A Visual Process Connector for Unix

Mitali Bhattacharyya**, **David Cohrs**, and **Barton Miller
University of Wisconsin at Madison

Distributed programming benefits greatly from visual tools that help you separate the interactions of processes from their implementation. Upconn is such a tool for Unix programmers.

Specifying the structure or the interconnection of processes in a distributed computation is a cumbersome programming task. The Upconn tool addresses this problem. Rather than requiring you to describe process interaction textually, Upconn lets you visually describe the connections between the processes in a distributed program and then execute the distributed program. Upconn simplifies the creation of these permanent connections between processes and allows a structural specification of the distributed program. Describing the connections with Upconn means that the computation performed by processes is separated from the creation of the connections between them. Upconn does all this with a simple, functional, and extensible interface.

To achieve these goals, we divided Upconn into several modules. You can extend Upconn by adding to a library of tools rather than by adding many special features to Upconn itself. For example, if you wanted to add a message-monitoring facility to Upconn, you would add the monitor as a process in the library instead of making it a special property of connections in the Upconn definition. The lack of extra integrated features prevents Upconn from becoming a monolithic tool, allows a modular design, and lets other applications use parts of the library package. You create an Upconn program by taking many of its parts from a library of processes.

Upconn has three main uses. First, it lets researchers study distributed processing in common Unix environments, reducing the dependence on specialized environments. Second, it lets you rapidly prototype new distributed applications. Third, Upconn is a learning tool that can help students focus on writing distributed programs without dealing with the complexities of the communication links.

Ideal process connector

By examining related work (see the box below) and our own goals, we outlined a set of features that we felt a process connector should provide for a distributed computing environment. Processes in a distributed program communicate via message passing, using a reliable, bidirectional interprocess communication path as their connections. Program objects can be processors or connections.

In our minds, an ideal connector would provide a visual, interactive environment — with graphical displays and pointing devices — to specify a distributed program's structure. In this environment, you could construct, modify, and execute distributed programs. You would specify processes — whether new processes or existing server processes — with a connector tool and make connections between them. As with the hierarchical process-composition model,[1] you could use this tool to define and replicate complex structures of processes and connections.

Second, our ideal tool would interactively execute distributed programs and monitor and control execution. Given a structural description of the program, the

Related work

Many of the ideas used in Upconn came from other systems we had studied, including the methods for connecting processes, the structural specification of programs, and a visual and interactive environment for program description and execution.

Process communication. Each system we studied includes a method for connecting processes. A process must first locate the process with which it wants to communicate and then set up a connection to that process. But it is difficult for unrelated processes to locate each other in a distributed system. Furthermore, code to handle these connections must be included in every program, resulting in code duplication.

To address these problems, we followed the switchboard solution in Demos.[1] The switchboard creates connections from itself to processes as they are created. For example, when process *A* wants to connect to process *B*, it asks the switchboard to add a link between process *A* and the switchboard to the link between the switchboard and process *B*. This lets a process form connections with another registered process by specifying the name of that process. Application programs need no longer know how connections are established.

The switchboard provides low-level support for process connection, so processes must still include the calls to the switchboard at runtime. A higher level tool can make these calls for the program and execute the program once its connections are in place. The Charlotte distributed operating system[2] has such a higher level connector, which separates the implementation of the processes from the implementation of the connections between these processes. The connector uses a special description file that lets you specify processes and the connections between them. Using the description file, the connector creates the processes, forms connections between them, and tells each process which connections it holds. The connector uses a switchboard to register the processes that it starts.

There are several ways to specify a description file's structure. You can specify single processes or arrays of processes. Arrays allow the specification and connection of several similar processes. They are also a simple structuring technique that lets you quickly specify many programs, such as replicated servers.

The hierarchical process-composition model[3] uses an object-oriented model to specify connections. It lets you define typed objects and operations on these objects. The model's basic objects are processes and links between processes. As in the Charlotte connector, the model separates the implementation of processes from connections. It does so with an object called the controller, which combines sets of processes and channels into one object, like how object modules are combined into an executable program by a linking loader. The hierarchical process-composition model is a low-level mechanism with no friendly user interface.

Visual tools. With graphical tools, you can interactively edit a pictorial representation of the processes and the connections between processes in a program.

The Poker Parallel Programming Environment,[4] which runs on Digital Equipment Corp. VAX 11/780s under Unix, developed such a graphical tool for the Configurable, Highly Parallel computer, which had 64 interconnected processors and whose architecture is a lattice made up of nodes that represent the processors in their system and arcs that are the programmable or communication channels between them. To create a parallel program, you edit a pictorial representation of the lattice and specify which processes should be placed on which processors and what the communication is between them. Poker is tailored to support a specialized, regular communication architecture. This simplified the display and specification of connections.

A visual interface is especially useful when combined with a system that lets you dynamically create and destroy connections. The Conman connectino manager[5] is one such process (and is similar to the switchboard) that lets you control the connection between processes. In Conman, input and output ports are associated with processes. These ports have names and are registered with Conman when the process starts. Connections are made by connecting an input port of one process to the output port of another. But Conman is tailored for interprocess communication on one workstation rather than for communication in an distributed program.

References

1. F. Baskett, J.H. Howard, and J.T. Montague, "Task Communication in Demos," *Proc. Sixth Symp. Operating Systems Princ.*, ACM, New York, 1977, pp. 23-31.
2. Y. Artsy, H. Chang, and R. Finkel, "Interprocess Communication in Charlotte," *IEEE Software*, Jan. 1987, pp. 22-28.
3. T. LeBlanc and S. Friedberg, "Hierarchical Process Composition in Distributed Operating Systems," *Proc. Fifth Int'l Conf. Distributed Computing Systems*, CS Press, Los Alamitos, Calif., 1985, pp. 26-34.
4. L. Snyder, "Parallel Programming and the Poker Programming Environment," *Computer*, July 1984, pp. 27-36.
5. P. Haeberli, "A Dataflow Manager for an Interactive Programming Environment," *Proc. Usenix Summer Conf.*, Usenix, Seal Beach, Calif., 1986, pp. 419-428.

connector would execute the program processes on the hosts in the network. By using multiple windows, the tool would monitor process activity through a monitoring window for individual processes and connections. It would automatically gather statistics about message traffic, such as the contents of messages and the times they were sent. You could also control execution by modifying connections during execution, as the Conman tool allows (the box on p. 44 describes Conman and other related work).

Third, the connector should provide process placement. You would specify a host on which a process should execute (your choice would of course be restricted to those hosts you may access). If you have no preference, the tool — using an appropriate load-sharing algorithm — would place the process on a host that can most readily handle the extra load. Chou has described several such load-sharing algorithms.[2]

Last, our ideal connector should be a modular tool that supplies a structure to which other functions can be added. Modularity is attained by including these functions in a library of processes that you can link to other programs. You would add new functions by extending the library and leaving the tool's primitives unchanged. Ordinary processes would provide the extended library functions. For example, you could add file access to the tool by providing a special process in the library instead of adding any new types of objects to the tool.

We designed Upconn to provide as many functions of the ideal connector as possible in the Unix environment.

Upconn elements

Upconn runs on Unix 4.3 BSD workstations under the X Window graphical environment. As Figure 1 shows, it has three major modules, each of which is a separate program:

- The Upedit editor graphically connects modules in distributed programs.
- Upsh executes the distributed program.
- Upstart, which is invoked by Upsh, starts the individual processes on the appropriate hosts.

Upedit lets you pictorially describe processes and the connections between them. With its command menu, you can manipulate the description of processes and connections. Upedit is an interactive editor whose objects are processes and connections rather than text files or documents.

Upedit creates a description file to permanently store a program's pictorial representation. This file is the only way that Upedit and Upsh communicate, using a file format understood by both programs.

Upsh and Upstart execute the distributed program. Upsh interprets the description file and uses Upstart to execute the individual processes. Upstart forms

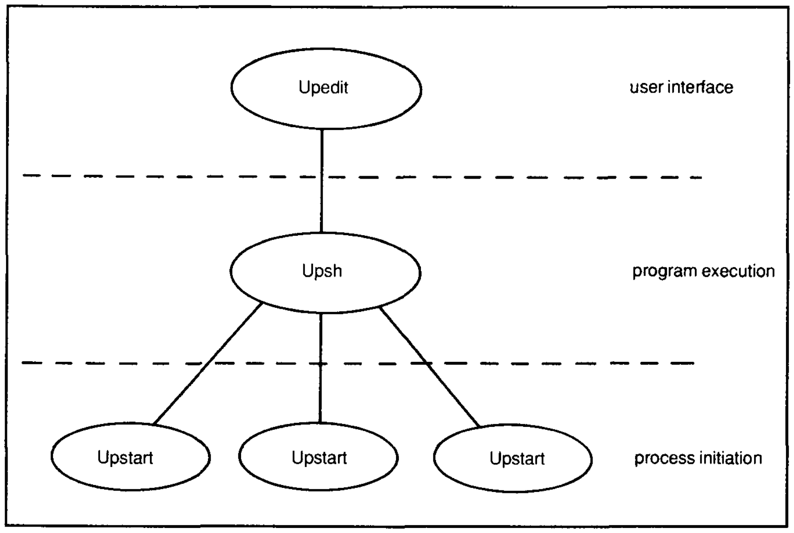


Figure 1. Upconn's functional hierarchy.

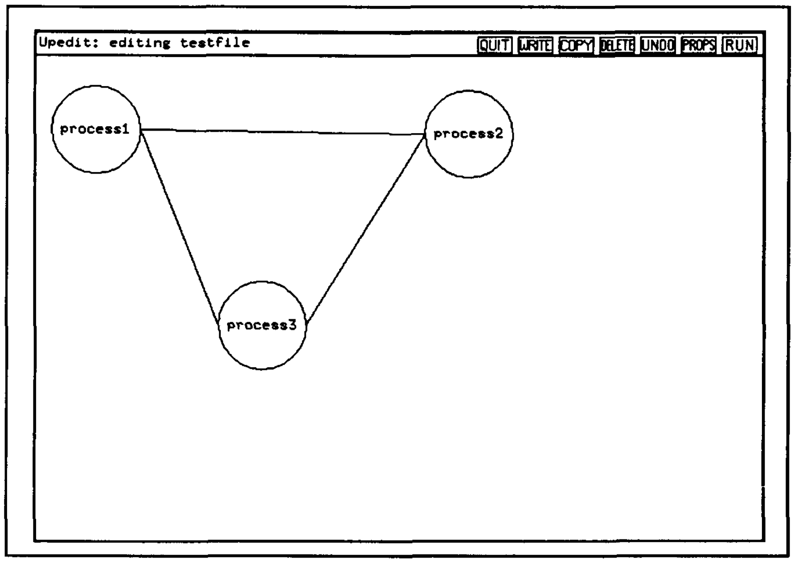


Figure 2. Example Upedit display.

DONE I CANCEL	Process properties
Process name:	process1
Program and args:	~dave/reader a b
Use host:	romano
Window:	yes
DONE I CANCEL	Connection properties
Process1:	process1
File descriptor 1:	5
Process 2:	process2
File descriptor 2:	5

Figure 3. Upedit process and connection property sheets.

the connections between processes using a switchboard service[3] and executes the individual processes. Upsh and Upstart separate the execution of a distributed program from its description.

Figure 2 shows the description file for the display in Figure 3; Figure 4 shows the description grammar. This separation of Upedit from the description file lets other programs generate description files. For example, if an application needs a different user interface or does not require a user interface at all, it can generate the description file itself. This is useful in an application that uses a program to perform a computation; the application can automatically generate a description file for the program and execute it with Upsh.

This modular design means Upsh users are not restricted to the Upedit environment.

Upedit. The editor is the primary user interface to Upconn. Upedit's basic functions are creating, modifying, replicating, and deleting processes and connections. You use Upedit to describe a distributed program's structure and to interactively execute the program.

Upedit displays the program in a window, as Figure 3 shows. With a mouse and keyboard, you create processes and define the connections between them. The Upedit window is divided into two regions. The upper section of the window, called the banner, displays the program name and the main editing options. Other commands are entered from the keyboard. The area below the banner is used to edit the program.

The Upedit display represents a process by a circle that you can display by pressing a mouse button. Connections are represented by a line and are created by selecting the two processes to connect with the mouse. A process may have connections to many other processes. In our current implementation, only one connection may exist between any pair of processes.

After you have created processes and connections, you can specify attributes of these objects. Process attributes include its name, the host it will run on, and whether the process needs its own monitor window to display debugging information. Connections attributes include the Unix file descriptors through which the processes should communicate.

If you do not specify some attributes, such as the process name or file descriptor, Upedit supplies default values. Upedit displays attributes in a property sheet (see Figure 5. We took the concept of property sheets from a similar mechanism in the Xerox Star word processor.[4]

Several commands are available to edit a program, including Copy, Delete, and Undo.

Once you have created the pictorial representation, you can use Upedit to start program execution. Upedit first translates the picture into the description-file format and then invokes Upsh to execute that description file.

After invoking Upsh, Upedit has nothing to do with the program execution. When execution is completed, control returns to Upedit.

Upsh and Upstart. Upsh reads the description file, constructs the commands necessary to execute the program on the remote hosts, and calls Upstart to execute the individual processes. If no host is specified in the description file, Upsh picks one arbitrarily from a list of hosts in a configuration file.

If a process's executable image does not exit on the host that is selected to execute it, Upsh opens a special connection (called the file-transfer path) to the copy of Upstart on the remote host and then copies the executable image to it (see Figure 6).

```
#!/usr/local/upsh
# Unix magic necessary to run this program

# first, the name of the process: "process1"
process process1;
# next, the command line necessary to execute the process
    args reader a b;
# the host on which to run the process
    using romano;
# the location to display the process in Upedit
    at (100,100);
# a flag that causes a monitor window to appear when the process runs
    window;

# the description for process "process2"
# no specific host or window for this process
process process2
    args writer;
    at (300,100);

# a connection between process1, on descriptor 5, and process 2, on descriptor 5
connect <5,process1> <5,process2>;
```

Figure 4. Example Upconn description file.

After the program begins, Upsh waits for all the processes to complete and then exits. You can cancel execution at any time by sending an interrupt request to the window running Upsh.

On the remote hosts, the local copies of Upstart form the connections between the processes and begin their execution. Upstart forms connections to the process using the switchboard server, implementing connections as a pair of 4.3 BSD Unix sockets.

Upsh uses Unix's remote-shell facility (Rsh and Rshd) to execute Upstart on the remote host. This facility authenticates users and initializes the remote runtime environment before letting processes execute. Figure 6 shows the interactions between Upsh, Upstart, Rsh, and Rshd.

When program execution begins, Upconn creates a window that displays output generated from Upsh. You may create a monitoring window for any process; output from Upstart and your process will be displayed in these windows. Monitoring windows are process attributes that you specify when you describe the program in Upedit.

```
<file>            :   <process_defs> <connect_defs>

<process_defs>    :   process_def { process_def }

<process_def>     :   "process" <word> ";" <clauses>
<clauses>         :   { <clause> }

<clause>          :   "using" <word> ";" |
                      "arg" <arglist> ";" |
                      "at" "(" <number> "," <number> ")" ";" |
                      "window" ";"

<arglist>         :   <word> { <word> }

<connect_defs>    :   { connect_def }

<connect_def>     :   "connect" "<" <number> "," <word> ">"
                      "<" <number> "," <word> ">" ";"

<word>            :   <char> { <char> }
<number>          :   <digit> { <digit> }

<char>            :   <letter> | <digit> | <other>
<letter>          :   "a" ... "z" | "A" ... "Z"
<digit>           :   "0" ... "9"
<other>           :   "!" | "@" | "$" | "%" | "^" | ...
```

Figure 5. Description-file grammar.

Utilities. We designed Upconn so users can add features to its basic set of primitives. We have added several utility processes and interface procedures designed so you can link them to your program with Upedit:

• The Upmonitor utility monitors messages between any pair of of user processes and either displays the messages in a window or logs them for later use. Upmonitor also lets you stop the message flow between the processes it is monitoring. To use Upmonitor, you connect two processes to Upmonitor rather than to each other.

• The Upfileserver utility process lets processes treat files as ordinary processes and access these files by passing messages. By providing file access through Upfileserver, we have incorporated file access into the Upconn semantics without adding special object types. Upfileserver supports read, write, seek, begin-transaction, and end-transaction operations. The begin-transaction and end-transaction requests provide mutual exclusion and synchronization.

Upconn's modular design meant that we could add these utilities through the extended library without changing the other parts of Upconn.

Upconn provides interface procedures through a library, called Uplibrary, that can be linked with individual processes. Library procedures include routines to provide mappings between process names and their associated file descriptors and routines to simplify access to Upfileserver.

The routines get_conn_fd and get_conn_name map between the Unix file descriptor and the Upconn name for a connection. Given the name of a process, get_conn_name returns the file descriptor associated with the connection to that process. You use get_conn_name when a process does not know the file descriptor to use to communicate with a specific

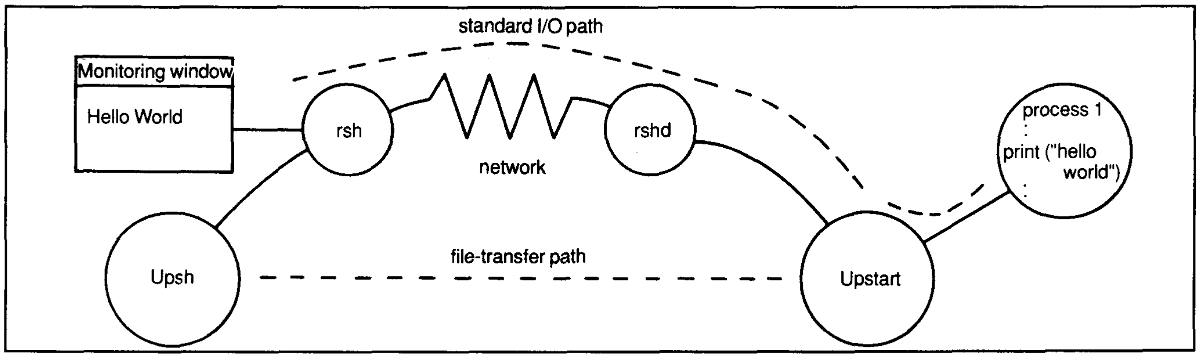
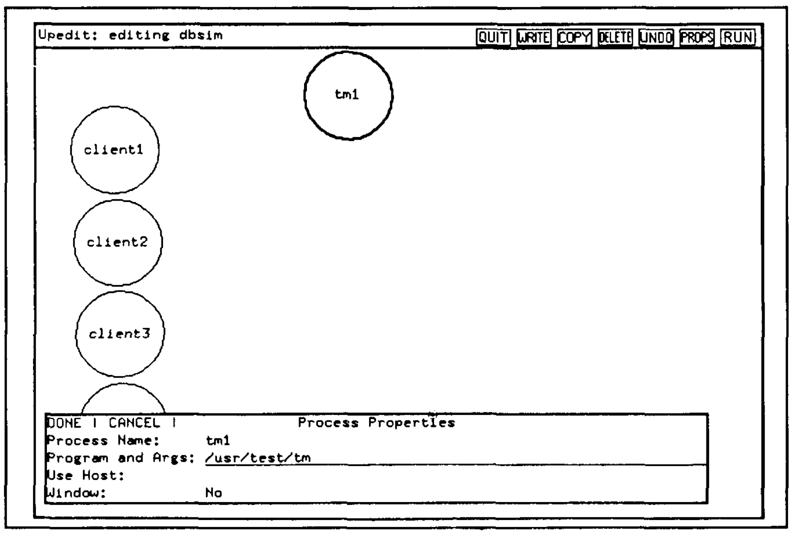


Figure 6. Interaction and communication of Upconn support processes.



```
DONE | CANCEL |                Process Properties
Process Name:      tm1
Program and Args:  /usr/test/tm
Use Host:
Window:            No
```

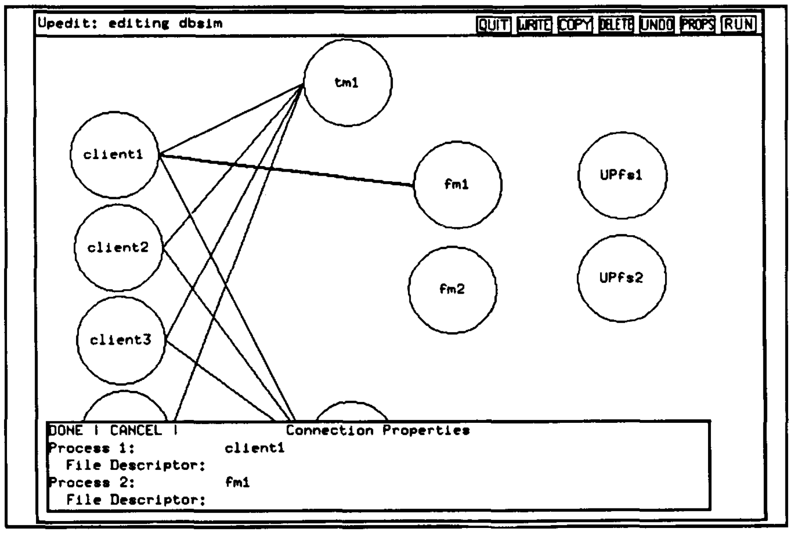
Figure 7. Adding tm1 to dbsim.

process. For example, if you do not specify a file descriptor when creating a connection in Upedit, Upstart will choose the file descriptor. But while Upstart knows the name of the process, it will not know the file descriptor's name. Calling get_conn_name will determine this name for the process.

Several routines — upfs_read, upfs_write, upfs_seek,upfs_begin_trans, and upfs_end_trans — provide access routines for Upfileserver, so you can treat Upfileserver as a file even though it is implemented as a process. The read, write, and seek routines have the same semantics as the standard Unix read, write, and seek routines, so access to files through Upfileserver is the same as access to regular Unix files.

You can also use simple Unix utilities like Cat, Sed, Tee, Yes, and Grep. A Unix utility will generally read from its standard input file and write to its standard output file when no special input or output files are specified, so they can be inserted into a program and have connections attached to their input and output descriptors.

For example, you can use Cat to generate a stream of data from a file and to display all incoming messages in a monitoring window. Sed can edit a stream of data passing between two processes. You can use Tee to log messages in a file while also passing them through to another process. You can use Yes to repeatedly generate messages, which is useful when debugging programs. And you can use Grep to filter out unnecessary messages when testing a program.



```
DONE | CANCEL |              Connection Properties
Process 1:          client1
 File Descriptor:
Process 2:          fm1
 File Descriptor:
```

Figure 8. Creating connections in dbsim.

Upconn example

An example of how you could use Upconn is to set up a simulation of a simple distributed database system. Consider a database system with several file managers, transaction coordinators, and clients. We used Upconn to set up the processes in the simulation, run the simulation, and make changes to the program layout. In this example, there are four simulated client processes: client1 through client4, a transaction-manager process ("tm" in Figure 6), and a file-manager process ("fm" in Figure 6). The file manager uses Upfileserver to store the database files.

First, we set up a simulation with four clients, two transaction managers, and two file managers. We created each process with Upedit. Figure 7 shows the Upedit window after we specified the four clients and while we were describing one of the transaction managers, tm1.

Next, we made connections between the processes. We needed a connection from each client to each transaction and file manager. We also needed connections between the transaction managers and from the file managers to the Upfileservers. Figure 8 shows the clients' connections to the transaction managers and a new connection being created between client1 and the fm1 file manager. Figure 9 shows the completely described program.

We then ran the simulation by selecting the Run command in the upper right corner of the Upedit window. After running it, we decided to change the simulation slightly by deleting client2. To do this, we first deleted all of client2's connections (one to each transaction manager and file manager) and then deleted client2 itself. Figure 10 shows the new version of the program. We could have then run the program again, made further changes, and run it yet again.

Upconn's visual interface can help you rapidly describe, modify, and execute programs. Compared to using a text editor, this graphical approach gives you more immediate feedback to changes in the program specification.

Applications

One of the main uses for Upconn is to teach students programming. A common student complaint about programming is that it is difficult at first to conceptualize and work with programs.

Upconn has also aided a research project in our department, the DIB project,[5] which generates automatic parallel implementations of backtracking programs. DIB uses Upsh and Upstart to place processes, form the necessary connections, and start program execution. DIB was originally developed for the Crystal multicomputer,[6] running on the Nugget operating system. Before Upconn was installed, DIB was not available on our Unix machines because there were few tools for distributed computing on these machines. Upconn is one of the first tools available in our environment to ease the constructino of distributed programs.

Upconn provides most — but not all — of the features our ideal connector tool would have. It still needs:

• Better process placement because, while Upconn now provides a placement mechanism, it does not help you develop load-balancing policies.

• Better facilities to replicate objects and groups of objects so we can build higher level semantic descriptions from the basic functions. Fortunately, this requires no changes to the description file's grammar.

• Faster start-up time because Unix's Rsh facility, which Upconn uses to start process execution, is relatively slow. We intend to have Upconn directly start remote processes.

• Greater support of distributed debugging by extending Upmonitor, which now only records and displays messages in an easy-to-read format. Planned extensions will let you stop and restart the monitor and let you replay and insert messages.

Upconn is designed to visually help you connect and execute programs. It simplifies your task by providing a graphical and structural approach to writing programs. It also separates the implementation of processes from the implementation of connections between them. Rather than provide an elaborate set of options, our connector tool provides a simple, functional interface and lets you add new features easily. You can use Upconn's functions without direct knowledge of the underlying implementation.

We designed Upconn to make programming possible on the workstations in our research environment and to make describing these programs easier. One key test of any such programming tool is whether people actually use it. People in our research community do use Upconn. ❖

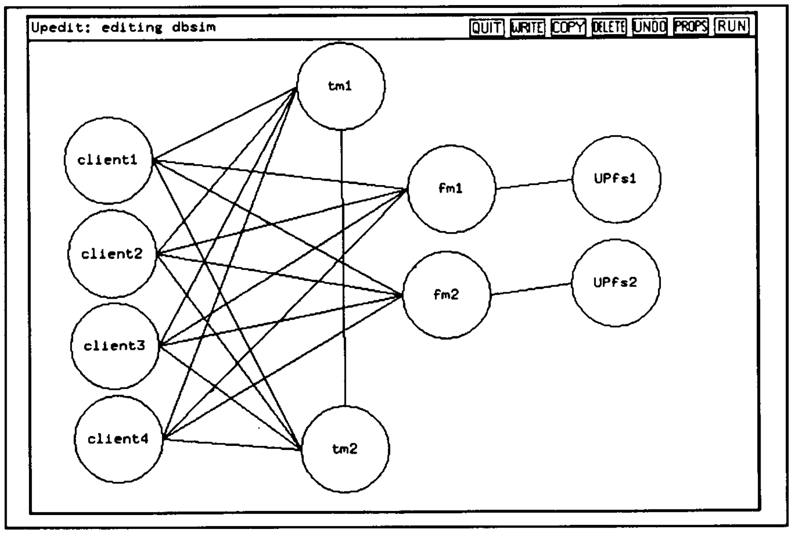


Figure 9. The full dbsim program.

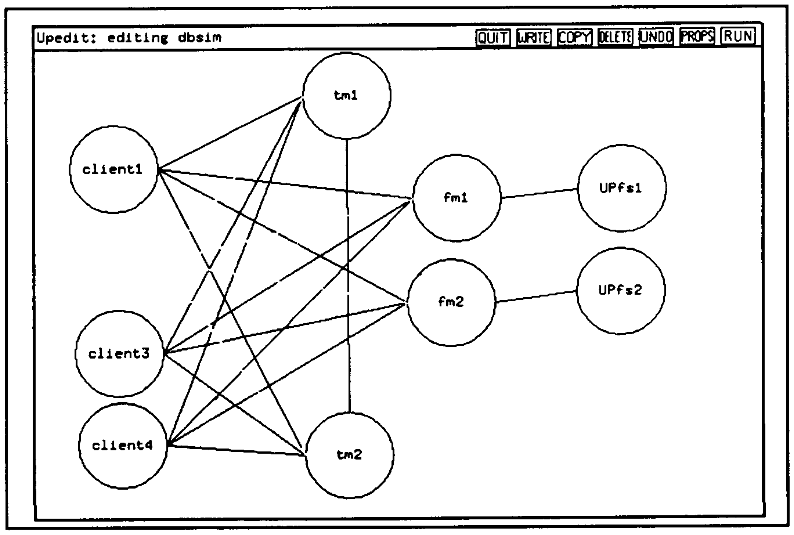


Figure 10. The modified dbsim program.

References

1. T. LeBlanc and S. Friedberg, "Hierarchical Process Composition in Distributed Operating Systems," *Proc. Fifth Int'l Conf. Distributed Computing Systems*, CS Press, Los Alamitos, Calif., 1985, pp. 26-34.
2. T.C.K. Chou and J.A. Abraham, "Load Balancing in Distributed Systems," *IEEE Trans. Software Eng.*, July 1982, pp. 401-412.
3. D. Draheim, B.P. Miller, and S. Snyder, "A Reliable and Secure Unix Connection Service," *Proc. Sixth Symp. Reliability in Distributed Software and Database Systems*, CS Press, Los Alamitos, Calif., 1987, pp. 15-21.
4. D.C. Smith et al., "The Star User Interface: An Overview," *Proc. Nat'l Computer Conf.*, AFIPS, Reston, Va., 1982, pp. 515-528.
5. R. Finkel and U. Manber, "DIB: A Distributed Implementation of Backtracking," *Proc. Fifth Int'l Conf. Distributed Computing Systems*, CS Press, Los Alamitos, Calif., 1985, pp. 446-452.
6. D. Dewitt, R. Finkel, and M. Solomon, "The Crystal Multicomputer: Design and Implementation Experience," *IEEE Trans. Software Eng.*, Aug. 1987, pp. 953-966.

Mitali Bhattacharyya is working toward a PhD in computer science from the University of Wisconsin at Madison. Her research interests include database systems and user interfaces.

Bhattacharyya received a BS and MS in computer science from the University of Wisconsin at Madison.

David Cohrs is working toward a PhD in computer science from the University of Wisconsin at Madison. His research interests include user interfaces, operating systems for multiprocessor and distributed systems, and networks.

Cohrs received a BS and MS in computer science from the University of Wisconsin at Madison. He is the recipient of an AT&T graduate fellowship.

Barton Miller is an assistant professor of computer science at the University of Wisconsin at Madison. His research interests include distributed operating systems, parallel and distributed debugging, distributed program measurement, and user interfaces.

Miller received a BA in computer science from the University of California at San Diego and an MS and PhD in computer science from the University of California at Berkeley.

Address questions about this article to the authors at Computer Sciences Dept., University of Wisconsin, 1210 W. Dayton St., Madison, WI 53706; Internet bart@cs.wisc.edu

Object-Oriented Design&Programming

... from the designers of EIFFEL!

THE BOOK

Object-Oriented Software Construction
by Bertrand Meyer

Prentice-Hall (1988), 552 pp, ISBN 0-13-629049-3.

The first complete presentation of the approach that is revolutionizing software engineering.

In America: Order from your bookseller or Interactive Software Engineering. In Europe: From your bookseller or Soc. des Outils du Logiciel, 4 rue R. Barthélémy, 92120 Montrouge, France. Phone: 1-46 57 13 36, Fax: 1-46 57 01 03.

THE COURSE

"Object-Oriented Design: The New Breakthrough in the Search for Software Quality and Productivity"

taught by Dr. Bertrand Meyer. A unique opportunity to learn firsthand how the next generation of software will be developed.

Boston, July 14-15; Toronto, July 18-19.

See our Eiffel ad, inside front cover.

Interactive
Software Engineering Inc.

270 Storke Road, Suite 7, Goleta, CA 93117 USA
Phone: 805-685-1006 – Fax: 805-685-6869

Reader Service Number 7

Attention Engineers!

NEED HELP WITH SCREENS, DEMOS AND GRAPHICS?

Try the **DisplayExpress**™ and **C-Display Librarian**™! This software can improve your competitive position and increase your productivity. SATISFACTION GUARANTEED OR YOUR MONEY BACK.

"Tremendous productivity tools"
*(**PC Week** October 27, 1987)*

"Many applications will benefit from the quick development times and minimal programming required to implement a finished, quality product"
*(**IEEE Software** November, 1987)*

"If you want to include bit-mapped graphics in your application, then this might be the package for you"
*(**Computer Language** June, 1987)*

DisplayExpress™ $69
C-Display Librarian™ $69

Send check or money order now to:

SYDETECH
SYSTEM DEVELOPMENT TECHNOLOGIES, INC.
43-23 COLDEN STREET, #17C, FLUSHING, N.Y. 11355

For COD orders or the name of the nearest dealer please call (718) 886-0221

Reader Service Number 8