Using $\mathbf{Hy}^+$ for Network Management and Distributed Debugging

Mariano P. Consens, Masum Z. Hasan, Alberto O. Mendelzon

Abstract

A network manager managing a computer network or a programmer attempting to understand and debug a distributed program both must deal with large volumes of data. Visualization is widely believed to help in these and similar tasks. We contend that visualization is indeed useful, but only if accompanied of the following facilities: abstraction, filtering, and layout control. The $\mathbf{Hy}^+$ visualization system and GraphLog query language provide these facilities. They support not just a fixed way of visualizing data, but visualizations that can be specified and manipulated through declarative queries, like data are manipulated in a database. In this paper we show how the $\mathbf{Hy}^+$/GraphLog system can be used by network managers or distributed debuggers to meet their information manipulation and visualization goals.

With the aid of this system, a network manager can specify network management functions as declarative data manipulation statements (GraphLog queries), for example, to define alerts. The manager can also filter network management information and control the layout and aspect of the resulting visualizations.

In the same way, the $\mathbf{Hy}^+$/GraphLog system can be used for debugging of distributed and parallel applications by specifying normal or abnormal patterns that the programmer is looking for as declarative GraphLog queries and manipulating the resulting visualizations to understand the behavior of the program.

1. Introduction

Both network management and distributed and parallel program debugging are two areas where huge volumes of data must be analyzed for presentation purposes. In this paper we show that a system that provides a convenient marriage of deductive database technology with the visualization technology is well suited for specification and presentation services of network management and distributed debugging. Database technology is well suited for dealing with large volumes of data. To understand the behavior of a complex system, visualization techniques are widely used. But simple and fixed way of visualizing data are not well suited to the intended application domains. Considering visualizations as objects of manipulations, like data in a database, allows flexible visualizations to be achieved. In other words, the proposed system allows a network manager or a debugger to specify and manipulate visualizations through declarative queries, like data are manipulated in a database. Visualizations become meaningful when they are combined with *abstraction*, *filtering*, and *layout control.* In both of these areas data is collected by real-time monitoring of the target system. The proposed system should also allow specification of monitoring functions as declarative data manipulations statements.

The $\mathbf{Hy}^+$/GraphLog system developed at the University of Toronto achieves all of the stated goals. $\mathbf{Hy}^+$ is a general purpose data visualization environment that has been applied to areas such as software engineering [12], network management [7], and distributed and parallel program debugging [9]. GraphLog [10] is the visual query language for the $\mathbf{Hy}^+$ system.

1.1. Network Management

Managing a large heterogeneous network is a very complex task. The proposed network management standards for TCP/IP-based Internets [33, 4, 27] and the ISO-OSI network management standard [22, 23, 21] are attempts by standards organizations to reduce the complexity involved. The manager-agent paradigm, where the manager keeps the global picture of the network, has been proposed in both standards.

The complexity of managing, operating, and controlling networks is reduced by systematically imposing different levels of abstraction onto the network. Doing so, allows the functional management to be performed in a meaningful and structured way. Network objects can be abstracted into higher-level entities based on different criteria set by the network manager, for example, geographical sites, such as countries, cities, buildings, floors, and rooms, or administrative responsibilities within the network, e.g., domains. Network objects can also be abstracted into entities that can be derived by the ISO/OSI protocol layering. Using the protocol layering, we can distinguish between two different views of the topology map of the network: the *logical map* which corresponds to the network layer, and the *physical map* which corresponds to the datalink and physical layers. The network addressing mechanism (e.g., IP) defines a logical or network layer map of the network. Devices like bridges, repeaters, and cables, together with the datalink and physical layer protocol objects define a physical layer map. Logical and physical layer maps can be abstracted into maps corresponding to geographical and administrative views. The clustering of network elements according to the views of the network at different levels of abstraction is emphasized in [25], [39].

The limitation in the size of the presentation media makes it difficult to visualize the status of the entire network. It might be impossible to squeeze a large and complex network with hundreds and thousands of devices and complex interconnections into one single screen, even if we use very sophisticated layout algorithms. When a problem occurs in the network, it would be helpful to see only the relevant areas of the network. It would be nice to be able to specify what we want in a simple and declarative way, filter out the problem area, and visualize it in terms of the topology of the network and the views at different levels of abstraction.

In this paper we propose **Hy**$^{+}$ as a front-end database visualization system for network management stations, which helps a network manager manipulate not just data, but the associated visualizations as well. With the aid of this system a network manager can also specify network management functions as declarative data manipulation statements, for example, define alerts.

1.2. Distributed Debugging

Debugging (whether for performance or correctness) and analyzing the behaviour of a parallel or distributed computation is usually performed by analyzing execution traces of a program. A complex distributed system or a parallel program with many interacting processes will produce huge volumes of trace data. We suggest using database technology to aid in the analysis of these data. We will argue that a parallel and distributed debugger should incorporate the following important and flexible facilities: *visualization*, *abstraction*, *filtering* and *layout*.

Abstraction serves two purposes: it reduces the volume of information displayed to the programmer, and it helps the programmer reason about his/her system using high-level semantics. Filtering allows the programmer to focus on the relevant or interesting portion of the original data. Visualization serves to make interesting behaviour patterns easily perceivable; but only when combined with abstraction, filtering, and sophisticated layout.

Debugging can be viewed as the detection and diagnosis of unexpected behaviour from actual behaviour at different levels of abstraction. The trace data can be viewed as a *causality graph*, as in [18], [42], or the Shoshin debugger [38], capturing information about interprocess communication events and their precedence. But a simplistic display of the causality graph to the user is impractical; there is too much information in it at too low a level. We propose instead a pattern-matching paradigm; the programmer can specify normal or abnormal patterns that he/she is looking for and, using filtering, ask the debugging tool to display them in various ways. In addition, the programmer can create new patterns at different levels of abstractions from existing data, providing alternative ways of looking at the same information. For example, one can create a *waits-for* graph, abstracted from the raw data, which makes deadlocks easy to see and whose size is proportional to the number of processes, not to the number of events in the computation.

Visualization is widely believed to be an important technique in helping debug parallel and distributed programs, but it is not obvious how and what to visualize. In the summary of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging, the remarks of Barton Miller of the University of Wisconsin during a panel on visualization are reported as follows [30]:

> ... Bart Miller stated that we don't want to visualize a large parallel program; we want to focus on the part of the program which has the problem. This requires visualization with analysis, not just visualization alone.

In this paper, we will describe the use of the $\mathbf{Hy}^+$ data visualization system for postmortem debugging of parallel and distributed programs. $\mathbf{Hy}^+$ emphasizes precisely those aspects mentioned above. Using GraphLog, we can filter raw trace data to focus on arbitrary subsets of the program; we can define new objects and relationships that embody our knowledge of the program's semantics and map the raw data into them; and we can observe program behaviour at varying levels of abstraction. With such a tool, the people who have the "intimate knowledge of the relation among processors and processes" mentioned by Miller are the ones who get to decide what is visualized and how to visualize it.

The rest of the paper is organized as follows. An overview of the $\mathbf{Hy}^+$ system and GraphLog is given in Section 2. In Section 3 we describe the use of the $\mathbf{Hy}^+$ system for network management. In Section 4 we describe distributed debugging with $\mathbf{Hy}^+$: the model of distributed computation that interests us, filtering and clustering in the process of debugging, the process of debugging with $\mathbf{Hy}^+$, the format of the traces used, and the transformation of the traces to a causality graph. We then discuss process and event abstractions and conclude the section by briefly showing how the behavior of several different versions of the *Dining Philosophers Problem* can be analyzed with the aid of the $\mathbf{Hy}^+$ system. We present comparison of our work with others in the literature in Section 5 and conclude with a discussion of future work in Section 6.

2. Overview of the $\mathbf{Hy}^+$ System

In this section, we present an overview of the $\mathbf{Hy}^+$ system developed at the University of Toronto in the context of its application within the CORDS project [34].

The $\mathbf{Hy}^+$ system provides extensive support for query visualization, visualizing the input instance, and visualizing the output instance in several different modes. The visualizations manipulated by the system are labeled graphs and *hygraphs*.

Many databases can be naturally viewed as graphs. A relational database can be represented by a directed multigraph having an edge labeled $r(c_1, \ldots, c_k)$ from a node labeled $(a_1, \ldots, a_i)$ to a node labeled $(b_1, \ldots, b_j)$ corresponding to each tuple $(a_1, \ldots, a_i, b_1, \ldots, b_j, c_1, \ldots, c_k)$ of each relation r in the database. Consider, for example, the following tuples: *connects(gateway('geol.gw'), ethernet(ether8), ip, csmacd)*, and *connects(gateway('geol.gw'), tokenring(tokenring1), ip, tokenring)*. The first tuple stores the following information: gateway *geol.gw* is connected (*connects*) to the ethernet segment *ether8* and protocols IP over CSMA/CD is run over the link. Figure 1 shows the graph representation of these tuples. Hygraphs, defined in [13], are a hybrid between Harel's higraphs [19] and directed hypergraphs.

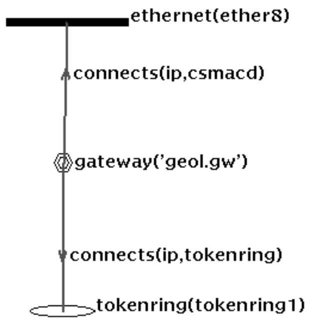

Figure 1: Visualizing tuples.

A hygraph extends the notion of a graph by incorporating *blobs* in addition to edges. A blob relates a containing node with a set of contained nodes and is diagrammatically represented as a region associated with the container node that encloses the contained nodes. Figure 2 shows a blob where the subnet (*blob node*) contains other network nodes. This containment relationship (*blob relation* or edge) *subnet_contains* is depicted as a box. The outer box may contain more than one blob relation box or other blob nodes deeply nested. Extending the representation to hygraphs allows varying levels of abstraction in the display of hierarchical data (e.g., by interactively hiding and showing blob contents) and provides a flexible mechanism for clustering information. The **Hy**$^{+}$ system has

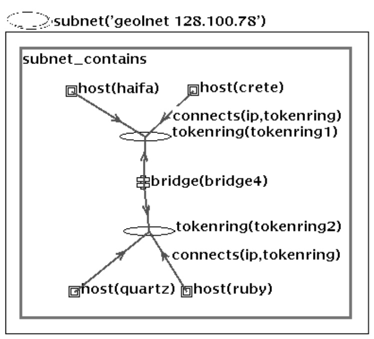

Figure 2: Visualizing a Hygraph.

browsers with extensive facilities for editing hygraphs (e.g., copy, cut, and paste; panning and zooming; executing several layout algorithms; moving hygraphs to and from files; editing of node and edge labels; etc.). These hygraph browsers can present the user with options to start browsing the structure being visualized in hypermedia-like fashion.

Figure 3 contains two **Hy**$^{+}$ views of the same visualization of an example database containing data for network management. Window Database 1.1 presents an overall view (with no labels displayed) of a graph containing several hundred objects and relationships. The schema of the database is displayed in a pane of Database 1.1 that lists the colors assigned to the edges corresponding to each of the relations in the adjacent visualization. Color-coding relations constitutes a very primitive, but visually appealing, example of tailoring the visualizations to the semantics of the applications. Another instance of the customization of the visualizations supported by the system is the selection of different icons for nodes based on the properties of the database objects represented by the nodes. Window Database 1.2 contains a zoomed-in view of the top-left portion of the graph (with labels displayed).

The visual queries supported by the **Hy**$^{+}$ system are expressions of the GraphLog query language [10]. GraphLog queries are graph patterns with nodes labeled by sequences of variables and constants and whose edges are labeled by *path regular expressions* on relations. The query evaluation process consists of finding in the database all instances of the given pattern and for each such instance performing some action, such as defining new edges, blobs, or nodes in the database graph (**Define** mode) or extracting from the database the instances of the pattern (**Show** mode). GraphLog has higher expressive power than first-order languages based on the relational algebra or calculus such as *TQuel* [36] or SQL. In particular it can express, with no need for recursion, queries that involve computing transitive closures or similar graph traversal operations. The language is also capable of expressing first-order aggregate queries as well as aggregation along path traversals (e.g., shortest path queries)[11]. The later capability is very useful for *performance debugging* of parallel and distributed programs and *traffic management* for network

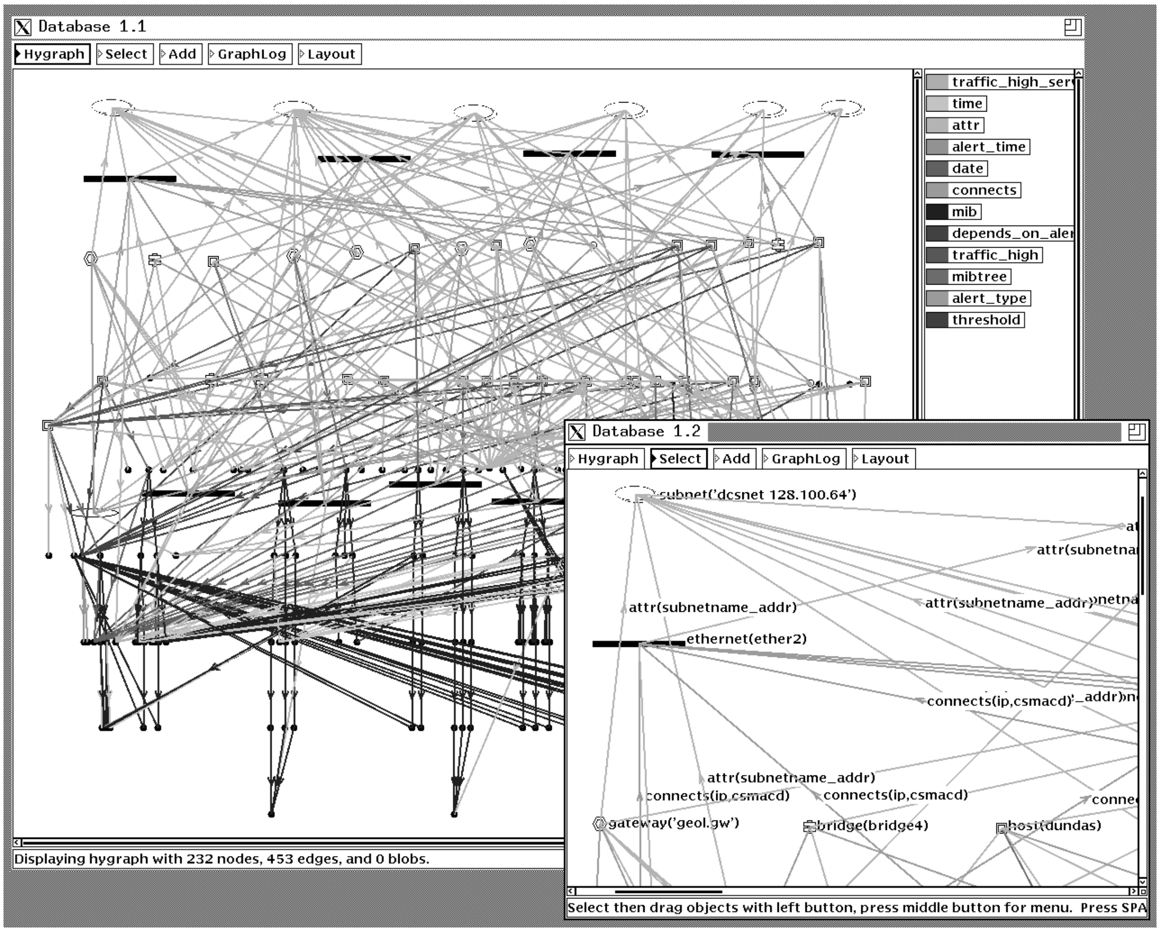

Figure 3: Browsing the example database.

management purposes.

It will be clear from the examples discussed in the paper that applications such as network management and distributed debugging require a language with the ability to express transitive closures. Once a query language supports transitive closures, its data complexity (i.e., the theoretical complexity of checking whether a tuple is in the answer to a given query, as a function of database size) will be at least in nondeterministic logarithmic space (a subset of polynomial time). GraphLog captures precisely the set of queries whose data complexity is nondeterministic logspace, and equivalently all first order queries extended with a transitive closure operator. Stated another way, it provides the user with maximum expressive power for the lowest possible computational complexity. The details of the characterizations of the expressive power of GraphLog and of its computational complexity can be found in the references cited above.

We will now give simple examples of GraphLog queries and show how the visual pattern that a user wants to see can be specified in GraphLog. If the required visualization is not in the database, it is defined using the define mode of the query and added to the database. This visualization can then be filtered out of the database using the show mode of the query. Consider the visualization of the database shown in Figure 3. This database contains enough information to create different visualizations required for a network management station. For example, we can create visualizations of the topology of the network at different levels of abstractions. To

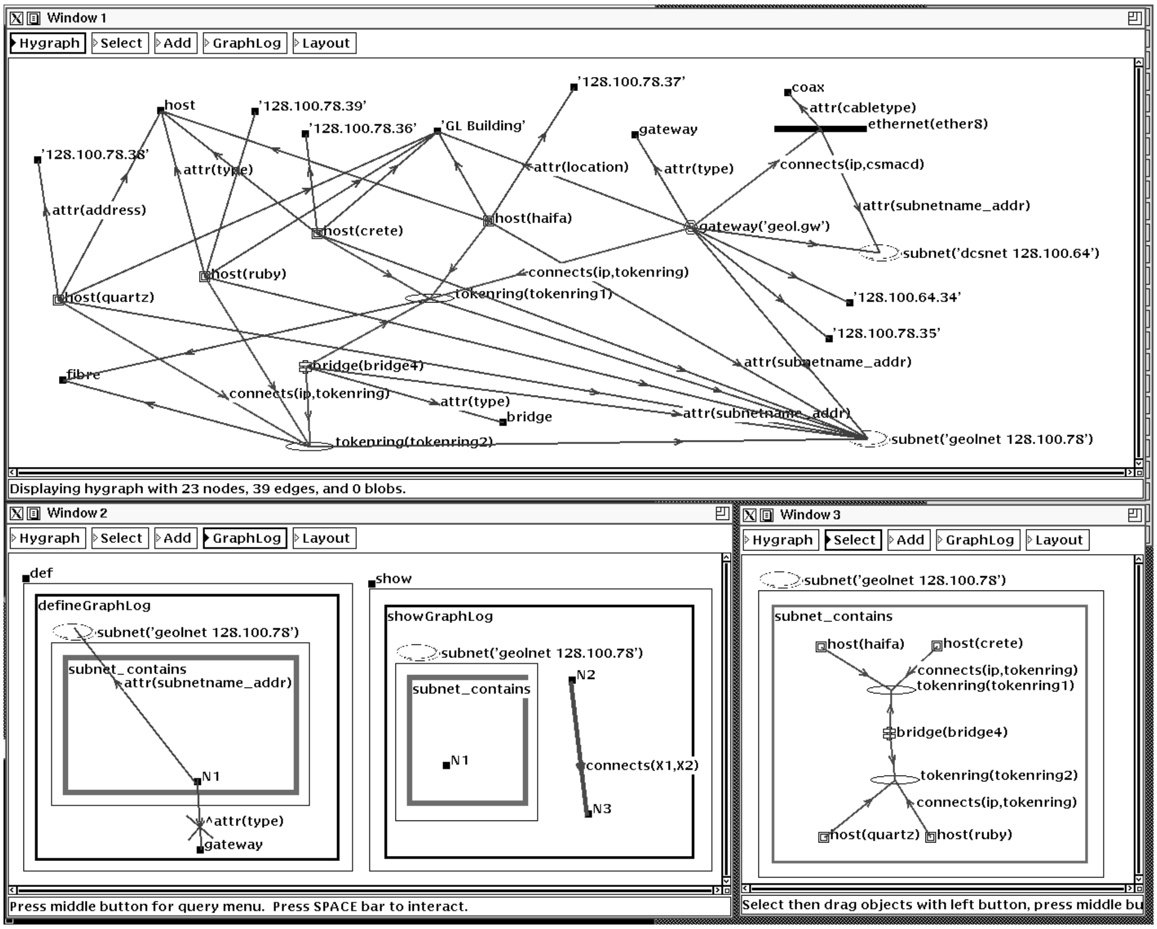

Figure 4: Example GraphLog queries and result.

simplify the example we will work on a small portion of the example database shown in Figure 3, so that the structure of the database is comprehensible. The portion of the database is shown in Window 1 of Figure 4. We want to create from this database, for example, the visualization shown in Figure 2. Query def in Window 2 of Figure 4 is executed in define mode (note the defineGraphLog box). This query states that any node N1 that has an attr(subnetname_addr) edge to the node subnet('geolnet 128.100.78') should be clustered together and blob edge called subnet_contains created. The crossed out edge labeled ^attr(type) (representing negation) excludes the gateways from being part of the resulting subnet. Note that the blob subnet_contains has been drawn thick. Any blob or edge that is drawn thick in the define mode is created and added to the database. The resulting subnet blob can be filtered out and visualized through the query show executed in show mode (note the showGraphLog box). The thick edges are the ones we want to see.

The **Hy**$^+$ visual query system is implemented as a front-end that can communicate with multiple back-ends for the actual evaluation of the queries. The front-end, written in Smalltalk, includes the user interface of the system. Currently, there are three back-end query processors: the LDL deductive database system developed at MCC [31], CORAL [32], and the third one implemented in Prolog [17].

The system has the ability to invoke external programs that, for instance, browse an object being represented by a node in one of the graphs displayed by the system. The **Hy**$^+$ visualizations are used as overviews to locate information and then invoke third-party browsers to display the contents associated with the relevant objects. The user can navigate, in hypermedia style, through the information contained in the MIB or through the code for debugging purposes. An important advantage of this approach over a purely navigational one is the ability to use the convenience and expressive power of GraphLog patterns to retrieve the objects of interest, instead of attempting an often impractical brute force search.

3. Network Management

In this section we describe what constitutes a network database and how we obtain different views of the network using the querying capabilities provided by the **Hy**$^+$ system. We will also explain how alerts can be defined using Graphlog for *event management* purposes, then we will provide an example that shows how **Hy**$^+$ visualization helps in pin-pointing faulty areas in a network. A detailed study of the use of **Hy**$^+$ for *fault management* can be found in [7].

A central database that stores all the information associated with the different views of the network and the management subsystems is a vital part of the network management station. This conceptual global database contains detailed information about the topology of the network (configuration data) and protocol related MIB objects [27] distributed over the network. The schema for each different MIB (private, enterprise, or experimental), as well as historical MIB objects, or traces, are also stored in the central database.

Using the sample network database shown in Figure 3, we will now show how the physical and logical maps can be created. By executing the queries shown in Window 1 of Figure 5 on the full database we get the physical topology map shown in Window 2 of the same figure.

The database is transformed using queries to obtain yet another different view of the network: the logical network layer map (Figure 6).

We can browse the current values and the history traces of MIB objects. For example, the query in Window 1 of Figure 7 displays (in Window 2) the history trace of MIB objects for the host named boomer. This involves filtering the values as well as the timestamps of the traces. The edge labeled with the regular expression mib(_)+ (shown dashed to emphasize that the edge in the query graph matches a path in the input graph) is used to return the entire MIB subtree for boomer.

To handle fault and performance bottlenecks, a network must be monitored continuously. Monitoring is performed normally on MIB objects. The result of monitoring is reflected in *alerts*. Defining alerts and correlating them requires sophisticated and flexible alert definition facilities. By considering the network as a distributed database and MIB as the schema of the database, alerts can be defined as declarative GraphLog queries.

Assume that we monitor the utilization of a server called samba. For this purpose we poll the tcpInSegs and tcpOutSegs TCP MIB objects of the server. The values returned are kept in the database as history traces.

An alert can be generated if the server utilization falls below a certain threshold in a particular period of time, for example, during an expected high-traffic period. If the alert goes off, it indicates a possible problem symptom. The queries in Figure 8 define the alert alert-serv1-util. Query define81 finds out the time (pointed by the distinguished edge previous_time) when the previous poll of samba was taken. Query define82 computes the rate of tcpInSegs and tcpOutSegs traffic in the latest poll interval during high-traffic time. Query define83 checks whether the rate computed in the previous query falls 80% below the expected traffic kept on the traffic_high_server edge of samba. If the condition is satisfied, an edge called problem-area is created from the node labeled samba to the node labeled with the name of the alert.

The queries define82 and define83 should be interpreted as two (active database) rules (see

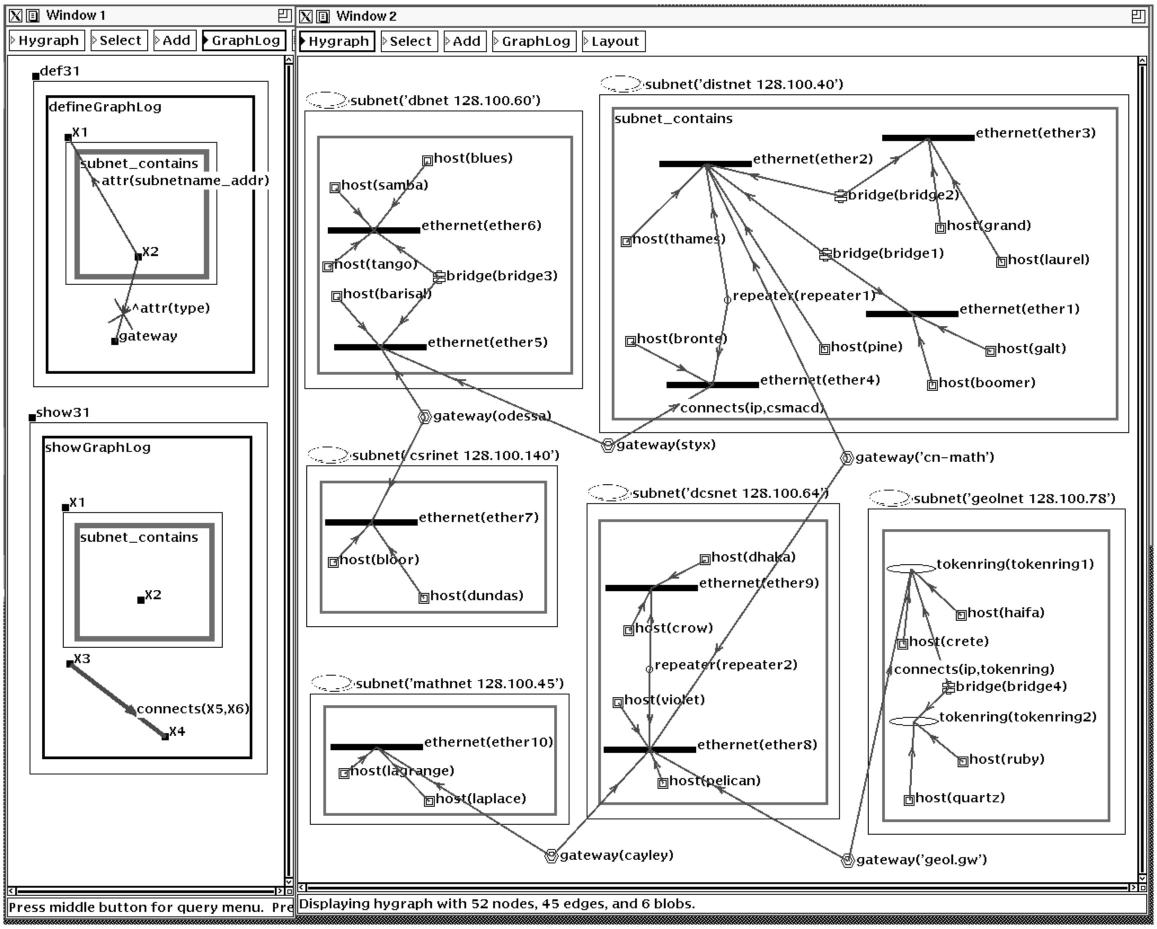

Figure 5: Defining subnets over the physical network topology.

[14, 2, 5]) that are triggered on non-empty retrieval events and perform the action of adding the distinguished edges.

Assume that the above alert has gone off and the problem area is the server samba. The alert may fire for various reasons: congestion in gateways between the server and the clients or a hardware fault.

Assume that we first hypothesized that the cause of the alert was congestion, but result of investigation (see [7]) showed no congestion, however it was found that the clients of samba are not generating the expected traffic to the server. Our second hypothesis is that the cause of the alert is due to a *hardware problem*. In this case the map showing physical details like repeaters, bridges, datalink layer protocols (CSMA/CD, tokenring), cable types (coax, fiber), and so on is brought up. Before looking for the physical causes of the problem, we superimpose the currently active TCP links on the portion of the physical map that depends on the alert.

Query define101 in Window 1 of Figure 9 can be interpreted as follows: create an edge called tcp_link between the hosts H that depend on the alert alert-serv1-util and the server samba, if the IP address found in the tcp-ConnRemAddress of H's TCP connection table matches the address of samba (which is stored in the attribute attr(address) of the configuration database). Query show101 in Window 1 produces, as a result, the visualization in window Window 2 showing the portion of the physical topology map that depends on the alert, to-

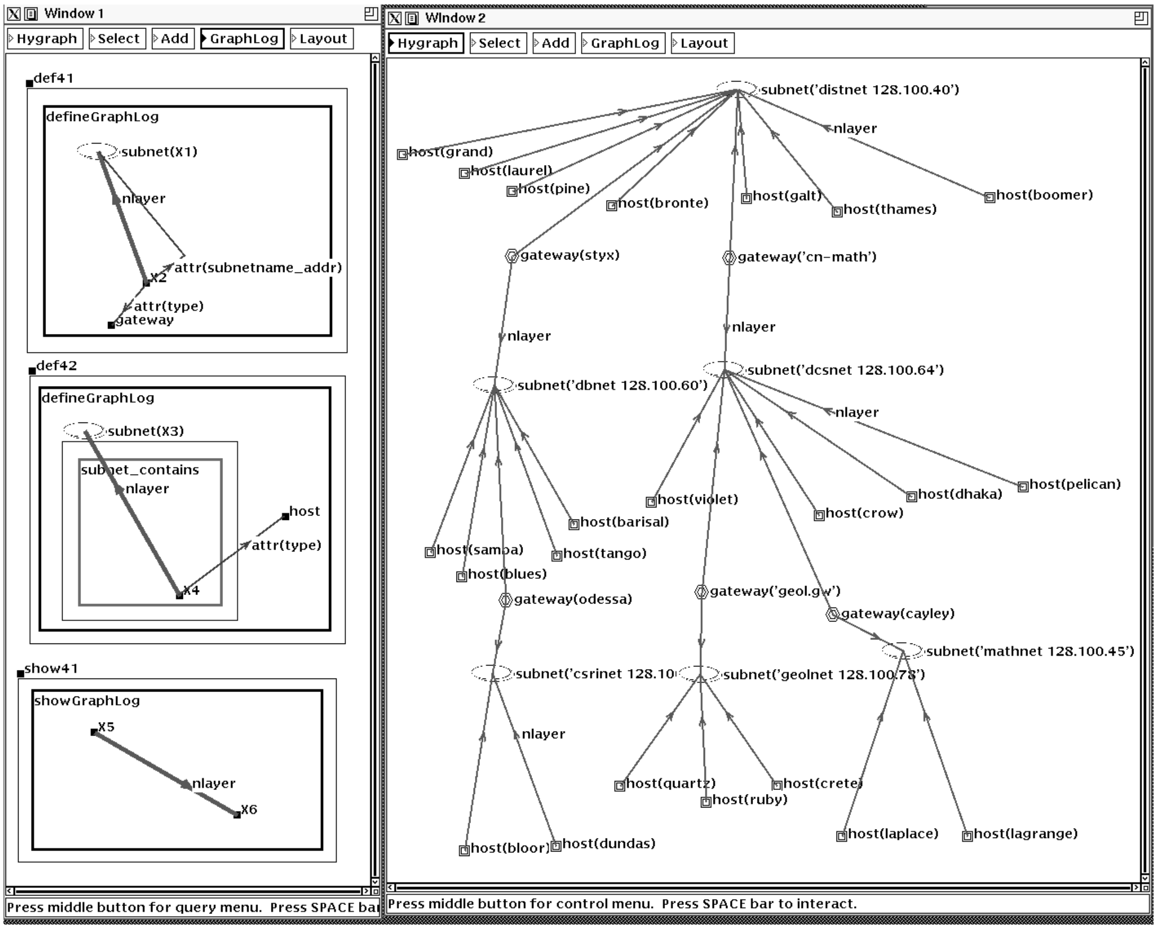

Figure 6: Defining and displaying the logical network layer map.

gether with the previously defined tcp_link edges superimposed on the map.

The visualization of TCP links allows us to pinpoint the problem area at the portion of the network beyond ether4. The problem could be at the repeater repeater1 or at the ethernet segments ether2. This follows from noticing that no TCP connection originates at ether1 or ether2. The SNMP manager could not access the current TCP objects from that part of the network.

The example shows how, when isolating faults in a network, it is advantageous to view network maps at different abstract levels, while proceeding in a structured manner to pinpoint the problem area. If we were limited to work only with the entire physical map of a large network, the complexity of the fault analysis and diagnosis procedure would increase significantly. This is just one particular instance that illustrates how the capabilities of the **Hy**$^+$ system can be exploited by human network managers.

4. Distributed Debugging with Hy$^+$

In this section we give a description of the process of postmortem debugging with **Hy**$^+$.

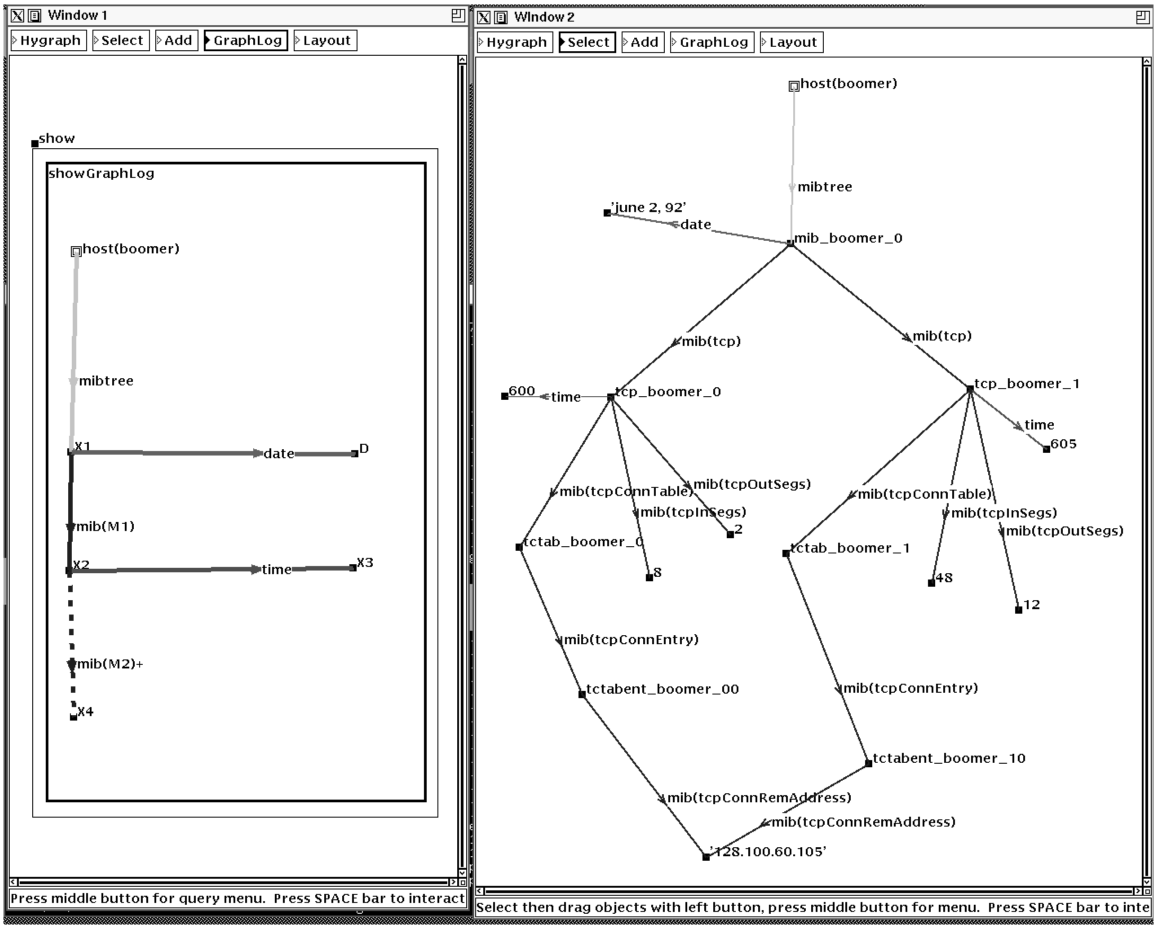

Figure 7: History trace of MIB objects for boomer.

4.1. Model of Computation

In our model, interprocess communication (IPC) is achieved only by synchronous or asynchronous message passing.

To debug a distributed system, it is necessary to monitor the execution behaviour of the system. This behaviour can be captured in terms of either the *states* or the occurrence of *events* in the system. Our system is based on the event-based model of behavior. Event-based models of behavior for debugging have been used by [1], [18], and others. An event is a uniquely identified runtime instance of an atomic action of interest [16]. We consider communication, creation and termination events as our events of interest. The communication events are the *send* and the *receive* events, and in the case of a synchronous or *rendezvous*-style communication two more events of interest are *return* or *reply* and *implied receive* of the returned message.

The set of events produced by an execution of a distributed system defines a *partial ordering*. A partial order of events in our system is determined by the total order of local events of each independent process, the precedence of a sending event in one process to the corresponding receive event in the receiving process, and the precedence of the return event to the corresponding implied receive event. Events that are incomparable in this partial order are (potentially) *concurrent*.

A partial order $P = (S, \rightarrow)$ consists of a set S called the domain of the partial order and an

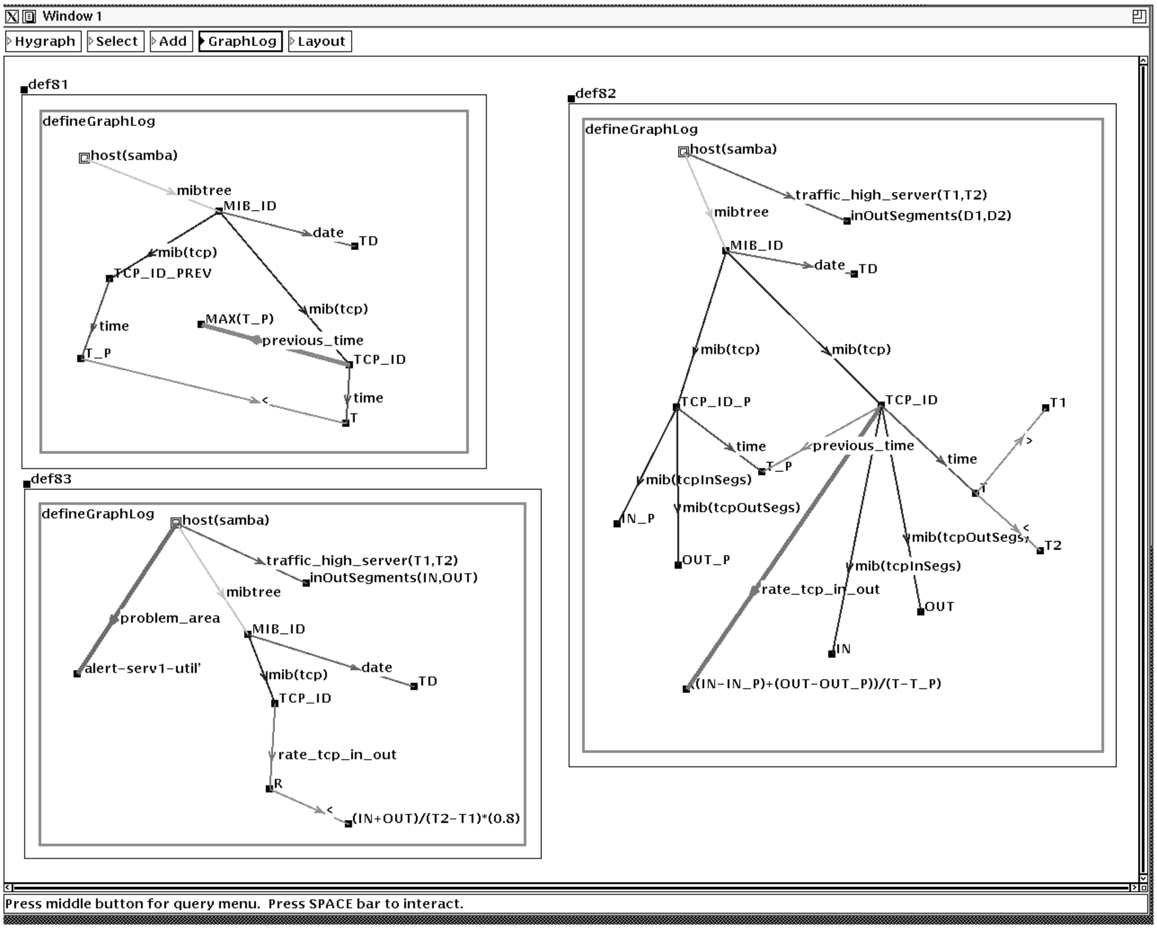

Figure 8: Defining an alert for possible problem symptoms.

irreflexive, transitive binary relation $\rightarrow$ on S. A binary relation $\rightarrow$ on S is irreflexive if, for all a in S, $a \not\rightarrow a$, and transitive if, for all a, b, c in S, and $a \rightarrow b$ and $b \rightarrow c$ imply $a \rightarrow c$.

Let $e_{i,k} \in S$ be the kth event in process i. Then we have the following:

$e_{i,k} \rightarrow e_{j,l}$, if $i = j$, and $k < l$ or if $i \neq j$ and $e_{i,k}$ is a *send* or a *return* event and $e_{j,l}$ is a *receive* or a *implied receive* event of the same message.

A partial order $P = (S, \rightarrow)$ can be viewed as a directed acyclic graph (DAG) $G = (N, E)$, where $N = S$ and (a, b) in E iff $a \rightarrow b$. That G is acyclic follows from the irreflexivity and transitivity of $\rightarrow$.

From this definition it follows that the set of events $e_{i,k} \in S$ forms a DAG, called the *causality graph*. $\mathbf{Hy}^{+}$ can be used effectively for dealing with graph structured databases. For example, the precedence relationship between two events can be decided by path traversals in a graph.

4.2. Abstraction and Filtering

As we mentioned in the Introduction, *Abstraction* and *filtering* are essential to effective visualizations. Both refer to a selective display of information. Abstraction, or clustering, reduces the amount of information displayed by reformulating the visualization at a higher level. For example, actual execution events can be grouped into higher level logical events that

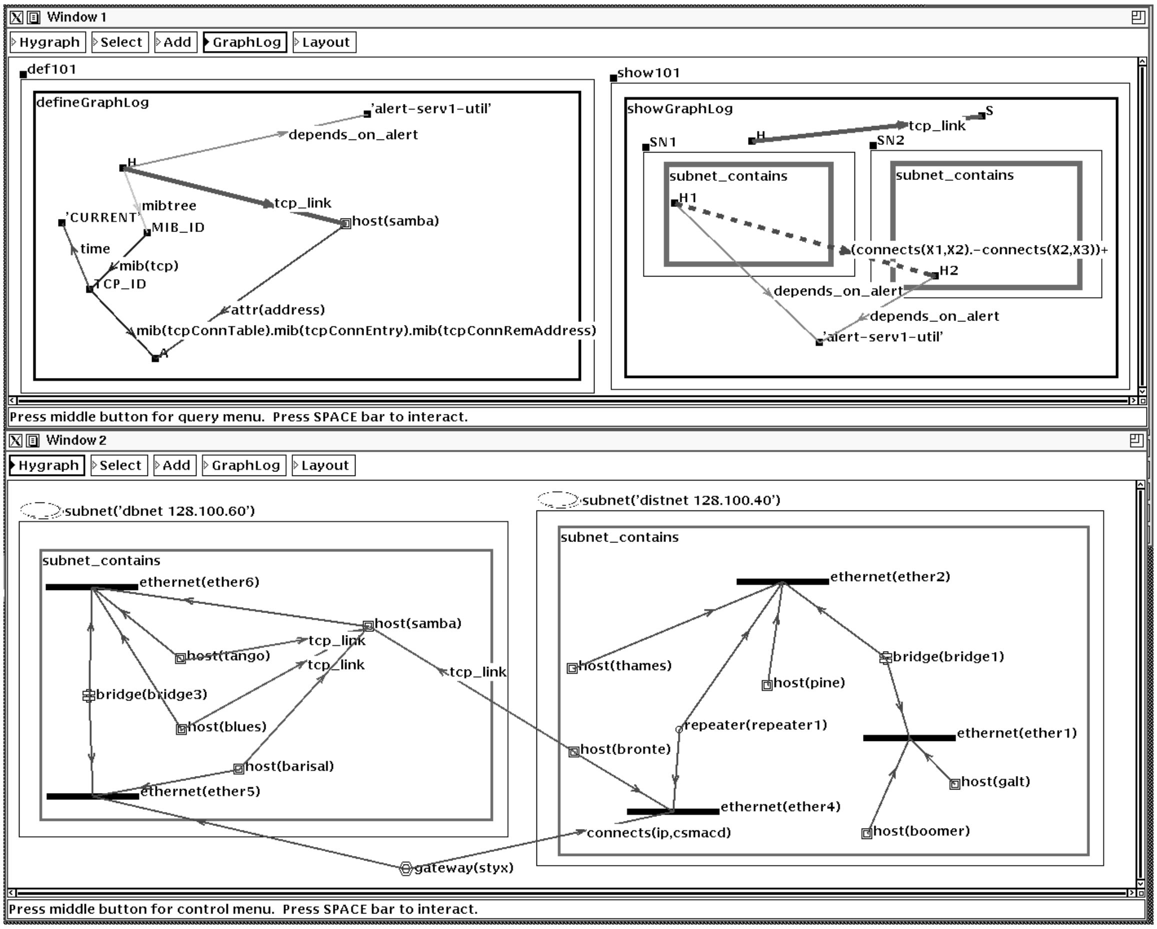

Figure 9: TCP links superimposed on the physical topology map.

make sense to the programmer. Filtering is also a reduction of the amount of information displayed, but for the purposes of focusing on a subset of data that is of current interest. For example, the programmer may want to concentrate attention on a small subset of all the processes that participated in a computation. We will refer to both abstraction and filtering generically as "filtering" in the sequel.

Filtering can be implemented in two ways: *on-line* filtering and *display* filtering. In on-line filtering, the monitor that records traces is closely integrated with the presentation module of the debugger, which instructs the monitor to only send those data that match the user's filter specification. For example, if the objective is abstraction, the primitive events are discarded and the high-level events are formed and sent to the user, as in [35], [1]. This approach reduces the amount of storage used to keep data. In our implementation we use the *display* filtering approach: all primitive event traces are stored for later use, and $\mathbf{Hy}^+$ queries are used to extract from them the information to be actually displayed.

4.3. Debugging with $\mathbf{Hy}^+$

We start the debugging process with a trace of program execution that could be generated by an instrumented version of the IPC package being used. In our experiments, we used traces produced by the IPC event collection modules of the *Panorama* [26] monitor. The steps of debugging with $\mathbf{Hy}^+$ are the following.

- Treat the traces of primitive communication, process creation, and termination events as tuples in the database.
- Create the causality graph through the use of simple GraphLog queries.
- Iterate through the following steps, in any order:
  - Define process and event abstractions declaratively at any level of hierarchy by creating hygraphs from the trace database and the causality graph.
  - Specify program behaviour as hygraph patterns (at any level of abstraction) expressed through GraphLog queries.
  - Focus on relevant information using GraphLog's filtering capability and control level of detail by interactively hiding and revealing blob contents.
  - Experiment with the layout algorithms provided by the system to discover communication and behaviour patterns in a program.

4.4. Description of Traces

In this subsection we describe the trace formats we obtained from running Hermes [37] programs. We shall use Hermes terminology for describing communication events, but the ideas are applicable in general, and the system is independent of any particular trace format.

In Hermes, synchronous communication is achieved by a call, which blocks the caller until the callee returns. There is also a send operation for asynchronous communication. A primitive event trace has the following format:

```
comm_event(originator_process_name,
           destination_process_name,
           originator_process_id,
           destination_process_id,
           originator_event_sequence_no,
           destination_event_sequence_no,
           service_name, programmer_defined_id,
           message)
```

where comm_event stands for one of call, send, receive or receive_implied. (A receive_implied event is assumed to occur after each return.) Other events that may appear in the trace, with a slightly different format, are create and end.

The following are examples of two event traces, taken from a "Dining Philosophers" program described later:

receive(philo, fork, 205, 196, 19, 10, getFork, 2, msg(3,yes)).

call(philo, fork, 196, 205, 12, freeFork, 2, msg(3)).

For example, the first event is a receive by process 205, of type philo, from a corresponding call by process 196, of type fork, it is the 19th event within the philo process and the 10th event within the fork process. The receive relates to a getFork service requested by the caller. The 2 is a programmer defined identifier (in this case, the sequence number of the requested fork) and msg(3,yes) is information associated with the message whose interpretation is application dependent. Note the second event, which is a call, has no destination sequence number, since this number is unknown at the time of the call.

4.5. Transforming the trace database

The trace as described above is the raw data that $\mathbf{Hy}^+$ takes as input. The window labelled *Original Trace Database* in Figure 10 shows the graphical representation of the initial trace database. The default interpretation of a tuple like p(X,Y,Z) in $\mathbf{Hy}^+$ is as an edge from node X to node Y labelled p(Z); this is not very useful here, since the first two arguments of the communication event tuples are process types, not individual processes. Also, the long list of parameters associated with each event clutters the diagram, the layout superimposes many of the nodes and edges, and it is difficult to understand the relationship of the diagram to the program's execution.

To reshape the information into usable form, we first issue a query containing six define-

GraphLog boxes to define logical concepts of interest to the programmer. The box labelled define21 creates a node with a distinctive "process" icon for each process instance and associates with it the process type, process id, and programmer-defined identifier. The boxes labelled define22 and define23 define event nodes and relate each process instance to all the events within that process instance by means of events edges. The box labelled define24 defines a new kind of edge that relates each call event within a calling process to the corresponding receive event in the callee. Finally, boxes define25 and define26 define the precedence relationship between events corresponding to the partial order defined in Section 4.1.

Note that these transformations are not application dependent, they only depend on the trace format. Once formulated, they can easily be canned and applied to execution traces of any application.

4.6. Process and Event Abstraction

Now that we have reformulated the raw data, we are in a position to do some clustering. Figure 11 shows clustering of all events within a process P into a process blob owned by P (box define11). Note the events within each process will be linearly related by the precedes edges we defined in the last subsection, thus simulating the *timeline* in a *process-time* diagram. The "show" box labelled show12 in Figure 11 asks the system to display all the process nodes, their process_events blobs, the precedes edges within the blobs, and the IPC comm edges between events. The Result 1.2 window is the result. The process blobs can themselves be clustered together if desired, thus achieving any level of abstraction. In Figure 11 seven of the ten blobs have been collapsed and their contents hidden to simplify the visualization and eliminate distractions.

An *abstract event* is a higher-level event consisting of primitive or other abstract events. For example, the set of communication events send, receive, return, and implied receive corresponding to a rendezvous-style synchronous event can be collapsed into a single abstract event. An example of event abstraction will be shown in a later section.

4.7. Case Study: The Dining Philosophers Problem

The examples we have been using come from a Hermes implementation of a well known problem in concurrent programming, the *Dining Philosophers Problem*: N philosophers sit around a round table. There are N forks on the table. To be able to eat a philosopher must pick up the forks that are at his left and right. In a more elaborate version of the problem a philosopher may eat many times until his plate (of spaghetti) is finished. Between two eating events he releases the acquired forks, then thinks for awhile.

Let's see how a naive programmer might use **Hy**$^+$ to help deal with problems such as deadlock and starvation while trying to solve this problem. Our programmer starts with the simple solution below.

```
process fork(i):
    get_fork:
        receive()
        if fork available
            available := no
            --give the fork to the
            --requesting philo
            return
        else
            --Block (do not return)
    free_fork:
        receive()
        available := yes
        return

process philo(i):
    --left fork
    call fork.get_fork(i)
    --right fork
    call fork.get_fork((i+1))
    eat
    call fork.free_fork(i)
    call fork.free_fork((i+1) mod N)
```

The main process (which is not shown above)

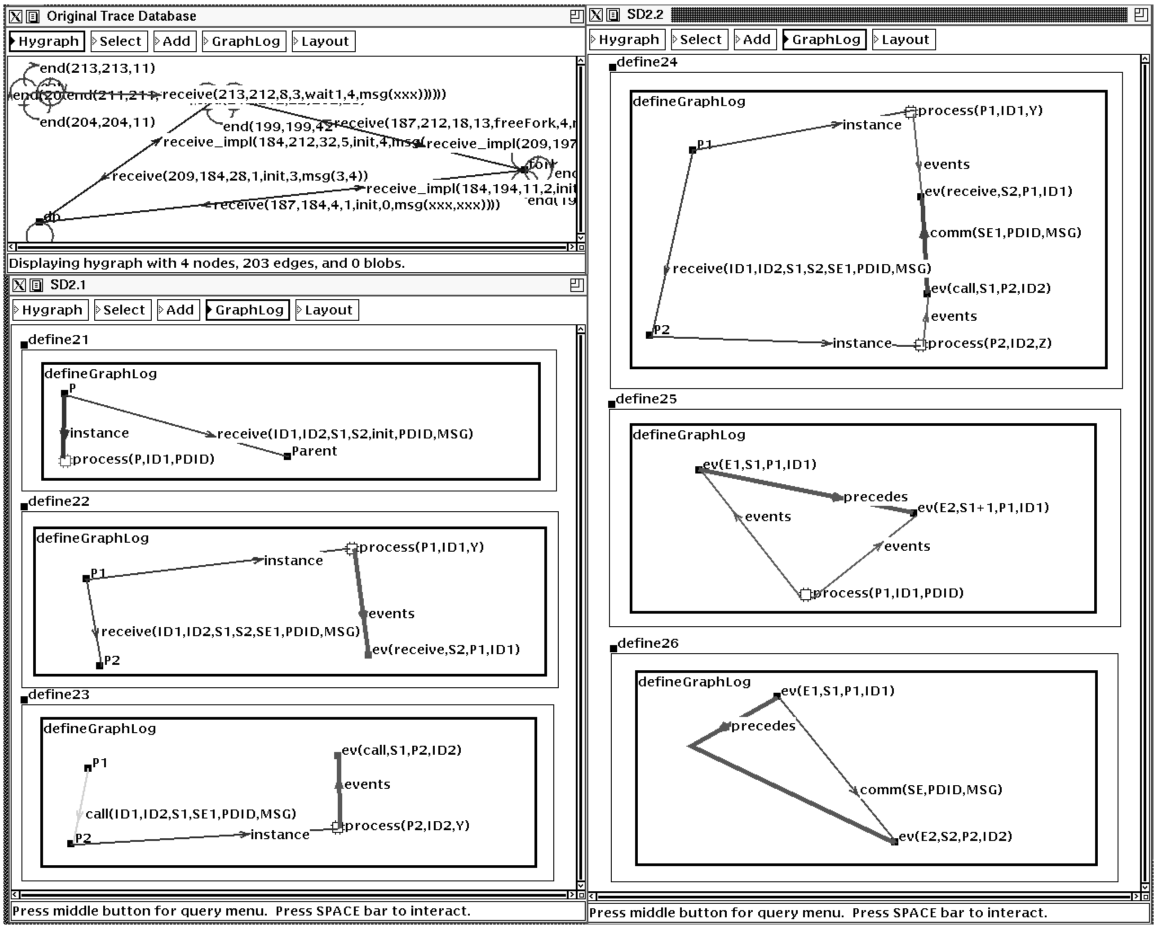

Figure 10: Transformation of the initial trace database.

creates N instances of the philo and the fork processes. The fork processes provide two services: get_fork and free_fork.

There is a bug in this program; it wrongly specifies the right fork of a philosopher by writing (i+1) instead of (i+1) mod N in line 18 above. Figure 12 shows how this bug manifests itself in a $\mathbf{Hy}^+$ visualization. The hole in the left-right fork initialization pattern in window Result 4.2 indicates that the last philosopher was not initialized with the correct forks by the main process. Correcting the bug produces instead the window Result 4.3 with the correct initialization pattern.

The program that results from correcting the first bug may cause a deadlock. Figure 13 shows the detection of deadlock in the second version of the program. The query define51 in window SD5.1 in Figure 13 creates abstract events by collapsing the send, receive, return, and receive_implied events into one single blob and query define52 in window SD5.1 defines the abstract precedence abs_precedes between abstract get_fork or free_fork events. This is used in window define53 to define a waits_for edge. The cycle in the waits_for graph in window Result 5.3 is an obvious indication of a deadlock.

The programmer now generates a third version by introducing a “fair policy” in the hope that it might avert the deadlock. In this solution, line 7 is replaced with return(“yes”) and line 9 with return(“no”), indicating explicitly that the fork is not available, so that the philo process frees the first acquired fork. A corresponding modification is introduced in the

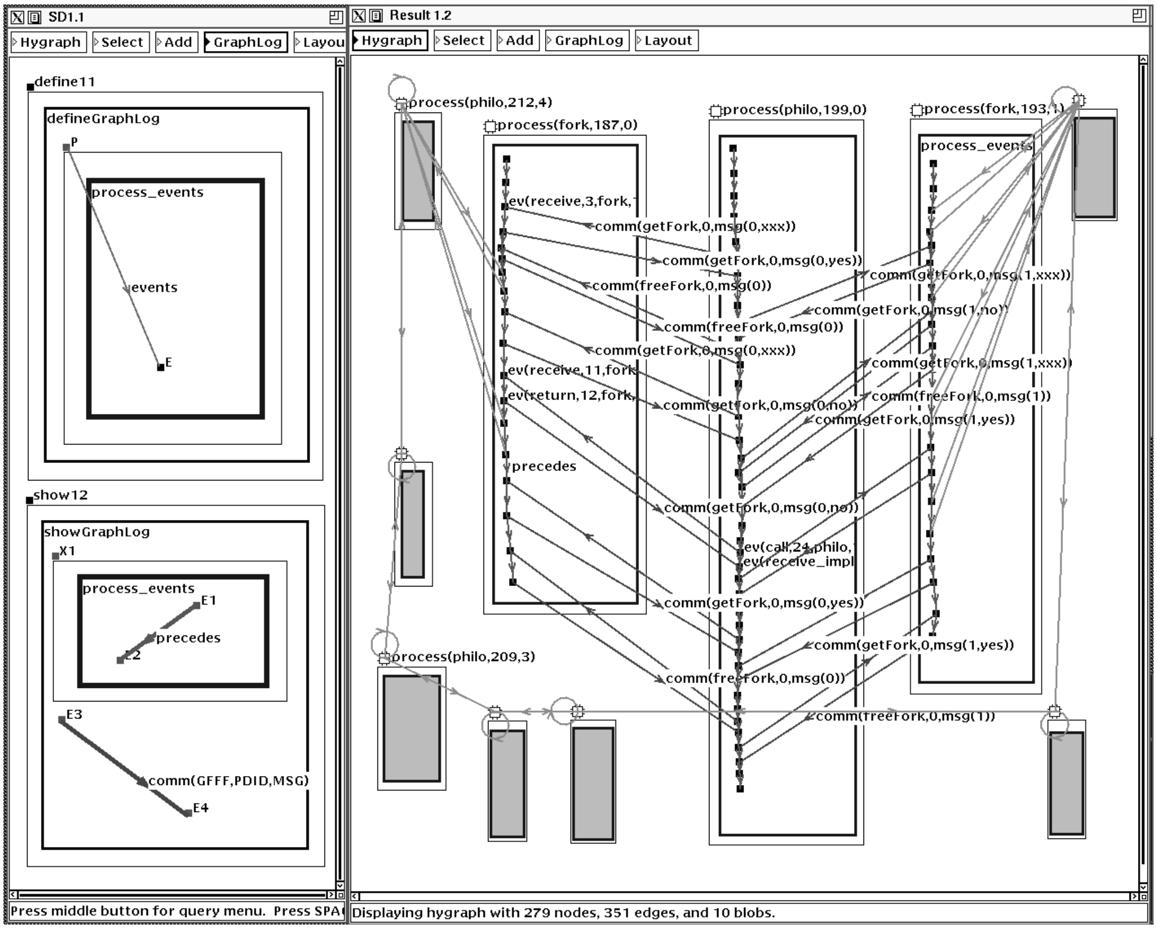

Figure 11: Event trace hygraphs.

philo process. In this version, all the philosophers succeed in eating. In Figure 14 two new edges are defined, eat_precedes, meaning one philosopher eats before another one does, and conc_eat, meaning that two philosophers eat concurrently. The visualization of conc_eat in window Result 8.3 shows that no neighbouring philosophers eat concurrently, which would be an error, meaning, for example, that fork availability is not correctly checked.

In [8], we show how several other potential error conditions can be declaratively formulated and visualized with $\mathbf{Hy}^+$.

5. Related Work

The $\mathbf{Hy}^+$ system has similarities to the heterogeneous network management systems proposed in [40, 15] and to the active database approach in [41], where network management functions are specified as data manipulation statements. These systems differ from $\mathbf{Hy}^+$ in the emphasis given by the latter to the manipulation of database visualizations as an integral part of a toolset for network management stations. Another important difference is the expressive power of the GraphLog query language. SQL-based systems are not capable of expressing transitive closure queries, while these are directly supported in the GraphLog query language in a very intuitive way. Transitive closure queries are necessary when we want to retrieve

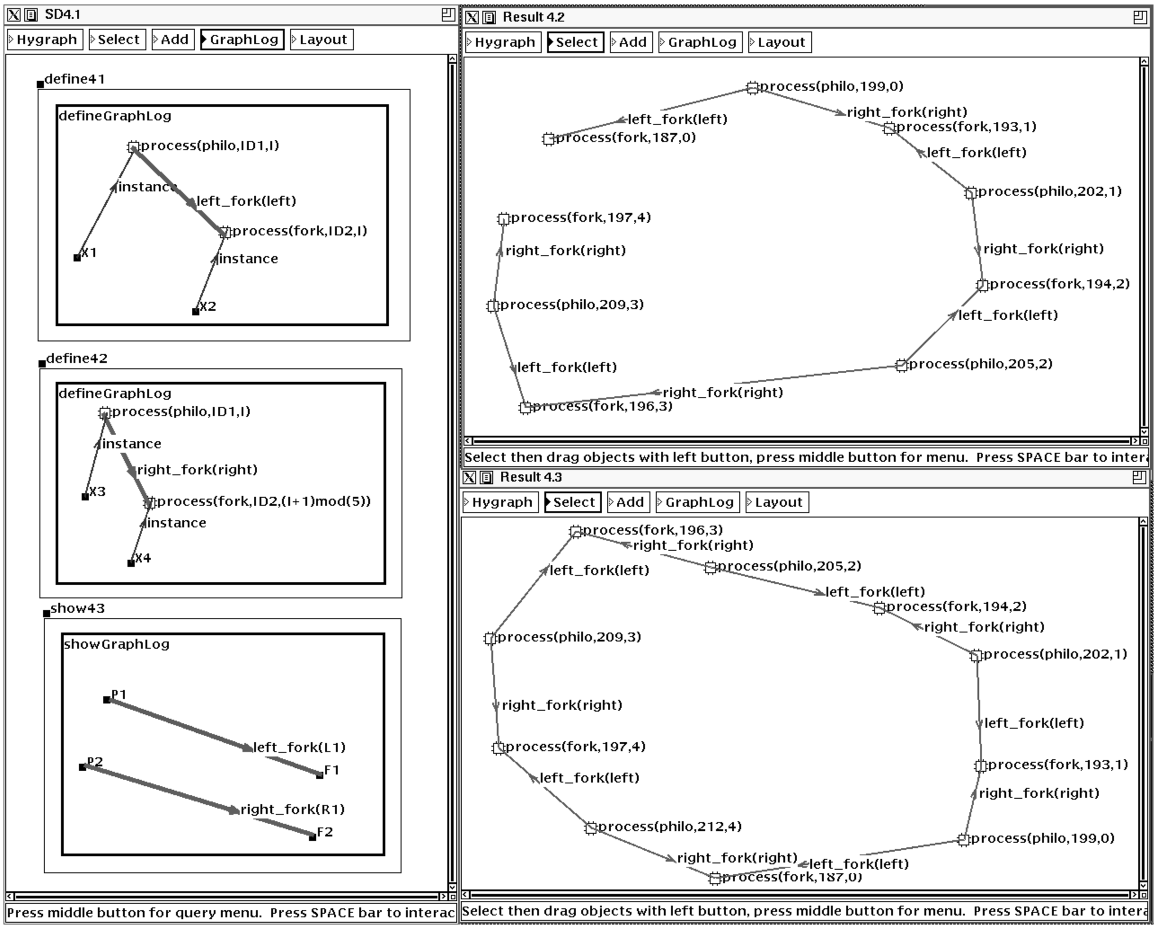

Figure 12: Left-right fork cycle.

network management information in terms of the connectivity and topology of the network.

An examination of the survey [28] reveals that most debugging systems do not provide all the facilities provided by the **Hy**$^+$ system into one single debugger.

Hy$^+$ debugger combines to a certain extent the methods proposed in [42], [35], and [1]. [42] does not provide a querying capability. The abstraction, filtering and visualization capabilities provided by [42] are also limited. The EBBA system [1] provides higher-level event abstraction mechanism, but ignores precedence information and lacks visualization capability provided by **Hy**$^+$. The query language TQuel in [35] is less expressive than GraphLog and the visualization aspects in the line of **Hy**$^+$ are missing in TQuel. Furthermore, the visual and graphical nature of GraphLog queries allow the debugger to express intuitively appealing queries. Snodgrass's monitor assumes a global clock, which does not give an accurate picture of a distributed computation.

6. Conclusion

We have described how the **Hy**$^+$ data visualization system can be used both for network management and postmortem debugging of parallel and distributed programs. We have emphasized abstraction, filtering, and layout control as prerequisites for a useful visualization system.

These preliminary results suggest that a sys-

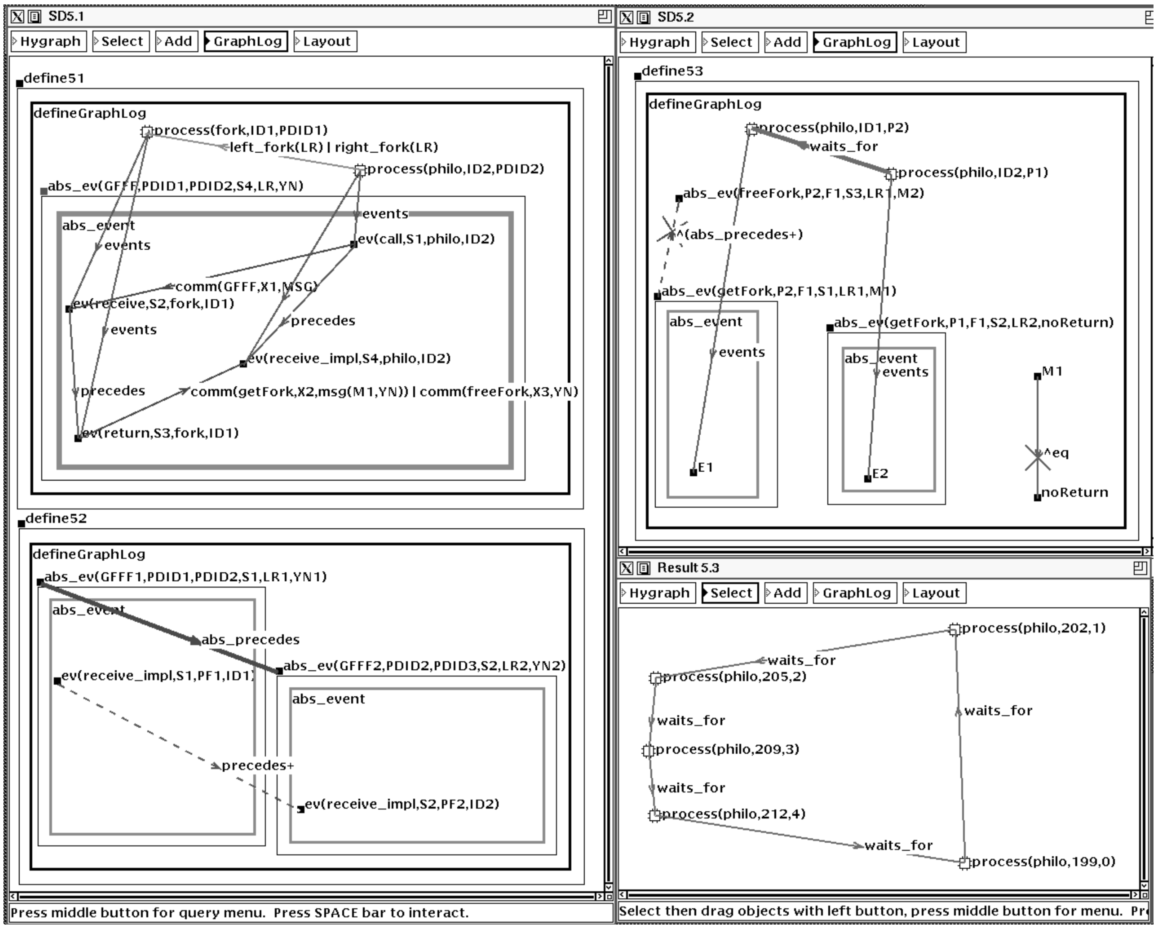

Figure 13: Detecting deadlock.

tem like **Hy**$^+$ can serve as a common front-end tool for the effective development, maintenance and operation of distributed applications. Network management and program monitoring are of course not unrelated: program behaviour can be observed not just in terms of the semantics of the program, but also in terms of the underlying network on which the distributed application is running. Testing and performance tuning of a distributed application can benefit from the facilities provided by network management. For example, communication links in an application (like the comm edges of Figure 11) can be mapped to the *network routes* the links are going over and the resulting network routes visualized. The information about the characteristics of the network routes can help the programmer to tune the performance of the program (i.e., by selecting a different route by assigning processes on some other processors). The capability of GraphLog to express first-order aggregate queries as well as aggregation along path traversals will be of great help for performance debugging or measurements of parallel and distributed programs.

7. Future Work

The expressive power of GraphLog can be enhanced by adding a temporal dimension. GraphLog augmented with temporal operators will be able to express important temporal properties of concurrent programs. Work is in progress to define graphical or GraphLog equivalents of temporal operators (GraphLog TL) of

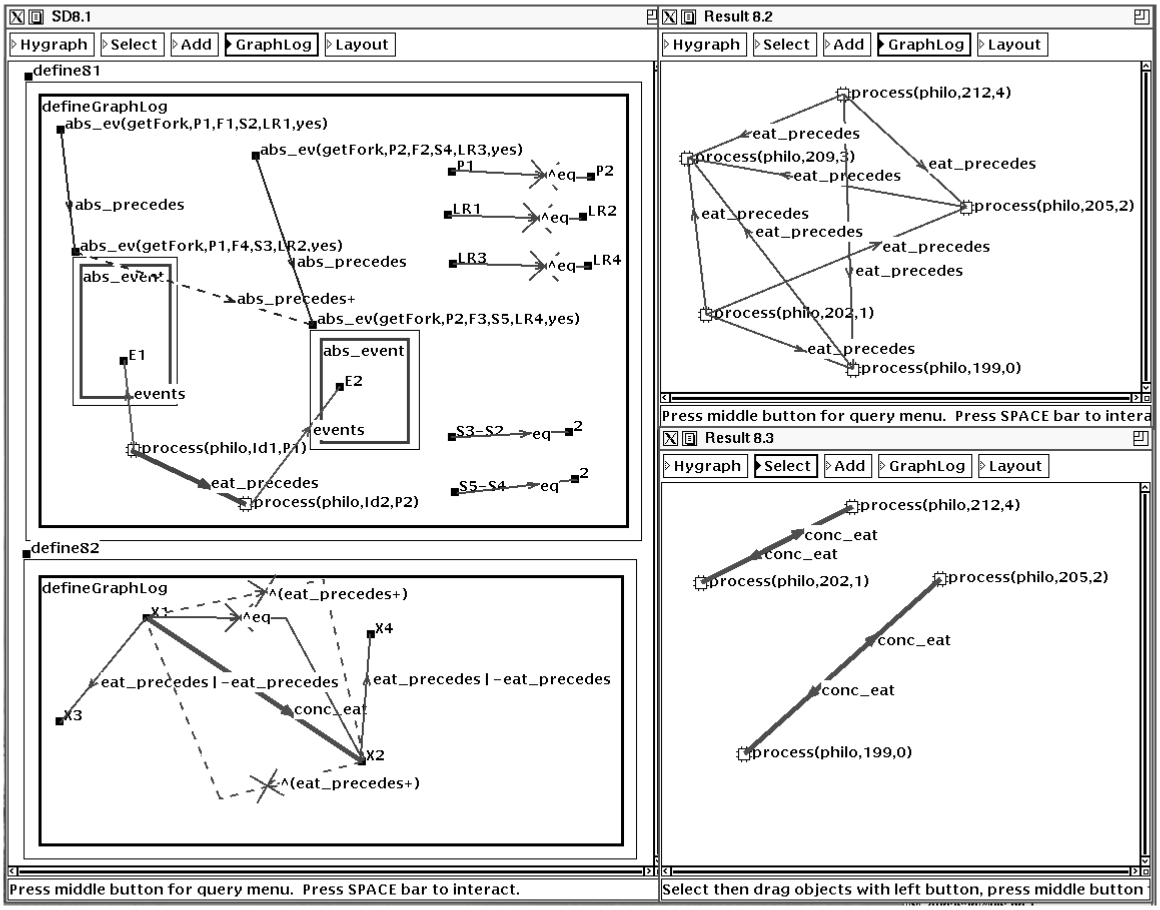

Figure 14: Precedence of eating events.

computation tree logic [6] and *real-time interval temporal logic* [29]. With this addition, it will be possible to not only reason about execution traces of a program written in a particular language, but also model a distributed problem using formal description techniques (FDT), such as *LOTOS* [24], *communicating finite state machines* [3], and *CSP* [20], generate the global state graph from the model, and then specify in GraphLog TL properties like deadlock-freedom or starvation-freedom.

We would also like to move from post-mortem to on-line debugging in $\mathbf{Hy}^+$. We plan to connect the system to an event monitor and show in (quasi) real time the result of continuously querying the incoming traces. In this way, we can dynamically obtain an evolving picture of the computation under observation, using GraphLog patterns to control the visualization. The capability to deal with a dynamically changing database in combination with GraphLog extended with temporal operators such as *real-time interval temporal logic* would make $\mathbf{Hy}^+$ a powerful tool for both interactive debugging and network management, and any other application where temporal monitoring of graph-like databases (as proposed in [41]) is required.

About the Authors

Mariano P. Consens is currently a Research Associate and a PhD student at the University of Toronto. His research interests are in the theory and practice of database and knowledge base systems and in the visualization of large amounts of data (in particular for application domains such as software engineering and network management). Consens obtained a Computer Engineering degree at the University of Uruguay, and a Master's degree in Computer Science at the University of Toronto. He is a member of ACM and IEEE.

Masum Hasan is a Research Associate at CSRI, University of Toronto and a Ph.D candidate in the Department of Computer Science, University of Waterloo. He obtained BEng, MEng in Computer Engineering from former USSR and MMath in Computer Science from University of Waterloo. His research interests are in distributed and parallel programming environment, network management, active databases, database and real-time data visualization.

Alberto Mendelzon is a Professor of Computer Science and Associate Director of the Computer Systems Research Institute at the University of Toronto. His interests are in data visualization, query languages, and the theory of databases and knowledge bases.

The authors can be reached at CSRI, University of Toronto, 6 King's College Road, Toronto, Ontario M5S 1A1, or via e-mail at {consens, zmhasan, mendel}@db.toronto.edu.

References

[1] Peter Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):11–22, January 1989.

[2] Catriel Beeri and Tova Milo. A model for active object oriented database. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 337–349, September 1991.

[3] D. Brad and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, April 1983.

[4] J. D. Case, M. S. Fedor, M. L. Schoffstal, and J. R. Davin. A simple network management protocol (SNMP). RFC 1157, SNMP Research, Performance Systems International, MIT Laboratory for Computer Science, May 1990.

[5] Stefano Ceri. A declarative approach to active databases. In *Proceedings of the Data Engineering*, pages 452–456, 1992.

[6] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):245–263, April 1986.

[7] Mariano Consens and Masum Hasan. Supporting network management through declaratively specified data visualizations. In H.G. Hegering and Y. Yemini, editors, *Proceedings of the IEEE/IFIP Third International Symposium on Integrated Network Management, III*, pages 725–738. Elsevier North Holland, April 1993.

[8] Mariano Consens, Masum Hasan, and Alberto Mendelzon. Debugging distributed and parallel programs by visualizing and querying event traces. In A.O. Mendelzon, editor, *Declarative database visualization: recent papers from the Hy+/GraphLog project*. CSRI, University of Toronto, June 1993. Technical Report CSRI-285.

[9] Mariano Consens, Masum Hasan, and Alberto Mendelzon. Debugging distributed programs by visualizing and querying event traces. In *Proceedings of the 3rd ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 181–183, May 1993. Extended abstract.

[10] Mariano Consens and Alberto Mendelzon. GraphLog: a visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 404–416, 1990.

[11] Mariano Consens and Alberto Mendelzon. Low complexity aggregation in GraphLog and Datalog. In *Proceedings*

of the Third International Conference on Database Theory, Lecture Notes in Computer Science Nr. 470, pages 379–394. Springer-Verlag, 1990. A revised version has been accepted for publication in TCS.

[12] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *14th. Intl. Conference on Software Engineering*, pages 138–156, 1992.

[13] Mariano P. Consens. Visual manipulations of database visualizations. PhD Thesis Proposal, 1992.

[14] U. Dayal. Active database management systems. In *Proceedings of the third International Conference on Management of Data*, pages 225–234, June 1988.

[15] A. Dupuy, S. Sengupta, O. Wolfson, and Y. Yemini. NETMATE: A network management environment. *IEEE Network Magazine*, pages 35–43, October 1991.

[16] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, pages 28–33, August 1991.

[17] Milan Fukar. Translating GraphLog into Prolog. Technical report, Center for Advanced Studies IBM Canada Limited, October 1991.

[18] A.J. Gordon and R.A. Finkel. Handling timing errors in distributed programs. *IEEE Transactions on Software Engineering*, 14(10):1525–1535, October 1988.

[19] David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[20] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[21] ISO 9596, information processing systems, open systems interconnection, management information protocol specification, common management information protocol. International Organization for Standardization, November 1990.

[22] ISO 7498, information processing systems, open systems interconnection, basic reference model part 4, OSI management framework. International Organization for Standardization, October 1986.

[23] ISO/IEC DIS 10165-1, 10165-2 and 10165-4, information processing systems, open systems interconnection, structure of management information, parts 1, 2 and 4. International Organization for Standardization, 1990.

[24] ISO 9596, information processing systems, open systems interconnection, LOTOS - a formal description technique based on the temporal ordering of observational bahavior, ISO/TC97/SC21 DP 8807. International Organization for Standardization, June 1985.

[25] G. Kar, B. Madden, and R. S. Gilbert. Heuristic layout algorithms for network management presentation services. *IEEE Network Magazine*, pages 29–36, November 1988.

[26] W. Korfhage. The panorama monitoring system. Technical report, Polytechnic University, Brooklyn, 1992.

[27] K. McCloghrie and M. T. Rose. Management information base for network management of TCP/IP based internets - MIB-II. RFC 1213, Hughes LAN Systems, Performance Systems International, March, 1991.

[28] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[29] P.M. Melliar-Smith. Extending interval logic to real time systems. In B. Banieqbal, H. Barringer, and A. Pnueli, editors, *Temporal Logic in Specification, LNCS 398*. Springler-Verlag, 1987.

[30] B. Miller. Panel discussion: Parallel program visualization: Useful tool or cheap thrill? *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):6–8, December 1991.

[31] Shamim Naqvi and Tsur Shalom. *A logical language for data and knowledge bases.* Computer Science Press, New York, 1989.

[32] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proceedings of International Conference on Very Large Databases*, 1992.

[33] M. Rose and K. McCloghrie. Structure and indentification of management information for TCP/IP-based internets. RFC 1155, Performance Systems International, Hughes LAN Systems, May, 1990.

[34] J. Slonim, M. Bauer, P. Finnigan, P. Larson, A. Mendelzon, R. McBride, T. Teorey, Y. Yemini, and S. Yemini. Towards a new distributed programming environment (CORDS). In *Proceedings of the 1991 CAS Conference*, Oct.. 1991.

[35] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

[36] R.T. Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[37] R.E. Strom, D.F. Bacon, A.P. Goldberg, A. Lowry, D.M. Yellin, and S.A. Yemini. *HERMES: A Language for Distributed Computing.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[38] David Taylor. A prototype debugger for Hermes. In *Proceedings of the 1992 CAS Conference*. IBM Canada Centre for Advanced Studies, November 1992.

[39] R. F. Valta. Design concepts for a global network management database. *Integrated Network Management, II*, pages 777–788, 1991.

[40] U. S. Warrier and Carl A. Sunshine. A platform for heterogeneous interconnection network management. In B. Meandzija and J. Westcott, editors, *Integrated Network Management, I, IFIP*, chapter 1, pages 13–24. North-Holland, 1989.

[41] O. Wolfson, S. Sengupta, and Y. Yemini. Managing communication networks by monitoring databases. *IEEE Transactions on Software Engineering*, 17(9):944–953, September 1991.

[42] Dror Zernik and Larry Rudolph. Animating work and time for debugging parallel programs foundation and experience. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 26(12):46–56, December 1991.