Lin 1990). In most packages, the command interpreter and the simulation engine are tightly coupled such that simulation engine cannot be replaced without replacing command interpreter.
- The components can be executed on different machines. Sometimes it is desirable to execute components in different machines. For example, it is possible that the GUI is only available on a slow machine. In such a case, we may prefer to execute the simulation engine on a faster machine where the GUI cannot be executed.

One may argue that communication overhead is very high in the loosely coupled structure. In COPS, most inter-component interactions are between the GUI and the command interpreter. The execution of the simulation engine is not affected by the inter-component communication. Because the inter-

Address correspondence to Dr. Yi-Bing Lin, Department of Computer Science and Information Engineering, National Chiao Tung University, Hsinchu, Taiwan, Republic of China. E-mail: liny@csie.nctu.edu.tw

J. SYSTEMS SOFTWARE 1997; 36: 181–190

655 Avenue of the Americas, New York, NY 10010

0164-1212/97/$17.00
SSDI 0164-1212(95)00201-4

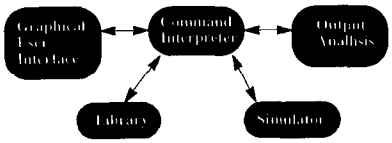

Figure 1. Architecture for COPS.

actions between the GUI and the command interpreter are faster than human response, the performance of the loosely coupled structure is not a problem.

The details of the COPS system were discussed in (Daly et al., 1992). The GUI related part of the command interpreter was described in (Lin and Daly, 1995). In this article, we focus on the design and implementation of the COPS GUI graphical objects. The COPS GUI is a general purpose GUI for applications that make use of graphs as an underlying data model (such as networking applications). This GUI is an example of using a user interface management system (Harbert et al., 1990; Johnson et al., 1993; Mandelkern, 1993), which allows users to create nodes, define and modify data attached to these nodes, and create and delete connections (edges) between nodes. The GUI support hierarchical graph structures. Thus, the data attached to a given graph object can also be a graph.

The GUI acts simply as a filter between the user and the graph application. It has no knowledge of the application nor the consequences of the actions performed by a user. It works hand in hand with a master process which interprets user's input and determines proper responses to this input. Thus, the GUI's role is simply to notify an application of a user's actions. This approach places the responsibility of responding to these actions to the application process.

Separating the task of accepting user input from determining system response makes the GUI quite general. The data model and the sequence of user interactions in completing a task is as simple or complex as the master process guiding the GUI.

The COPS GUI was built using the C + + language. We assume that the reader understands the language.

2. GUI SYSTEM OVERVIEW

When a user starts up the COPS GUI, they are presented with a graph editor where one can create/delete nodes and connect/disconnect nodes with links. In addition, each node can have data associated with it. The associated data is referred to as a *GUI Object*. Each GUI Object has a corresponding dialog object managed by the GUI with which the user can interact, modify, or view the data associated with the object. GUI objects come in three flavors. Each type differs from the other by the kind of dialog item that used to display the data associated with the object. Each is briefly described below.

Graph Editor. A graph editor is a fully functional graphical editor that allows a user to construct, modify, and view graph like structures (see Figure 2(a)). It consists of three areas, a *canvas* area in which the user interacts directly with the graph elements, a *palette* area in which users can select different kind of nodes and links, and a *menu bar* which gives users access to variety of different functions. A graph editor appears when a user first starts up the COPS GUI. In essence, upon startup, the GUI creates a root GUI Object which is edited via a graph editor.

In COPS, the nodes are assumed to have a number of input and output ports connected to it. Thus, when connecting two nodes via a link, the output port of one node is connected to the input port of another. The graph editor allows addition, deletion,

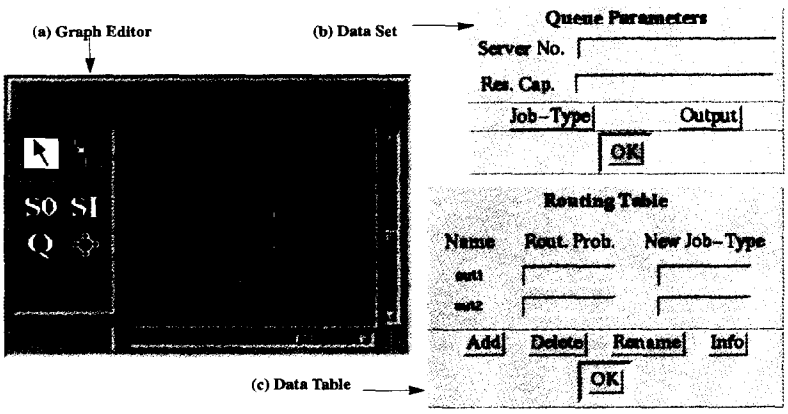

Figure 2. Data Graph, Data Set, and Data Table.

and renaming of the I/O ports. For graph models that do not include ports, the graph editor can be set so that each node only has a single input and a single output port, collapsing the graph model to that of connecting nodes via links without ports. The addition/deletion capability can also be disabled.

Data Set. Data associated with a GUI object can simply be a set of token/value pairs. This type of GUI Object is a data set and is edittable via a dialog box which lists the tokens and current values and allows editing of these value (see Figure 2(b)).

Data Table. Data tables are use for data that is matrix-like in structure. Like data sets, the data associated with a data table will be presented in a dialog box. However, rather than having a single value for each token, an array of values for that token is given. The data is presented as a matrix with values corresponding to the same token given as columns, and groupings of different tokens given as row. The data table allows the user to add rows of data, delete rows of data, or edit individual pieces of data within a row (see Figure 2(c)).

The GUI is essentially a naive process in the sense that it does not know how to respond to a given user's action. In addition, it has no direct access to the library which holds data values included within the GUI objects. Instead, the GUI acts as a slave process to the command interpreter that controls the library and determines the proper response to a user's action. The command interpreter is indeed a separate process and, in concept, need not even reside on the same machine as the GUI.

The command interpreter keeps a registry of all active data objects currently being represented by GUI Objects in the GUI. Because GUI Objects correspond to objects in the application domain, user actions on given active GUI objects can be mapped directly to actions on application objects. The correct response to a user action depends on the GUI Object to which the user's action was addressed.

The sequence of user action and system response is a communication between the command interpreter process and the GUI process, as shown in Figure 3. After the user initiates an action via the GUI (Step 1), the GUI informs the command interpreter which action was taken and on which GUI Object (Step 2). The command interpreter will then consult its library and determine an appropriate response for that action (Step 3). Responses include silent acknowledgment of the action, creation of a

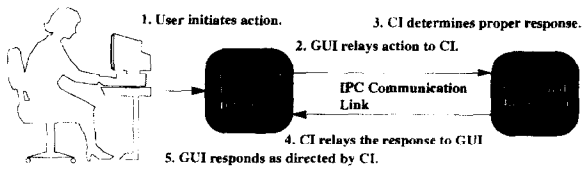

Figure 3. Communication between command interpreter (CI) and GUI.

new GUI Object (as defined by the command interpreter), destruction of an already existing GUI object, negative acknowledgement of an error (Step 4). This response is then communicated back to the GUI, who then carries out the response as instructed (Step 5).

The COPS GUI is built on top of a C + + class library called HP InterViews (Hewlett Packard, 1992). The library encapsulates a general Motif Style GUI into an object oriented paradigm. It is a commercial, Motif version of the InterViews package that originated out of Stanford University. Some of the higher level objects in the InterViews library include dialog boxes, text editors, menus, and other objects commonly found in general purpose GUIs.

3. IMPLEMENTATIONS OF THE COPS GUI

This section describes the implementations of the COPS GUI graphical objects. In COPS, the objects are implemented in C + + language (Geigel, 1993).

3.1. GUI Objects

The `GUIObject` class is an abstract class that represents a group of edittable data managed by the GUI. Each `GUIObject` has associated with a dialog object. The dialog object is used by the user to manipulate the data within the `GUIObject`. Three types of `GUIObjects` are defined in the COPS GUI:

`GraphEd`—This class represents graph data. The corresponding dialog object is a graph editor that allows a user to interactively manipulate individual graph objects.

`DataBox`—This class represents a group of textual data. The corresponding dialog object is a Motif style dialog box with textual fields.

`DataTable`—This class represents a matrix of textual data. The corresponding dialog object is a Motif style dialog box that displays the matrix in tabular form.

`GUIObjects` are the only objects that can directly send messages to the command interpreter

process. The messages are delivered through a static `CommChannel` object that performs the actual communication. The `CommChannel` object is described in more detail in the next section.

3.2. Graph Editor

The graph editor is the means by which a user may interactively manipulate the elements of a graph. It allows creation, deletion, and movement of nodes and links.

The general editor consists of three areas (see Figure 4): the *drawing area* in which interaction with the graph is performed, the *drawing palette* for selecting "tools" used to interact with graph elements (e.g., add new node, rotate, move, zoom, pan, etc.), and a *menubar* for triggering other kinds of functionality (e.g., file operations, editing operations such as cut/paste, etc.).

Once created, these three areas are combined into a single graph editor application. Each area is discussed in more details below.

Drawing Palette. The drawing palette on a graph editor contains model editing controls of the drawing area. The palette is divided into three subareas or *decks* as follows: The first deck contains the *select* control (see Figure 4). This function places the editor into *select mode* where one can select and move objects in the draw area. The second deck contains the *creator* controls. These controls are used to create new nodes and links and introduce them into the draw area. In Figure 4, four node types are defined: **SO** (source), **SI** (sink), **Q** (queue), and subnetwork. In the graph editor palette, there is a single creator control for creating links, and one creator control for each node type. The final deck contains a zoom control which allows zooming of the drawing area.

Creation of new links and nodes into the draw area is all taken care of by two COPS classes `LinkCreatorControl` and `NodeCreatorControl`. When the class is instantiated, a pointer to an example object is passed in. When the control is activated, a copy of this example object is created and placed in the draw area.

Menu Bar. The menu bar is used to create a series of pull down menus. It consists of four pulldown menus: one for file operations (such as Load, Store, New, and so on), one for editing operations (such as Cut, Paste, Clear, and Change Node Name), one for port operations (such as Add Port, Delete Port, and Change Port Name), and a final one for command interpreter defined operations (i.e., the Options menu in Figure 4). Menu items in the Options pulldown menu are determined by the command interpreter, which is intended for application specific functions. When activated, these menuitems will simply notify the command interpreter that they have been activated and the command interpreter will react appropriately. For example, COPS uses this pull down to start a network simulation.

The action that occurs when a menu item is activated is defined by the virtual `MenuItem` function `Do()` which is redefined for each derived class.

The main purpose of the graph editor is for a user to interactively manipulate graph elements (i.e., nodes and links). The COPS GUI defines several classes in the construction of what it knows as a node. A general class is `G_Node`, which is used in the construction of a COPS GUI node (see Figure 5). Similarly, with links, the COPS GUI defines several classes used to make connections between ports. The general class which includes all links used by the GUI is `G_Link` (see Figure 5).

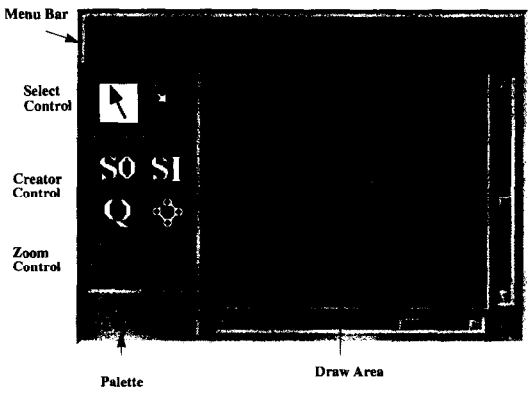

Figure 4. Layout of a Graph Editor.

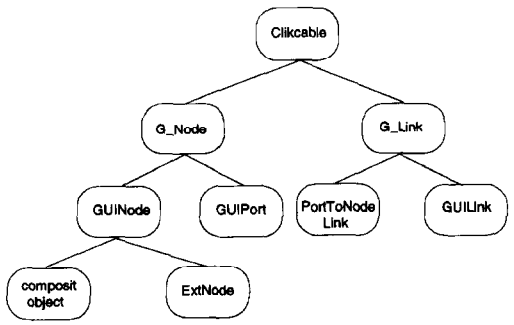

Figure 5. Class hierarchy of Graph Objects.

A COPS GUI node is represented by the class GUINode. Its visual representation is shown in Figure 6. In the figure, the GUINode consists of a base node with ports attached to it. The ports are represented by the smaller squares to the left and right of the base node. These ports are represented by the class GUIPort, and are themselves, derived from G_Node. GUIPorts are connected to their corresponding GUINode by a special class of G_Link called PortToNodeLink. This link is invisible and is managed by the GUI and thus cannot be explicitly created, deleted, or moved by the user. Each port has a name which is displayed next to the square representing the port.

Connections between GUINodes are made via GUILinks, a subclass of G_Link. Note that GUILinks connect GUIPorts and *not* GUINodes directly. These links are created, and deleted explicitly by the user. The user can attempt to connect two GUINodes directly. In this case, the graph editor will query the user between which ports the link is to be created. This query is performed by the Graph Editors link creation palette tool (LinkCreatorControl).

Because GUINodes are to be manipulated as a unit, when a user selects it, moves it, cuts it, or edits it, all ports connected to the node must be selected, moved, cut, or edited with it. Both G_link and G_node are derived from the class Clickable. The Clickable class provides for its subclasses to define callback functions to be invoked when the clickable is moved, selected, unselected, added, copied, etc. This is done via virtual functions defined by the Clickable and redefined by any subclass of Clickable. Thus, these virtual functions must be defined for each subclass of G_Node and G_Link. A subclass is allowed to set many of the parameters associated with a Clickable. (One such parameter is visibility which allows the PortToNodeLink to be invisible.) In Figure 5, the CompositeObject and ExtNode classes represent specialized types of GUINodes. An ExtNode represents a graph's connection to the outside world (i.e., it is either a source or a sink). The CompositeObject is a node that represents a subnetwork.

3.3. Data Box

The data box is used to represent a set of edittable data displayed by a Motif Style dialog box (see Figure 7). The dialog box contains a number of editable fields each having its own textual label. It may also contain user defined buttons which, when pressed, will evoke the presentation of other data related to the data being presented in the databox. Lastly, the box contains a standard "Ok" button which is used to make the dialog box disappear. Like other GUIObjects, reaction to button presses, as well as the changing of a value within a data field, is determined by the command interpreter process.

Several classes are used to define databoxes:

DataBox—This class represents the data box itself.

DataItems—This class represents a collection of items that can be placed within a data box's interior. Each DataBox contains a single member of this class.

DataItem—This class represents a single item of which the DataItems class is comprised.

There are two types of items that can be found in a DataBox's interior, resulting in two subclasses of DataItem:

DataValue—This class represents edittable data fields. Each field has a textual label associated with it. The field/label combination make up the DataValue class (see Figure 8).

Action—This class represents user-defined buttons which are to be placed in a DataBox (e.g., the Output button in Figure 8).

Consider the layout example of a DataBox shown in Figure 8. The DataBox itself has two areas: the

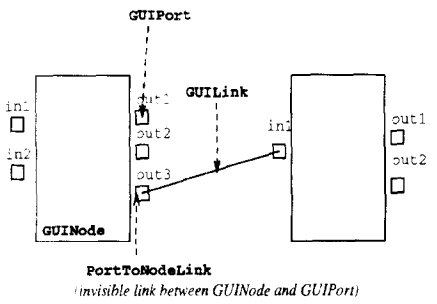

Figure 6. Layout of a GUI node.

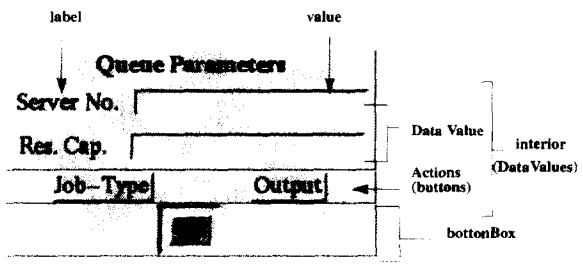

Figure 7. Layout of a DataBox.

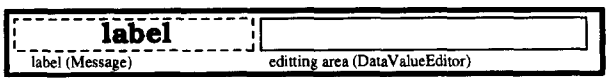

Figure 8. Layout of a DataValue.

interior, which is a `DataItems` object, and a button box, which contains standard buttons that are to appear with the `DataBox` (in our case, merely the "Ok" button). Note that user-defined buttons are found in the interior and *not* in the button box.

All `DataValues` are stacked one on top of another as they are listed in the definition of the databox. `Actions` are aligned horizontally in a box and this box is placed at the bottom of the `DataBox`'s interior (after all of the `DataValues`).

3.4. Data Table

The data table is used to represent an array of edittable data displayed by a Motif Style dialog box (see Figure 9). The data concept is much like a relational database system in which each row of the array represents a single record of data and each column represents field within a data record. Each record has a keyfield which uniquely identifies the record.

Buttons are provided to allow the user to add records, delete record, select and receive additional information about a record, and change the name (value) of a key field in a record. The data table is also provided with a list of default values for each of the data fields. These default values will be supplied when a new row is added to the table (except for the key field which must be explicitly entered by the user). Several classes are used to implement the data table.

`DataTable`—This class represents the data table itself.

`DataTableRow`—This class represents a single row of data in the table.

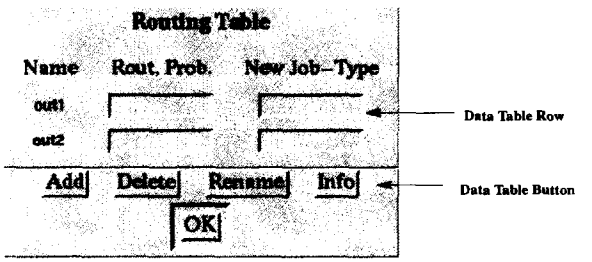

Figure 9. Layout of a DataTable.

`DataTableRowKey`—This class represents the portion of a `DataTableRow` that holds the key field value. It is currently implemented as a radio button (which is how the user selects a given row).

`DataTableValueEditor`—This class represents a single line text editor in which individual field values of a row can be edited.

A data table contains four buttons: add (a row), delete (a row), remain (a row), and (get) information (see Figure 9).

The `DataTableRow` consists of a `DataTableRowKey` and a number of `DataTableValueEditors`, one for each field in the row (see Figure 10). The `DataTableRowKey` is a radio button. This radio button is used to select a current row. This button will inform the command interpreter that the user is choosing the currently selected row. Pushing the radio button on the `DataTableRowKey` makes a row the currently selected row on which button actions are to take place.

The `DataTableRows` are assembled into a vertical strip managed by the `DataTable` (see Figure 9). The four data table buttons are arranged into a horizontal strip. This strip is placed below the data rows. Finally, a sole "Ok" button is placed below the data table button box. This "Ok" button is used to dismiss the data table.

4. COMMUNICATION BETWEEN GUI AND CI

The `CommChannel` object is the communications link between the COPS GUI and the command interpreter process. All messages from individual `GUIObjects` to the command interpreter and vice versa are routed through the `CommChannel` object. The `CommChannel` keeps a registry of all active `GUIObjects` (with their id numbers) being managed by the COPS GUI so that it may route replies from the command interpreter to the appropriate `GUIObjects`. Because of single `CommChannel` acts as courier to all `GUIObjects`, only one instantiation of the `CommChannel` is necessary. This single instantiation must be available to all `GUIObjects` and thus is specified as a static member of the `GUIObject` class.

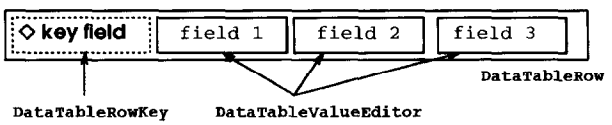

Figure 10. Layout of a DataTableRow.

The CommChannel communicates with the command interpreter process in two ways. First, there is a two way *Inter Process Communication* (IPC) link between the two processes. The CommChannel itself is a subclass of the more general Client class which is a class that establishes IPC with an instantiation of a Server class. All messages between GUIObjects and the command interpreter are conveyed using this IPC link.

Every communication is initiated by the GUI and consists three steps. The structure of the communication between the GUI and the command interpreter via the CommChannel is demonstrated in Figure 11.

4.1. Step 1: User Activated Action

When the user performs an action on one of the GUIObjects, this action is translated by the GUIObject into a textual message to be sent to the command interpreter. The message will contain the identification number of the GUIObject sending it. This message is sent to the single CommChannel which routes it to the command interpreter via the IPC link. It then waits for a reply.

Graph Editor. Most actions taken within a graph editor must be conveyed to the command interpreter for proper registry and response. The GraphEd object, being the only GUIObject involved with a graph editor, has the only direct link to the command interpreter. Thus, all other objects (menuitems, palette, nodes, and links) must route message through the GraphEd to which it belongs. A pointer to this GraphEd object is contained in each one of the graph editor subobjects that need to communicate to the command interpreter.

GraphEd can be used to edit previously save graphs. The contents of a GraphEd is saved into a file upon the GraphEd's exit. It is the responsibility of the command interpreter to supply an initial graph file when defining a GraphEd in a GUI dialog file called cops.tmp. If it doesn't, the draw area of the graph editor will be empty upon its instantiation.

Data Box. Each data item within a data box is assigned a numerical code value that is sent to the command interpreter when that item is acted upon. This code value is determined by the position of the item in the definition of the data box as found in a GUI Dialog file. This code is given to each DataItem upon instantiation.

As mentioned previously, all communication from the GUI to the command interpreter is done via a single CommChannel object. This CommChannel object is accessible by all GUIObjects (including the DataBox), but is *not* accessible by components of a GUIObject (e.g., DataItems). Thus, when a data item is acted upon the message to be sent to the command interpreter indicating this action must be routed through the DataBox object to which a DataItem belongs. This is done via the DataItem's announce method. This method passes an appropriate message to be routed through the DataBox and eventually to the command interpreter.

The sending of a message to the command interpreter is triggered by a user action on a DataItem. For Actions, this happens when the button corresponding the Action object is pressed. For DataValues, a message is sent when a new value is entered in the DataValueEditor. The code, as well as the new value entered, is first communicated to the DataValue to which the DataValueEditor belongs. The DataValue, then, in turn, passes the message to the DataBox to which it belongs. DataBox communication is illustrated in Figure 12.

Data Table. Communication between the data table elements and the command interpreter is handled in very much the same way as with the DataBox. Because the DataTable object is the only object with access to the CommChannel, all

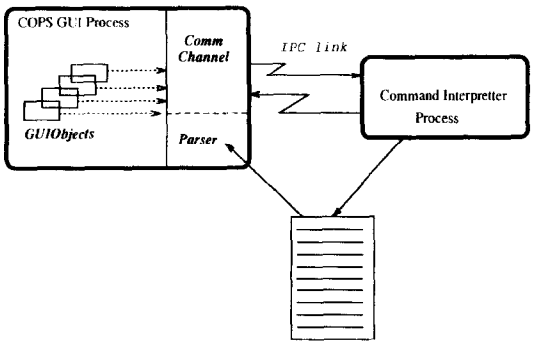

Figure 11. Communicating using the CommChannel.

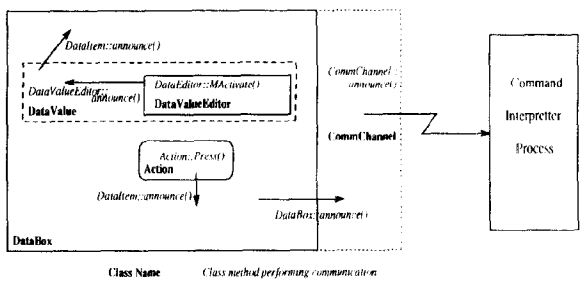

Figure 12. Communication from DataBox components.

communication from interior objects must be routed through the `DataTable` object.

Every data table button and every `DataTableRow` has a pointer to the `DataTable` to which it belongs.

The messages that get sent from the GUI (a graph editor, a data set, or a data table) to he command interpreter is given in (Geigel, 1993).

4.2. Step 2: The Command Interpreter Response

The command interpreter runs in a constant loop waiting for commands from the GUI. When it receives a command, it reads it, interprets it, and then sends a textual reply back to the GUI. The command interpreter will keep a list of objects, corresponding to the dialogs (data sets, data tables, and data graphs), that are currently active. When a command is received by the command interpreter, it should route the command and it's arguments to the appropriate object. The details of the command interpreter's reaction is discussed elsewhere (Lin and Daly, 1995). The reply of the command interpreter can be one of several messages:

- Return a silent acknowledgment: The acknowledgement indicates that the command interpreter has received and successfully interpreted the message from the command from the GUI and that no further action is required by the GUI.
- Issue an error message to the user via an error dialog box: This indicates that the command interpreter has received the command from the GUI, however, some error has occurred during the interpretation of the command. A description of the error is given. This response will result in the GUI displaying an error notification box.
- Return an id number to the `GUIObject` from where the original message originated: This response conveys the id of a newly created node into a graph. It is the correct response to the GUI node creation commands.
- Have the GUI exit gracefully: The command interpreter instructs the GUI to gracefully exit. If an error message is given, it is displayed to the user before exiting. This is for the case of an abnormal exit from the program.
- Remove one or more active `GUIObjects` from the GUI: The command interpreter instructs the GUI to destroy a number of graphical objects associated with the given ids.

As a result of the command, each of the above action can be immediately followed by an indication that a new GUI Object (graph editor, data box, or data table) should be created by the GUI. The description of this new Object will be described in the file `cops.tmp` (see the next subsection).

4.3. Step 3: The GUI Response

When a reply is received, actions may be taken in the GUI. In many cases, the response to a `GUIObject` message may be the creation of a new `GUIObject` to be introduced to the GUI or modification to an existing `GUIObject`. If this is the case, the description of creating/modifying a `GUIObject` is generated by the command interpreter and communicated to the GUI via an ASCII file named `cops.tmp`. It is the responsibility of the `CommChannel` to read this file, either modify the `GUIObjects` or create the new `GUIObjects`, register them, and introduce them into the GUI. The `CommChannel` class includes a member of the `Parser` class, which does the actual parsing of the `cops.tmp` file and returns pointers to `GUIObjects` defined in it.

The syntax of the `cops.tmp` file is given in Appendix A.

5. SUMMARY

This article presented the design of a flexible graphical user interface (GUI) for network simulation. The flexible GUI was implemented in a simulation environment called COPS (Daly et al., 1992). The implementation details were given. Our GUI design has the following features:

- The GUI provides convenient ways for parameter creation/setup in all simulation modules.
- The existing GUI can be easily replaced by a new GUI implemented in different languages/graphical tools.
- Several GUIs can be attached to the same simulation environment.

REFERENCES

Daly, D., Kant, K., Lin, Y.-B. Mak, V., Mok, D., COPS: A Computer Operations Performance Simulation System, in *Second Bellcore Symposium on Performance Modeling*, 1992.

Funka-Lea, C. A., Kontogiorgos, T. D., Morris, R. J. T., and Rubin, L. D., Interactive Visual Modeling for Performance, *IEEE Software*, 58–67 (September 1991).

Geigel, J., The COPS GUI—A Programmer's Guide. Technical report, Bellcore, 1993.

Harbert, A., Lively, W., and Sheppard, S., A Graphical Specification System for User-Interface Design, *IEEE Software*, 12–20 (1990).

Hewlett Packard. *InterViews Plus Programmer's Guide*, 1992.

Johnson, J. A., Nardi, B. A., Zarmer, C. L. and Miller, J. R., ACE: Building Interactive Graphical Interactions, *Communication of ACM*, 36(4):40–56 (1993).

Lin, Y.-B., Understanding the Limits of Optimistic and Conservative Parallel Simulation. Ph.D. thesis, Department of Computer Science and Engineering, University of Washington, 1990. Technical Report 90-08-02, Department of Computer Science and Engineering, University of Washington, August, 1990.

Lin, Y.-B., and Daly, D., A Flexible Graphical User Interface for Performance Modeling. To appear in *Software —Practice & Experience*, 1995.

Mandelkern, D., Graphical User Interfaces: The Next Generation, *Communication of ACM*, 36(4):36–39 (1993).

MIL 3, Inc. *Opnet Tool Operations Manual*, 1991.

Northern Telecom Ltd. *Object Time Supplementary User Guide*, 1992.

Ousterhout, John K. Tcl: An Embeddable Command Language, in *Proceedings 1990 Winter USENIX Conference*, 1990.

Vaughan, P. W., and Newton, D. E., PRISM: An Object-Oriented System modeling Toolkit. To appear in *International Journal in Computer Simulation*, 1993.

Vlissides, J. M., and Linton, M. A. Unidraw: A Framework for Building Domain-Specific Graphical Editors, *ACM Transactions on Information Systems*, 8(3):237–268 (July 1990).

APPENDIX A
THE SYNTAX OF THE GUI DIALOG FILE

To edit the data of a node, it is the responsibility of the command interpreter to inform the GUI of the structure of the data and the method of editing to be used. This is conveyed via a *GUI dialog file* which will be produced by he command interpreter when a specific "`file:`" reply is returned to the GUI. The syntax of the GUI Dialog file is given below in BNF. Each production is explained in detail when it is introduced. Keywords are presented in san serif type. We only describe the syntax for the data set. The syntax for the data table and the graph can be found in (Geigel, 1993).

A.1. The File

⟨*file*⟩ ⇒ ⟨*object_list*⟩

⟨*object_list*⟩ ⇒ ⟨*object_list*⟩⟨*object*⟩|⟨*object*⟩

The GUI dialog file is the means by which the command interpreter communicates graph specific data to the GUI. It also defines the method by which this data will be edited.

The GUI Dialog file is merely a list of descriptions of GUI Objects that the GUI will create, display, and/or modify as the result of a command sent to the CI.

A.2. Objects

⟨*object*⟩ ⇒ ⟨*data_set*⟩|⟨*graph*⟩|⟨*data_table*⟩

There are three types of data objects, each corresponding to a different user dialog method: the dataset, which results in the creation of a dialog box, a graph, which results in the creation of a graph editor, and a data table, which results in the creation of a scrollable table within a dialog box.

A.3. Data Set

⟨*data_set*⟩ ⇒ %DATA{⟨*id*⟩; ⟨*data_items*⟩}

⟨*id*⟩ ⇒ ⟨*integer*⟩

In a ⟨*data_set*⟩, the GUI is given a set of data for which the user to edit. A data_set in A GUI dialog file results in a dialog box containing the data in the data_set. The dialog box will contain a "done" button which the user must explicitly press to conclude the edits on the data in the set.

⟨*data_items*⟩ ⇒ ⟨*data_item*⟩⟨*data_uitems*⟩|
⟨*data_item*⟩

⟨*data_item*⟩ ⇒ ⟨*data_value*⟩|⟨*action*⟩

There are two types of data items: data values and actions, each described in detail below.

A.3.1. Data values

⟨*data_value*⟩ ⇒ %VALUE
(⟨*quoted_string*⟩, ⟨*string*⟩);

Data values are individual pieces of editable data. For each editable value, the following parameters are given (in this order): First is a prompt that identifies the data. This is the ⟨*quoted_string*⟩ parameter. (A ⟨quoted_string⟩ is a character string enclosed in quotes. The string can contain white space. This is in contrast to a ⟨string⟩ which is not quoted and cannot contain white space). Secondly, the current value of the data item is given. This is the value that will originally be displayed for the data item when the dialog box appears. When a data item has been modified by the user, an integer corresponding to the order of the modified item in the dialog box, as well as the new value entered by the user, will be sent back to the command interpreter. Error checking of the user input is the responsibility of the CI.

A.3.2. Actions

⟨*action*⟩ ⇒ %ACTION(⟨*quoted_string*⟩);

The second type of ⟨*data_item*⟩ is the ⟨*action*⟩. An ⟨*action*⟩ results in the creation of a button in the dialog

box. The string ⟨*quoted_string*⟩ indicates the label to be placed on the button. When this button is pressed by the user, an integer corresponding to the order in which the action appears in the dialog box description. It will be sent to the command interpreter. In most cases, this will trigger the creation of another GUI dialog file as a response from the command interpreter.

A.3.3. Dataset specification example. Below is an example of a dataset specification:

```
%DATA {  2222222 ;
    %VALUE ("Data Item #1", value1);
    %VALUE ("Data Item #2", value2);
    %VALUE ("Integer Data Item", 34);
    %ACTION ("Press Me");
    %ACTION ("Classes");
}
```

This specification will create a dialog box with id 222222 that has three fields labeled "Data Item #1", "Data Item #2", and "Integer Data Item", with initial values value1, value2, and 34, respectively. The dialog box will also contain two buttons labeled "Press Me" and "Classes".