Visualizing and Querying Distributed Event Traces with $\mathbf{Hy}^+$

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

Computer Systems Research Institute, University of Toronto, 6 King's College Road, Toronto, Ontario, canada, M5S 1A1

Abstract. A programmer attempting to understand and debug a distributed program deals with large volumes of trace data that describe the program's behaviour. Visualization is widely believed to help in this and similar tasks. We contend that visualization is indeed useful, but only if accompanied of powerful data management facilities to support abstraction and filtering. 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 distributed program debuggers to meet their information manipulation and visualization goals.

The $\mathbf{Hy}^+$/GraphLog system can be used for observing behaviour 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 behaviour of the program.

1 Introduction

Distributed and parallel program debugging requires the analysis of large volumes of trace data. We show that a system that provides a convenient marriage of deductive databases and visualization technology is well suited for the specification and presentation services required for analyzing the behaviour of distributed and parallel programs. To understand the behavior of a complex system, visualization techniques are widely used. But simple and fixed ways of visualizing data are not well suited to our intended application domain. Our proposal allows the programmer not only to specify visualizations through declarative queries, but also to store and manipulate visualizations like data are manipulated in a database. These queries provide a powerful mechanism for *abstraction*, that is, defining higher level objects that match the programmer's understanding of the computation, and for *filtering*, that is, focusing on particular subsets of the trace data.

The $\mathbf{Hy}^+$/GraphLog system developed at the University of Toronto is a general purpose environment for visualizing structural data that has been applied to areas such as software engineering [6], hypertext [3] and network management [2]. GraphLog [4] is the visual query language for the $\mathbf{Hy}^+$ system.

Debugging can be viewed as the detection and diagnosis of unexpected behaviour by examining actual behaviour at different levels of abstraction. The trace data can be viewed as a *causality graph*, as in [18], 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.

In this paper, we will describe the use of the **Hy$^+$** data visualization system for postmortem debugging of parallel and distributed programs. **Hy$^+$** emphasizes visualization with analysis. 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 actually know the meaning of the computation are the ones who get to decide what is visualized and how to visualize it.

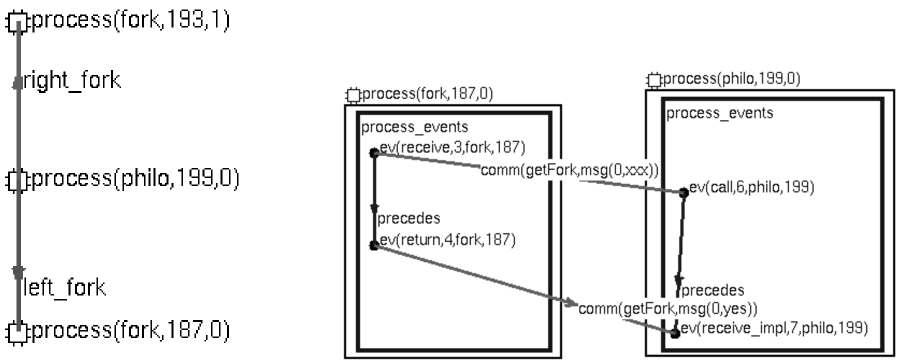
The rest of the paper is organized as follows. An overview of the **Hy$^+$** system and GraphLog is given in Section 2. In Section 3 we describe the methods of distributed trace analysis, and causality graph. The process of trace analysis with **Hy$^+$**, the format of the traces used, the transformation of the traces to a causality graph, and the process and event abstraction with **Hy$^+$** are discussed in Section 4. We then discuss in Section 5, how the behavior of several different versions of the *Dining Philosophers Problem* has been analyzed with the aid of the **Hy$^+$** system. Section 6 provides a brief overview of dynamic analysis of traces with **Hy$^+$**. We then conclude with a discussion of future work in Section 7.

2 Overview of the Hy$^+$ System

The **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* [7].

Each tuple $(a_1, \ldots, a_i, b_1, \ldots, b_j,\ c_1, \ldots, c_k)$ in 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 of each relation r in the database.

For example, suppose we have a database describing an execution of the *dining philosophers* program described in Section 5. The database has two relations, *left_fork* and *right_fork*, so that *left_fork(p,f)* means that philosopher p has fork f to its left, and similarly for *right_fork(p,f)*. Since philosophers and forks are both implemented as processes in this program, a philosopher is represented by a term of the form $process(philo, id1, id2)$ where $philo$ is a constant indicating the type of process and $id1$ and $id2$ are system- and programmer-defined process identifiers. Consider, for example, the tuples: *left_fork(process(philo,199,0), process(fork,187,0))*, and *right_fork(process(philo,199,0), process(fork,193,1))*. The first tuple says that the left fork of a philosopher process with process-id 199 and programmer-defined-id 0 is a fork process with identifiers 187 and 0. Figure 1(a) shows the graph representation of these tuples. The function symbol in the term (*process* in this case) is used to select the icon that represents the node. The



(a) Tuples (b) Hygraphs

Fig. 1. Visualizing tuples and hygraphs.

data model we use is more general than graphs. Hygraphs, are a hybrid between Harel's higraphs [9] and directed hypergraphs. 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 1(b) shows two blobs where the philo and fork processes (*blob nodes*) contain their respective communication events *ev(...)*. A node may have multiple blobs associated with it, and blobs may be arbitrarily nested. Extending the representation to hygraphs allows varying levels of abstraction in the display of hierarchical data and provides a flexible mechanism for clustering information. The **Hy$^+$** system has browsers with extensive facilities for editing hygraphs (e.g., copy, cut, and paste; selective collapsing and exploding of blobs; panning and zooming; executing several layout algorithms; moving hygraphs to and from files;

editing node and edge labels; etc.).

The visual queries supported by the **Hy$^+$** system are expressions of the GraphLog query language [4]. GraphLog queries are graph patterns whose nodes are 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 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)[5]. It will be clear from the examples discussed in the paper that debugging requires a language with the ability to express transitive closures.

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. We have experimented with several different back-end query processors: Prolog [8], the LDL deductive database system from MCC [13], and CORAL [14], the deductive database system from Wisconsin. The current implementation uses CORAL. The architecture of the **Hy$^+$** system is shown in Figure 2.

An examination of the survey [12] reveals that most distributed program analysis systems do not provide all the facilities provided by the **Hy$^+$** system in one single system.

Hy$^+$ combines to a certain extent the methods proposed in [15], [18], and [1]. The system described in [15] uses the query language TQuel [16]. [18] does not provide a querying capability, and the abstraction, filtering and visualization capabilities are also limited. The EBBA system [1] provides higher-level event abstraction mechanism, but ignores precedence information and lacks visualization capability provided by **Hy$^+$**. Furthermore, the visual and graphical nature of GraphLog queries allow the user to express intuitively appealing queries.

3 Methods of Program Trace Analysis

There are two ways of analyzing distributed program traces: *static* and *dynamic*. In static analysis, the execution traces of a program are collected and analyzed off-line on the whole trace database. Dynamic analysis can be performed in two ways: 1) *on-line*, through real-time feed of the event traces as the program executes; 2) postmortem *animation* of stored event traces, that is, the dynamic behaviour of the program is simulated by replaying the events in the collected traces. In the second method the stored trace data is fed to the database line by line (or in chunks of lines as specified by the user), thus giving the illusion that the trace data from the execution monitors are being fed to the database

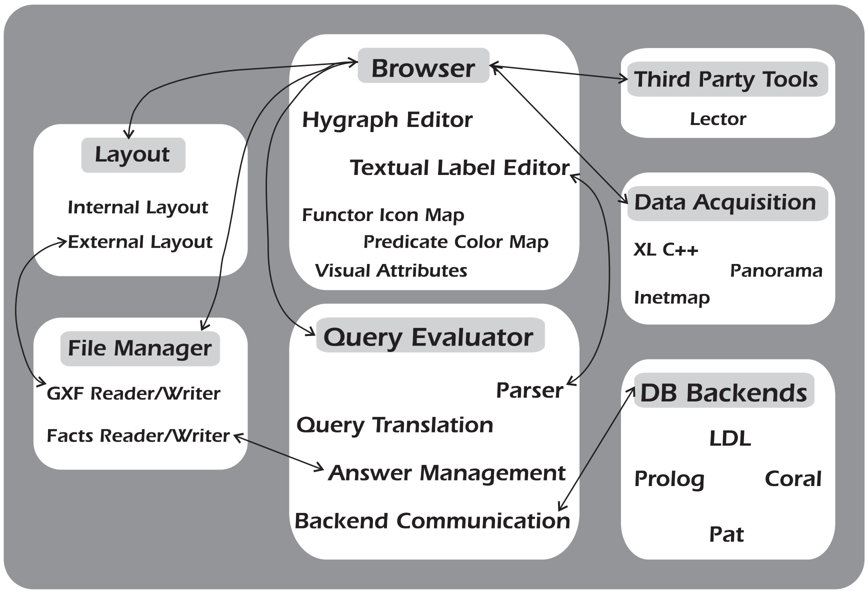


Fig. 2. Overview of the **Hy**$^+$ architecture.

in real-time. Real-time analysis, as opposed to simulating dynamic behaviour, is problematic for various reasons.

3.1 Causality Graph

In our model of distributed computation interprocess communication (IPC) is achieved only by synchronous or asynchronous message passing. The set of events (*send*, *receive*, *return* or *reply* and *implied receive* of the returned message) 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*. The partial order is defined as follows.

Let $e_{i,k} \in S$ be the *kth* event in process i. Then we define $e_{i,k} \rightarrow e_{j,l}$, 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.

From this definition it follows that the set of events $e_{i,k} \in S$ forms a directed acyclic graph (DAG), called the *causality graph*. **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 Process of Trace Analysis with Hy$^+$

We start the analysis 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* [11] monitor. The steps of debugging with **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 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.

In dynamic analysis, each time the database is updated, the specified queries are evaluated. Details are discussed in Section 6.

4.1 Description of Traces

In this subsection we describe the trace formats we obtained from running *Hermes* [17] programs. Hermes is an experimental programming language for distributed computing developed at IBM Research. 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( source_process_name, destination_process_name,
            source_process_id, destination_process_id,
            source_event_serial_no, destination_event_serial_no,
            service_name, programmer_defined_id, message)
```

where comm_event stands for one of call, send, receive, return 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_impl(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_impl by process 205, of type philo, from a corresponding return 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_impl relates to a getFork service requested by the caller. The 2 is a programmer defined identifier. 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.

4.2 Transforming the trace database

The trace as described above is the raw data that $\mathbf{Hy}^+$ takes as input. The graphical representation of the initial trace database is shown in Figure 3. The

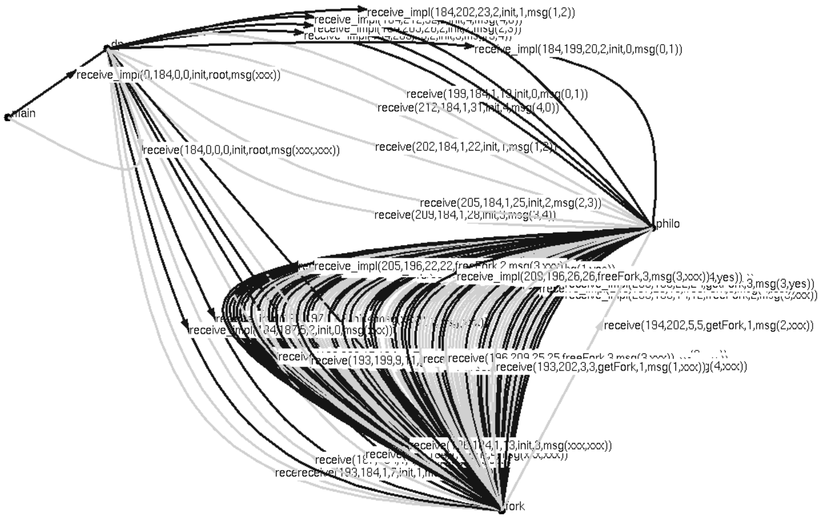


Fig. 3. Unusable $\mathbf{Hy}^+$ representation of the initial trace database.

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 issue a query containing a number of defineGraphLog boxes as shown in Figure 6, to define logical concepts of interest to the programmer. Before describing these queries, let us go over the process of query formulation in GraphLog. A GraphLog query is a set of boxes of two kinds: *define* and *show*. Both types of boxes contain a graphical pattern in which certain objects, shown thicker in the pictures, are *distinguished*. The meaning of a define box is: for each instance of the non-distinguished part of the pattern found in the database, define a corresponding instance of the distinguished object. The meaning of a show box is: for each instance of the pattern found in the database, add to the answer all the distinguished objects matched.

For example, box define1 in Figure 4 has a distinguished instance edge. The meaning is: for each process P that has received an initialization message from the root, create a process node and connect P to it with a new instance edge. We can think of this as a Horn clause or rule where the "if" part corresponds to the receive edge specialized to have init as a constant argument, and the "then" part to the distinguished instance edge. An example of a show box is shown in box show1 in Figure 4. The distinguished instance edge is to be shown whenever the first argument of its process endpoint is fork. Intuitively, this is simply a request to show all the instances of fork processes, and thus equivalent to a relational selection and projection. We will show more sophisticated uses of show boxes later on.

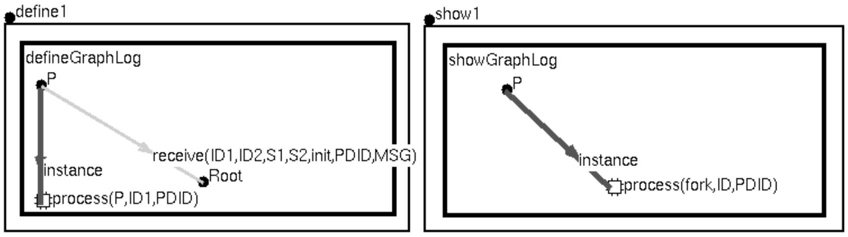


Fig. 4. **Hy**$^{+}$ queries in define and show mode.

We now describe the queries we will use to form the *causality graph* from the raw trace data. The define1 query in Figure 4 just described is the first query of the preprocessing queries. The boxes labelled define2 and define3 in Figure 6 define *event* nodes (ev) and relate each process instance to all the events within that process instance by means of events edges. The box labelled define4 defines a new kind of edge (IPC edge) that relates each call event within a calling process to the corresponding receive event in the callee. Finally, boxes define5 and define6 define the precedence relationship precedes between events corresponding to the partial order defined in Section 3. Similarly the *return* and *receive_impl* events are defined. Thus these preprocessing queries construct the *causality graph*.

These transformations are somewhat complicated, but the programmer does not need to specify them each time he or she starts to analyze a new program.

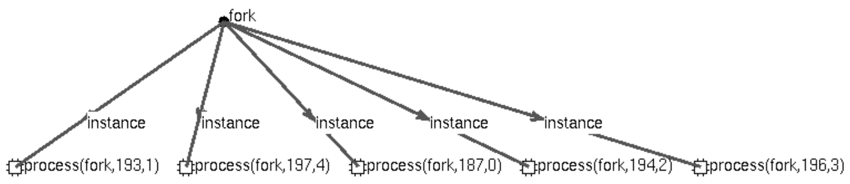


Fig. 5. Result of show query.

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

4.3 Process and Event Abstraction

Now that we have reformulated the raw data, we are in a position to do some clustering. Figure 7 shows clustering of all events within a process P into a process blob owned by P (box define1). 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 show1 in Figure 7 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 PT diagram window is the result. The process blobs can themselves be clustered together if desired, thus achieving any level of abstraction. In Figure 7 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 next section.

5 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 a while.

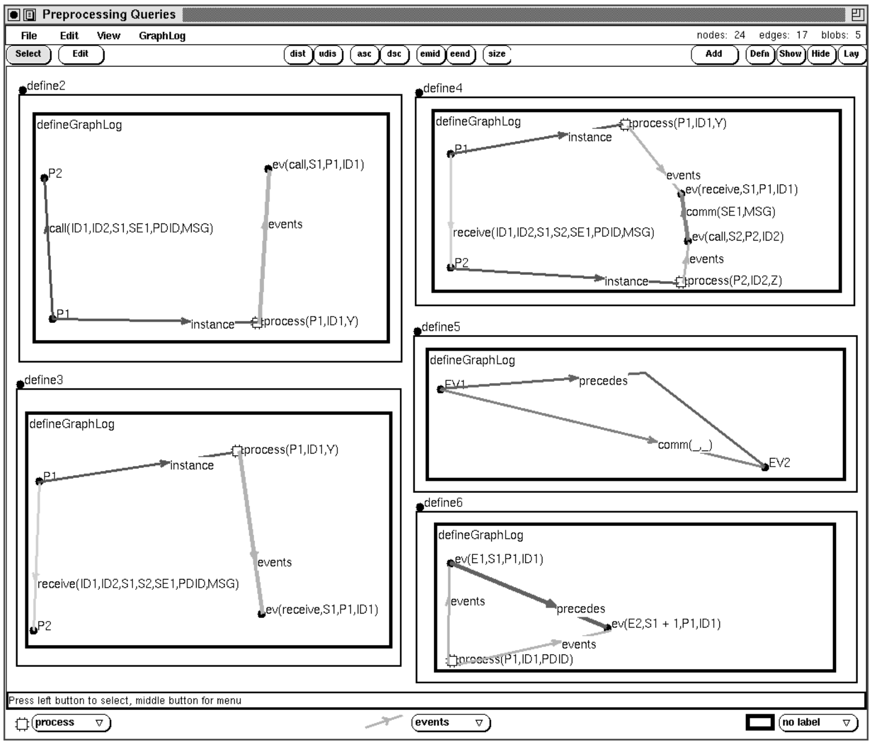


Fig. 6. Preprocess queries to form causality graph.

Let's see how a programmer might use **Hy**$^+$ to observe his program's behaviour. The programmer first starts with the simple solution in Figure 8.

The root process dp creates 5 instances of the philo and the fork processes and passes the philosopher processes their corresponding left and right forks they are allowed to use. Left fork of philo[i] is fork_id[i] and the right fork is fork_id[(i+1) mod 5]. 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 13 of the dp process above. The programmer will, in general, formulate a number of GraphLog queries to observe the behaviour of his/her program and evaluate them over the stored trace database. For example, in this particular case, programmer will formulate queries showing the left and right fork initialization pattern, fork holding and releasing patterns, deadlock pattern and the ordering of eating events by the 5 philosophers. The evaluation of the left and right fork initialization pattern query in Figure 9 reveals the bug as a hole in the pattern as shown in Figure 10(a). The query in Figure 9 operates on the message contents (msg(L,R)) of the initialization message received by the philosopher processes from the root process. For example, a left fork edge is created between philo(I) and fork(K), if

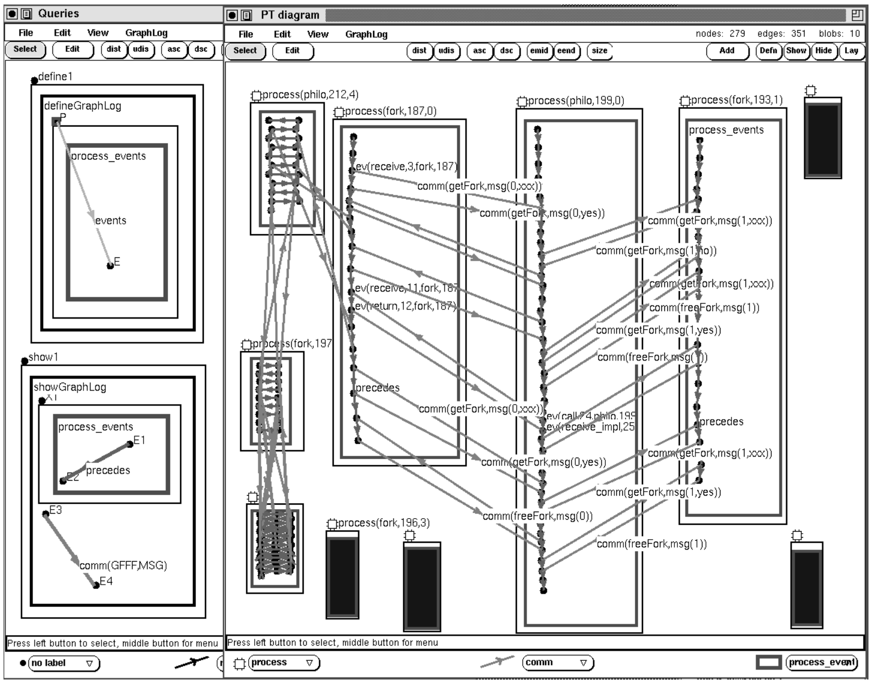


Fig. 7. Event trace hygraphs.

$K = L$, where L is the left fork id in msg(L,R) received by philo(I) during initialization. Correcting the bug produces instead the Figure 10(b) with the correct initialization pattern.

The program that results from correcting the first bug may cause a deadlock. This type of bug is called a *timing bug* and results from the nondeterministic nature of distributed programs running on a distributed network of processors. We will now see how the deadlock and its cause is observed. The query define1 in Figure 11 defines a holds edge between a philosopher and a fork process, if a getFork request is granted. If the request fails, the request_fails edge is created. The crossed out edge labelled ^precedes (representing negation) states that no return event followed the receive event in the fork. The query define3 in Figure 11 defines the *wait-for* cycle indicating deadlock, the result of which is shown in Figure 12(a). By showing the holds events in Figure 12(b) the real cause of the deadlock is observed. It shows that all the philosophers were able to simultaneously hold the left fork, thus waiting forever for the right fork.

The programmer now writes a third version by introducing a "fair policy" in the hope that it might avert the deadlock. In this solution, line 8 of the fork process is replaced with return("yes") and line 10 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 philo process. A

```
process dp:
    --create 5 fork and philo processes
    while i < 5 do
        fork_id[i] := create("fork")
        create("philo")
        i := i + 1
    endwhile
    --initialize the philo processes
    --by passing them appropriate
    --left and right fork ids
    while i < 5 do
        left_fork := fork_id[i];
        right_fork := fork_id[i+1];
        call philo[i].init(left_fork, right_fork)
    endwhile

process fork:
    get_fork:
        receive()
        if fork available
            available := FALSE
            --give the fork to the
            --requesting philo
            return
        else
            --Block (do not return)
    end get_fork
    free_fork:
        receive()
        available := YES
        return
    end free_fork
process philo:
    init:
        receive(left_fork, right_fork)
    end init
    call fork[left_fork].get_fork()
    call fork[right_fork].get_fork()
    eat
    call fork[left_fork].free_fork()
    call fork[right_fork].free_fork()
```

Fig. 8. Dinning Philosophers Code.

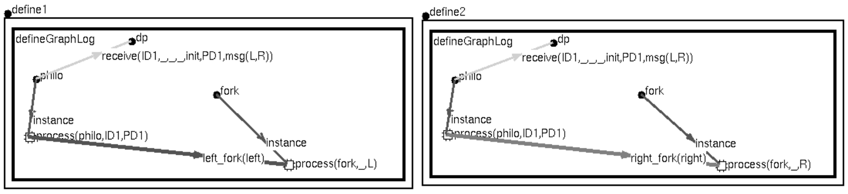
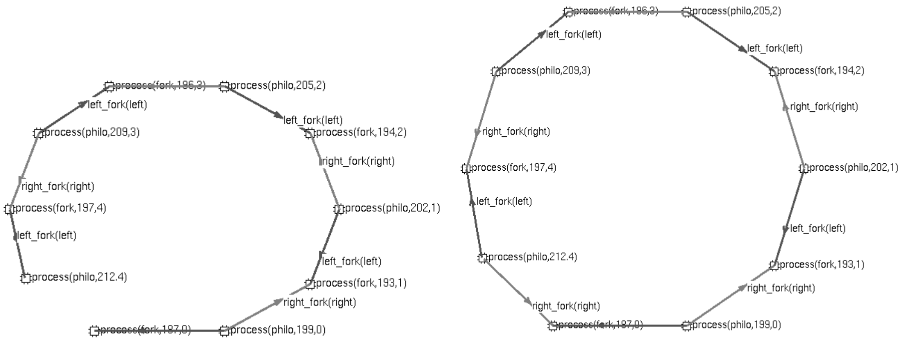


Fig. 9. Assigning Left and right forks.



(a) Incorrect initialization

(b) Correct initialization

Fig. 10. Left and right fork initialization patterns.

bug is introduced in line 5 of the fork process, namely the programmer mistakenly writes TRUE instead of FALSE. This means that the fork is always available even though another philosopher might have acquired it.

The queries showing the ordering of eating events give a hint about the bug. To formulate the ordering queries we first create abstract events (abs_ev(getFork, ...) and abs_ev(freeFork, ...)) by collapsing the send, receive, return, and receive_implied events into one single blob through the query define1 in Figure 13. We then define the precedence abs_precedes between the abstract events using the queries define2 and define3 in the same figure. The ordering of eating events is defined in the query define1 in Figure 14. The query states that if the last getFork event of a philosopher transitively precedes (abs_precedes+) the same event in another philosopher, then create a eat_precedes edge between these two philosophers. The dashed line indicates transitive closure. The result of this query is shown in Figure 15(a). By complementing eat_precedes edges we get the philosophers who eat concurrently, shown as conc_eat edges in define2 query in Figure 14. The regular expression eat_precedes | -eat_precedes states that the eat_precedes edges are to be matched in the forward or the reverse direction. The visualization of conc_eat in Figure 15(b) shows that two neighboring philosophers, in this case philosophers 2 and 3 eat concurrently, thus indicating that

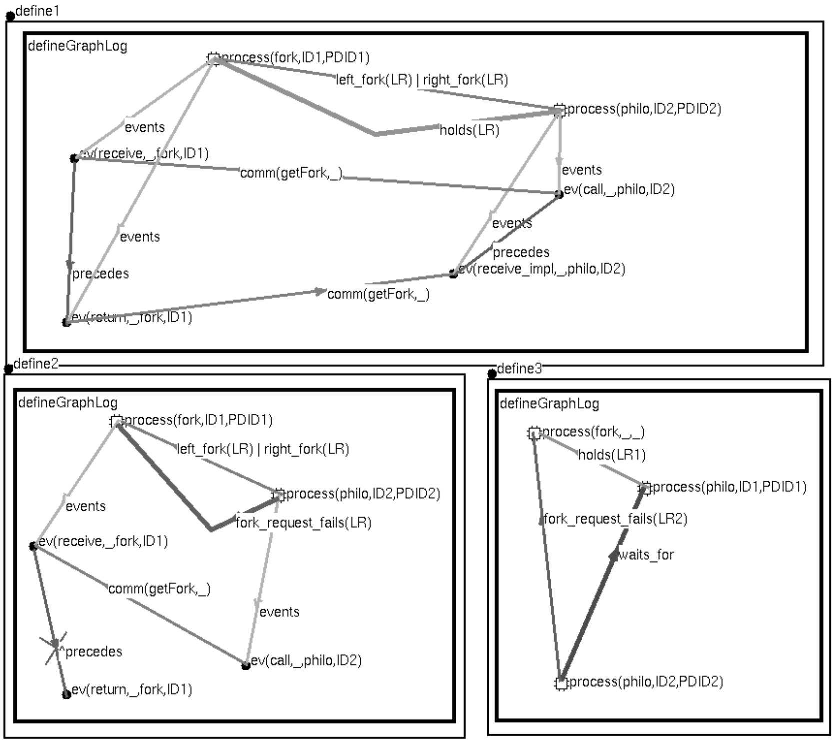
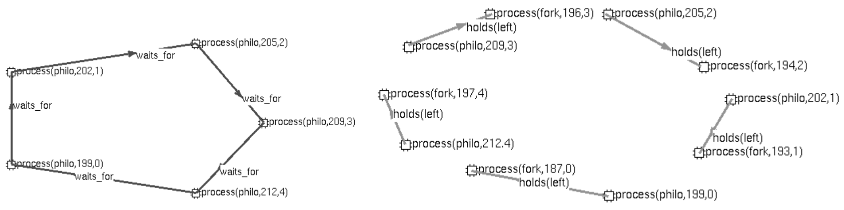


Fig. 11. Defining patterns to detect deadlock.



(a) Wait-for cycle

(b) Holding pattern

Fig. 12. Deadlock detection.

the availability of the forks has not been checked properly.

6 Dynamic Analysis

Using static analysis alone, it may be difficult to detect the real cause of a problem. For example, consider the bug in the initialization of the philosopher processes with correct left and right forks. It may be difficult to see immediately

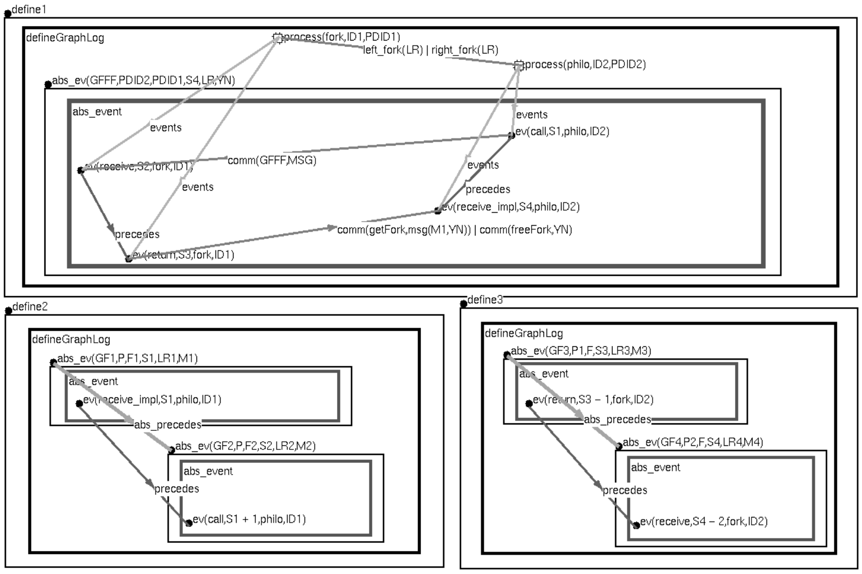


Fig. 13. Defining abstract events.

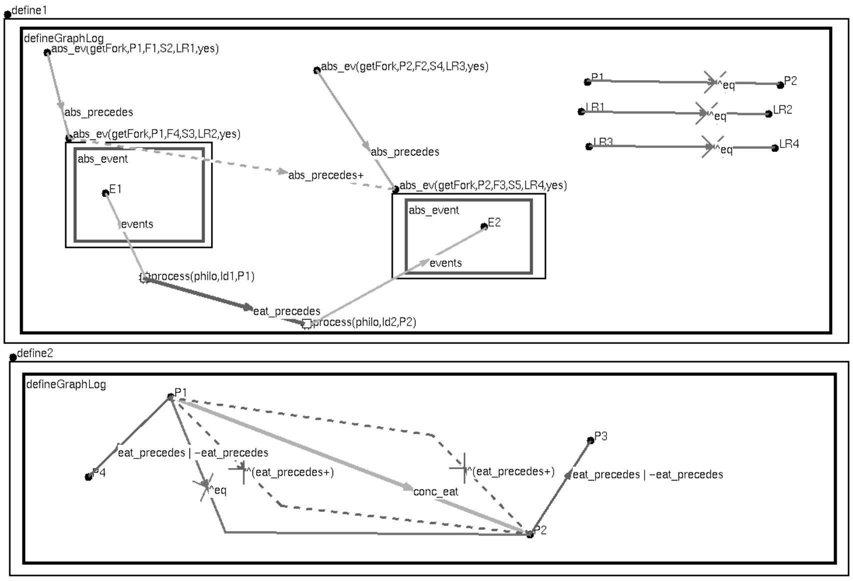


Fig. 14. Defining ordering of eating events.

(a) Precedence (b) Concurrence

Fig. 15. Precedence and Concurrence of eating events.

why the hole was formed by analyzing the whole database. The database may be very large, since the execution of the program proceeds in spite of the wrong initialization. Using dynamic analysis, the simulation can be stopped as soon as the hole is detected and the tuples that have just been fed into the database can be analyzed. The analysis will indicate that the message content during initialization of the last philosopher is msg(4, 5) instead of msg(4, 0).

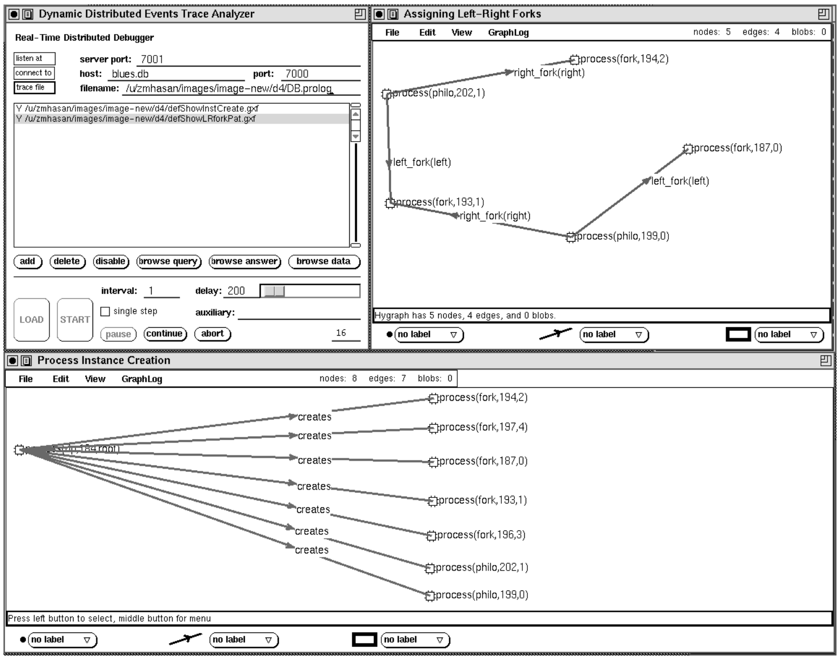


Fig. 16. Snapshot 1 of dynamic analysis.

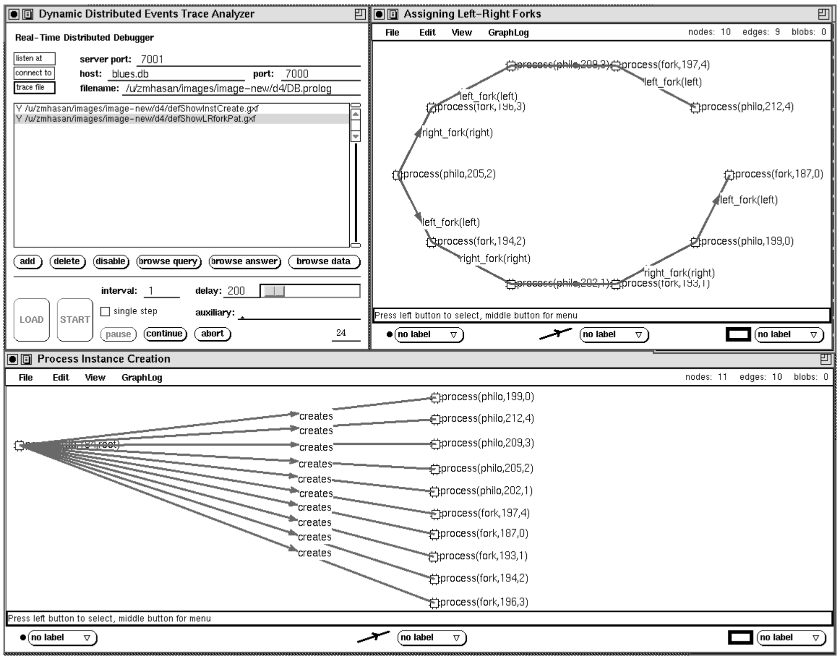


Fig. 17. Snapshot 2 of dynamic analysis.

Hy$^+$ supports a simple dynamic analysis facility, illustrated in Figure 16. The Window Dynamic Distributed Events Trace Analyzer in the figure shows the control panel for dynamic analysis. Traces are read in from a log file or from a (UNIX) socket. Trace data are fed into the database system tuple by tuple or grouped into sets at intervals specified by the programmer through the control panel. A delay can be specified to control the speed of simulation. Each time new data are fed in, a set of queries is evaluated. In Figure 16, two such queries have been specified by the programmer by giving their file names in the control panel. Snapshots of the latest results of the queries are shown in the other two windows in the figure. As the simulation progresses, at one point the user notices that even though all the process instances have been created, the left-right forks assignment pattern is not a cycle as it should have been. Figure 17 shows another frame where the user has detected the hole in the left and right forks assignment pattern and stopped the simulation for close observation of the trace.

7 Conclusion and Future Work

We have described how the **Hy**$^+$ data visualization system can be used for analysis of distributed event traces, in particular, for observing the behaviour of parallel and distributed programs. We have shown how a a combination of deductive database technology with novel visualization techniques can provide a useful tool for understanding distributed and parallel programs, by providing abstraction and filtering as well as visualization facilities.

In the current version of dynamic analysis, the queries are evaluated in a naive fashion, namely, each time the database is updated with a tuple (or tuples) all the selected queries are evaluated and results are recomputed from scratch. Ideally, we would like to have active database features, that is, firing only affected queries. The queries would be evaluated incrementally based on timestamps of events, instead of recomputing all results from scratch.

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 a *temporal event and interval specification language* (TEISL) incorporating operators similar to interval operators in [16] and a visual equivalent of the proposed operators [10]. TEISL will be incorporated seamlessly into GraphLog. The proposed extension will be a vehicle for visual specification of temporal events in an active database and temporal queries in a temporal database. Work is in progress to add active database features as required by the dynamic analysis. The resulting tool will also be useful for behavior observation of active databases [10].

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. 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.
3. Mariano Consens and Alberto Mendelzon. Expressing structural hypertext queries in GraphLog. In *Proceedings of the Second ACM Hypertext Conference*, pages 269–292, 1989.
4. 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.
5. 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.
6. Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *14th. Intl. Conference on Software Engineering*, pages 138–156, 1992.
7. Mariano P. Consens. Creating and Filtering Structural Data Visualizations using Hygraph Patterns. Fothcoming PhD Thesis, University of Toronto, 1993.

8. Milan Fukar. Translating GraphLog into Prolog. Technical report, Center for Advanced Studies IBM Canada Limited, October 1991.
9. David Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.
10. Masum Z. Hasan. A visual and temporal framework for behavior observation of active databases. PhD Thesis Proposal, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 1994.
11. W. Korfhage. The panorama monitoring system. Technical report, Polytechnic University, Brooklyn, 1992.
12. Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.
13. Shamim Naqvi and Tsur Shalom. *A logical language for data and knowledge bases.* Computer Science Press, New York, 1989.
14. R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proceedings of International Conference on Very Large Databases*, 1992.
15. Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.
16. R.T. Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.
17. 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.
18. 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.

This article was processed using the LaTeX macro package with LLNCS style