Debugging Distributed Programs by Visualizing and Querying Event Traces

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

consens@db.toronto.edu zmhasan@db.toronto.edu mendel@db.toronto.edu

Computer Systems Research Institute
University of Toronto
Toronto, Canada M5S 1A1

1 Introduction

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 the analysis of these data. We will argue that a parallel and distributed debugger should incorporate the following important and flexible facilities: *visualization*, *abstraction* and *filtering* and *layout*. We will describe technology developed as part of the **Hy**$^+$ visualization system and how it can be applied to parallel and distributed debugging.

Abstraction serves two purposes: it reduces the volume of information displayed to the programmer, and it helps the programmer reason about 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 [GF88], [ZR91], 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, and at too low a level. We propose instead a pattern matching paradigm; the programmer can specify normal or abnormal patterns that he 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, that 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:[Mil91]

> ...Bart Miller stated that he likes visualization but sees some problems. He gave two examples in which nice patterns emerged in pictures only when the layout of processors or processes was carefully arranged. Such visualization requires intimate knowledge of the relation among processors and processes and is problem-specific. He believes that for visualization to gain wide acceptance, it must be mostly "automagic," must allow views at different levels of abstraction, and must not require skill equal to that required for writing the target program. He claimed 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 post-mortem debugging of parallel and distributed programs. $\mathbf{Hy}^+$ emphasizes precisely those aspects mentioned above. Instead of a fixed way of visualizing program behaviour, it provides a general and powerful query language, called *GraphLog* [CM90a]. 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 gets 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 model of distributed computation we are interested in and describe in detail the process of debugging with $\mathbf{Hy}^+$. In Section 4 we use as an example several different versions of the *Dining Philosophers Problem*, showing how their behaviour 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 $\mathbf{Hy}^+$ and GraphLog

We will describe how the technology embodied in $\mathbf{Hy}^+$ could be used as the basis for the database visualization component of a debugger for message-passing concurrent programs, but first we need to give a general overview of the system.

Data in $\mathbf{Hy}^+$ is visualized as directed labelled graphs and *hygraphs*. A hygraph is a "graph" that in addition to edges also incorporates *blobs*. A blob relates a containing node with a set of contained nodes. Visually, a blob is represented as a rectangular region that associates containing nodes with the contained ones. Hygraph nodes are labelled with first-order logic ground terms, such as `ioprocess` or `event(receive,15,fork,187)`. Edges are

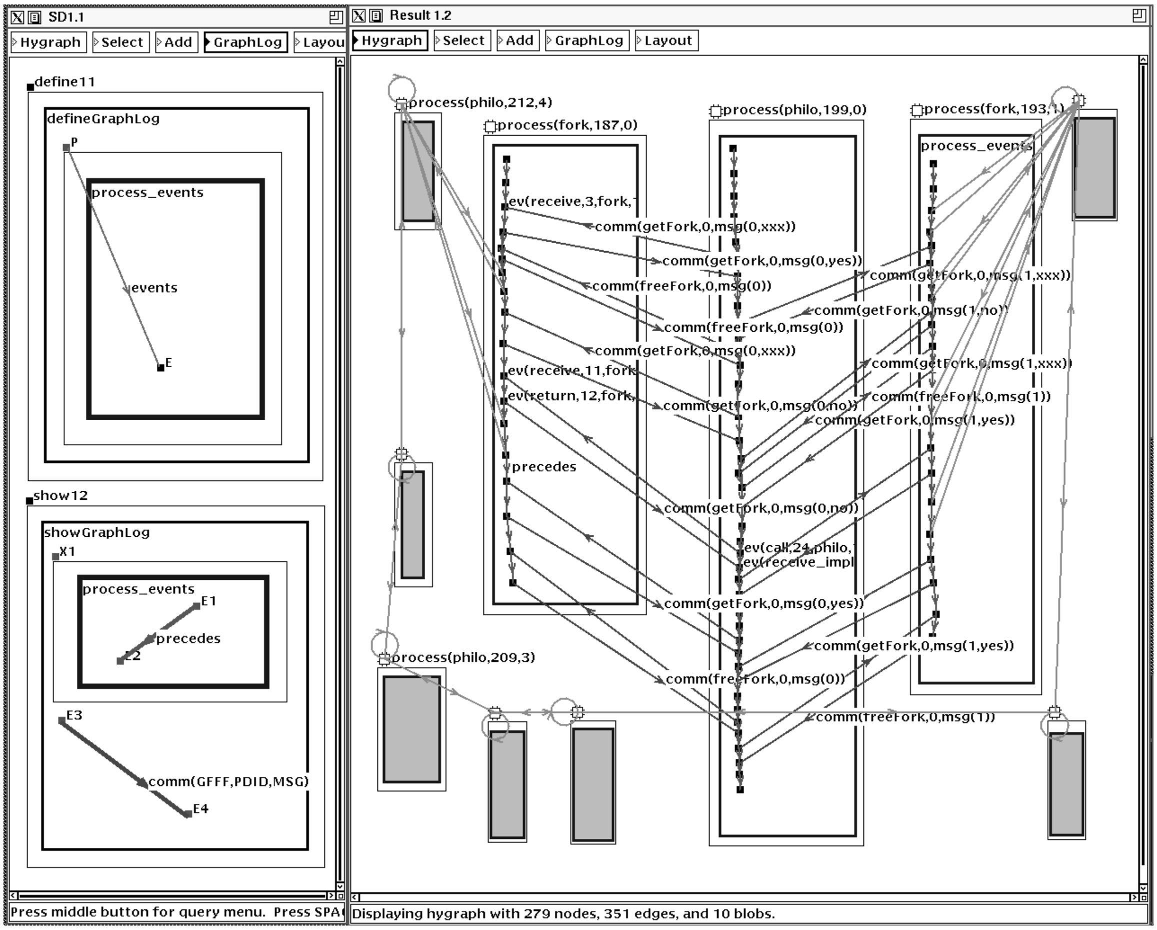

Figure 1: Event trace hygraphs.

labelled with facts such as `comm(freeFork,0,msg(1))`. Blobs are labelled with predicate names, such as `contains`. An edge labelled $r(c_1, \ldots, c_k)$ from a node labelled a to a node labelled b corresponds to the first-order fact (or relational database tuple) $r(a, b, c_1, \ldots, c_k)$.

For example, the window labelled *Result 1.2* in Figure 1 shows a particular way to visualize data from an execution of a "dining philosophers" program. The chip-like icons represent fork or philosopher processes, and are labelled with the process ID and additional information. Each process node is attached to a *process_events* blob containing nodes that represent all the events in this process's computation; events within the same process are connected by sequential precedence edges. Edges from an event in one box to an event in another box represent inter-process communication events. Seven of the ten blobs have been collapsed and their contents hidden to simplify the picture. Colours are used to distinguish the various kinds of edges and blobs.

GraphLog queries are graph patterns with nodes labelled by first-order terms, and edges labelled by *path regular expressions* on predicates. 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 arcs or blobs in the database, or extracting from the database the instances of the pattern. GraphLog can express, with no need for recursion, queries that involve computing transitive closures or similar graph traversal operations, making it more powerful than first-order languages based on the relational algebra or calculus such as *TQuel* [Sno87]. The language is also capable of expressing first order aggregate queries as well as aggregation along path traversals (e.g., shortest path queries), thus making it useful for performance debugging or measurement of parallel and distributed applications. Formal semantics of GraphLog queries, as well as a characterization of its expressive power, can be found in [CM90a, CM90b].

For example, looking at Figure 1 again, the box labelled *define11* in window *SD1.1* was used to create the *process_events* blobs. It can be interpreted as follows: whenever you find an edge *events* from a process node P to an event node E, put E into the *process_events* blob associated with P. The box labelled *show12* in Figure 1 is an example of a *filter* query: it 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. In fact, the *Result 1.2* window is the result of running these two queries on an earlier visualization (as will be shown later) that did not have blobs in it.

The **Hy**$^+$ system has browser with extensive facilities for editing hygraphs and performing other operations on hygraphs, for example, executing several layout algorithms, interactively hiding and showing blob contents, etc.

3 Debugging with Hy$^+$

In this section we give a detailed description of the process of postmortem debugging with **Hy**$^+$. We start our discussion with the model of distributed and parallel computations we are interested in; since *filtering* and *clustering* during the process of debugging may be performed in different ways, we explain how they are performed in the process of debugging with **Hy**$^+$; we then explain the debugging process: describe the format of the event traces; show how this trace is converted into the *causality graph*; finally we discuss event abstraction.

3.1 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 [Bat89], [GF88] and others. An event is a uniquely identified runtime instance of an atomic action of interest [Fid91]. 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 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.

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.* **Hy**$^+$ can be used effectively for dealing with graph structured database. For example, the precedence relationship between two events can be decided by path traversals in a graph.

3.2 Filtering and Clustering

Filtering and *clustering* are two methodologies used to reduce information for debugging. Clustering or abstraction can be viewed as one form of filtering. Filtering can be performed at software or at hardware level. We are concerned with software filtering. There are two possible ways of performing software filtering: *"on-line"* filtering and *display* filtering. In the "on-line" filtering the monitor that records traces is closely integrated with the high-level module of the debugger. In this case only the data that matches the specification is sent to the user. In the case of high level abstraction, the primitive events that match with the specification are discarded, the high-level events are formed and sent to the user as in [Sno88], [Bat89]. This approach reduces the amount the storage space used to keep data. Our approach is that of display filtering and abstraction, that is, all the primitive event traces are stored for later use, and only relevant information is displayed according to the need of the debugger through the use of facilities provided by **Hy**$^+$.

3.3 Debugging with 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 have used an instrumented version of DCE [Fou92] as well as traces produced by the IPC event collection modules from the *Panorama* [Kor92] monitor. The steps 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.

3.4 Description of Traces

As we have mentioned before, the execution trace contains the communication events, process creation and termination events occurring in the application. The format of the trace is similar to that of [CW89]. We have used Hermes [SBG+91] as our target system. We shall use Hermes terminology for describing communication events, but ideas presented in this paper are applicable to debugging any distributed or parallel application in general, given that we have traces in the format described later. The system itself is independent of any particular trace format. Any trace format according to users need can be used.

In Hermes synchronous communication, is achieved by a call, call blocks until the callee returns. A send is an asynchronous communication event. 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 call, send, receive or receive_implied. (A receive after a return is receive_implied). Other events in the trace are create and end. They have a slightly different format. The following are examples of two event traces:

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

receive(fork, philo, 196, 205, 11, 20, freeFork, 2, msg(3)).

The originator_event_sequence_no and destination_event_sequence_no are the local event counter values. These numbers are used to define the local event precedences of a process. They are also used to match sending and receive event pairs in different processes. A call and return comm_event would not have any destination_event_sequence_no, since that number is unknown at the time of call or return. service_name is used to identify a service or port being called. A process may provide more than one services. If a process calls more than one services in another process, we've to distinguish the calls. programmer_defined_id is a useful parameter to have. A programmer while debugging a program may want to identify processes by an identifier defined by her (not by system defined process id which may not be known to the programmer), for example, left or right fork of a philosopher in a dining philosophers problem.

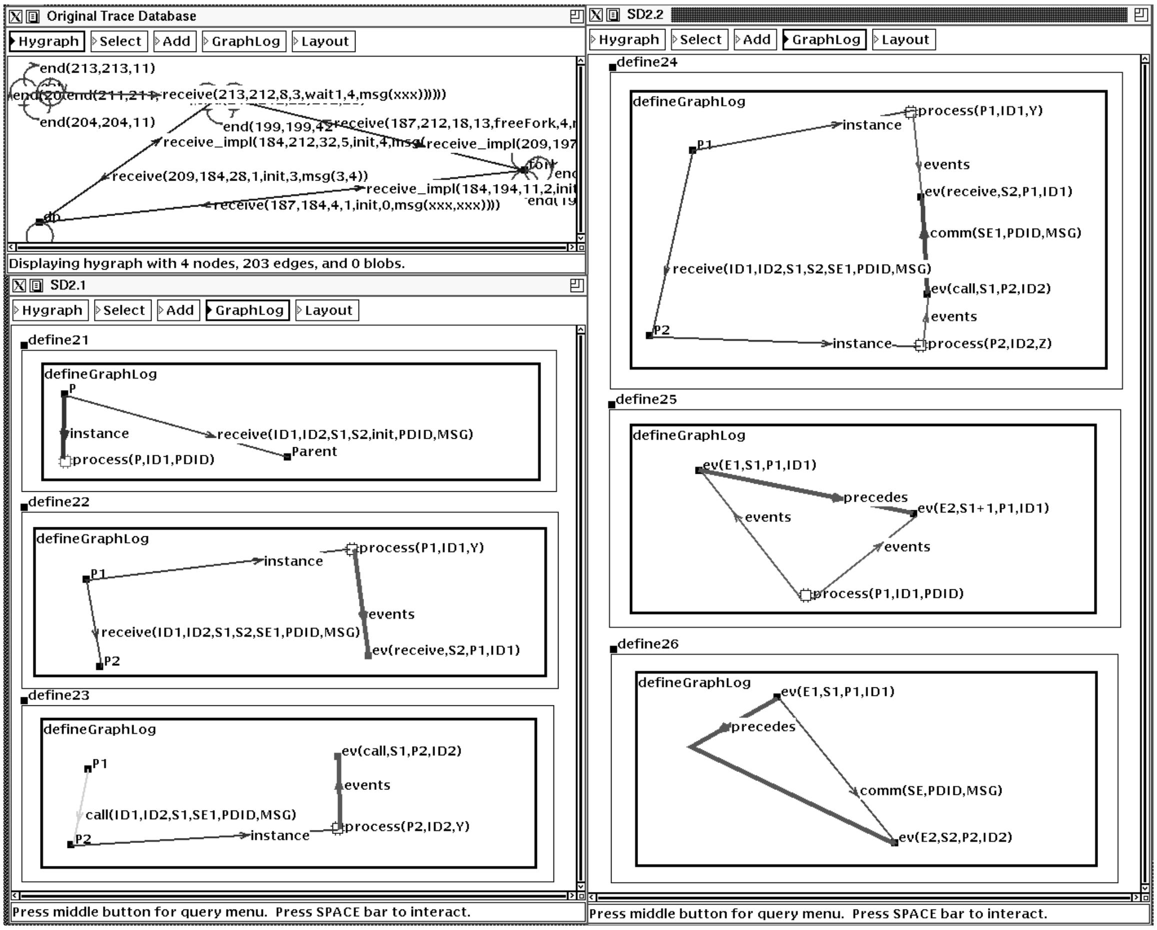

Figure 2: Transformation of the initial trace database.

3.5 Transforming the trace database

The trace constitutes the minimum database (in $\mathbf{Hy}^+$) to start with. Window *Original Trace Database* in Figure 2 shows the graphical representation of the initial trace database. A number of queries on this database transforms it into a new database suitable for use in debugging.

Query *define21* in window *SD2.1* creates the process (*functor*) nodes by carrying over the process name, process id and the programmer defined id (pdid) with it. We assume that a process is initialized with pdid during its initialization (note the init term in the receive trace). In Hermes the parent process after creating its child process(es) calls the created process(es) at the initialization port to pass it the initialization parameters. Local event nodes ev of a process are created through queries *define22 and define23* (only call and receive event nodes creation are shown). The query *define22* can be read as follows: if process P1 receives from another process P2, then create an event node ev connecting it through the edge events to the process node which is an instance of process P1.

Query *define24* defines the communication edges comm with the service name, pdid and message contents carried over the edges. The receive edge between P1 and P2 is necessary to match the call, receive pair: S1 and S2 in the receive edge have to match with S1 and S2 of the ev nodes. *define25* defines the precedence relationships between local events of a process and *define26* defines that of between communication events in different processes, thus forming the *causality graph*.

Transformation of the trace database is independent of any particular application. The above queries can be canned and applied to execution traces of any application.

Other useful queries will be shown in the next section as we walk through the example dining philosophers problem. Some of these queries are problem specific.

3.6 Process and Event Abstraction

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.

Arbitrary event abstractions may violate precedence or partial order relationship which in turn may contradict the intuition of a causality graph. Some examples of violations of irreflexivity and transitivity are shown in Figure 3.

It is easy to define the precedence relationship between primitive events. But it is not possible to define a general precedence relationship between abstract events without violating the precedence relationship. The only solution to the problem is to restrict the pattern of the abstract event specification so that the partial order relationship is maintained [Che89], [ZR91]. In this case, The debugger defining the abstract behavior, has to always aware of the fact that abstraction this way is not possible, because it violates the precedence relationship. If the system attempts to guard against the violation the programmer may not be able to specify the system behavior according to her understanding of the system. The question then arises, do we really need the partial order relationship to be satisfied when we abstract events (or processes into cluster)? We contend that since the abstraction is a user definable object, let the user also define the precedence relationship among abstract events that appeals to her intuition about the program being debugged or behavior observed. In other words, precedence relationship at a level higher than the primitive events is user definable. Of course, false precedence may result from unrestricted definition of precedence. In case of "on-line" filtering the underlying information is lost and this makes it impossible to see the real precedences of the abstract events. The facilities provided by the $\mathbf{Hy}^+$ system and the *display filtering* nature of it allows the debugger to investigate the real precedences closely by going back and forth in the original and defined databases.

4 Case Study: The Dining Philosophers Problem

In this section we describe a debugging scenario using $\mathbf{Hy}^+$. Employing a We will show how we specify As we have mentioned before that by debugging we mean both correctness and performance debugging and also observing interesting program behavior.

The *Dining Philosophers Problem*: N philosophers sit around a round table. There are N forks on the table. In order to be able to eat a philosopher has to pick up the forks that

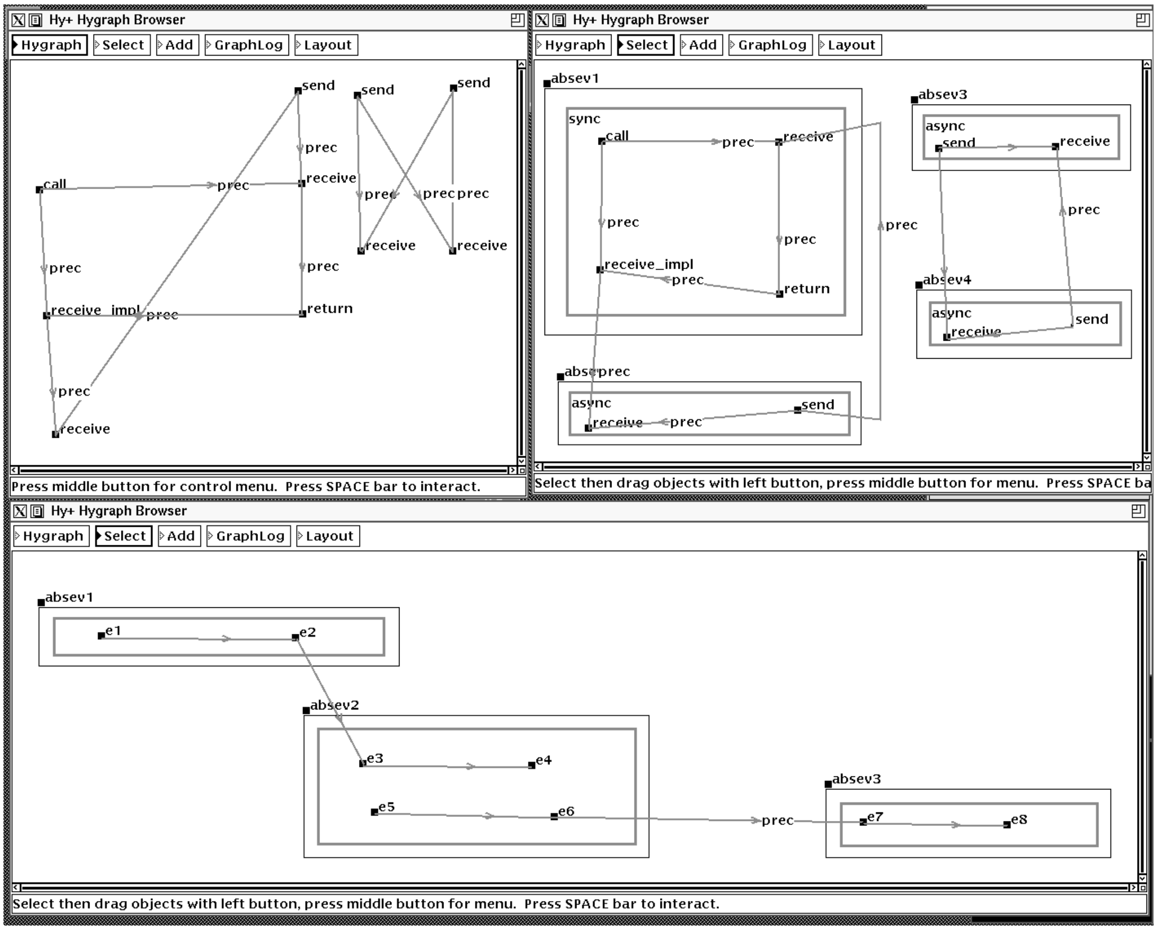

Figure 3: Abstractions that may violate precedence relationship.

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 think for a while.

Different solutions exist for avoiding deadlock and starvation. For example, both deadlok and starvation are avoided if only $N-1$ philosophers are allowed to pick the forks concurrently. But for the purpose of showing how **Hy**$^+$ system can be used in different ways to debug a distributed and parallel program, we will provide solutions to the problem as a naive programmer might while solving this problem and debugging it.

A possible simple solution to this problem is shown below.

```
process fork(i):
        get_fork:
                receive()
                if fork available
                        available := no
                        --give the fork to the requesting philo
                        return
```

```
8               else
9                       --Block (do not return)
10      free_fork:
11              receive()
12              available := yes
13              return

14  process philo(i):
15      --left fork
16      call fork.get_fork(i)
17      --right fork
18      call fork.get_fork((i+1) mod N)
19      eat
20      call fork.free_fork(i)
21      call fork.free_fork((i+1) mod N)
```

The main process (which is not shown above) creates N instances of the philo and the fork processes. The fork processes provide two services: get_fork and free_fork.

Three different versions of the dining philosophers problem were written in Hermes. In the first version the programmer wrongly specifies the right fork of a philosopher by writing (i+1) instead of (i+1) mod N in line 18 of the pseudocode shown above. This particular solution shown above constitutes the second version. This solution may cause a timing error, thus leading to a deadlock. In the final version, the programmer being aware of the timing error introduces a "fair policy" in the hope that it might avert the deadlock being happen. 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. Obvious modification is necessary in the philo process for this.

The second and the third version needs some discussion. The second version which is seemingly a correct program has a timing bug in it. In some run all the philosophers may succeed in eating, in other a deadlock may occur. Deadlock happens when all the philosophers succeeds in getting the left fork simultaneously. Note that deadlock may also happen if the programmer forgets to place the return statement in the fork process. The nature of these two deadlocks is different. In the first case, there is a *wait-for* cycle between processes, that is, one process waits for another process to release a fork. This deadlock occurs due to the timing error as opposed to the one where the programmer forgets to place the return statement.

We introduce two terms for deadlock: a *deadlock situation* and a *deadlock*. A Deadlock Situation is the one that based on the semantics of the program indicates that if appropriate action is not taken, there is a very high probability that this situation will lead to a Deadlock.

If all of the philosophers succeeds in acquiring the left forks concurrently, thus introducing a deadlock situation, they will never be able to acquire the right forks, since fork processes do not return if a fork is not available.

Programmer after becoming aware of the timing bug introduces a *fair policy*. The fair policy states that if a philo process acquires a fork and can not immediately acquire the second fork, it releases the acquired fork immediately. In this solution a philo does not block on a fork process if the fork is not available, rather in response to a get_fork request a fork replies with a "yes" or "no" based on whether fork is available or not. After releasing the fork the philo waits for a random amount of time, then tries again to acquire both the forks. This random waiting may prevent philosophers acquiring and releasing forks concurrently and thus avoiding a *livelock* situation.

In the following section we will show the actual process of debugging using $\mathbf{Hy}^+$. We have used execution event traces of the three versions of the problem.

In the process of debugging we check that:

- Left and right forks of a philosopher have been specified correctly.
- Detect the deadlock by creating a pattern, that is, the wait-for cycle.
- The fairness policy has been asserted, that is either of the following patterns is observed:
  get_fork(left), return(yes), getfork(right), return(no), free_fork(left);
  get_fork(left), return(no), get_fork(right), return(yes), free_fork(right)
- The pattern of successful eating events:
  get_fork(left); return(yes);
  get_fork(right); return(yes);
  free_fork(left);
  free_fork(right)

  get_fork and free_fork events may happen in any order, that is, they may have any *interleaved* pattern [Bat89].
- All the philosophers succeeds in eating even in the face of a *deadlock situation.* More than one *deadlock situation* may exist in the execution trace.
- There is no contradiction in eating, that is, two neighboring philosophers do not eat concurrently. This may happen if the availability of the fork is not checked.

Abstract events are introduced as necessary. Process clustering or abstraction also can be introduced as required.

4.1 Debugging The Dining Philosophers Problem

In this section we describe the process of debugging the three versions of the example problem. We describe the $\mathbf{Hy}^+$ queries and the results. Queries are executed on the transformed trace database (as shown in the previous section). We also show portions of appropriate Hermes codes. The program consists of three processes: the main process, the philo process and the fork process. The main process creates five instances (processes) of the philo and the fork processes.

4.1.1 Version I

The bug in this version is located in the main process (of Hermes code) shown below, where instead of specifying `j <- (n + 1) mod 5`, `j <- (n + 1)` has been specified for the right fork.

```
i := 0;
-- create the fork processes and save the ports of the get_fork
-- and free_fork services in a table so that the correct left and
-- right fork ports could be passed to appropriate philo processes
while i < 5 repeat
      forkCreate := create of process fork;
      call forkCreate("init", i, forkGet, forkFree, terminal);
      new forkServRec;
      forkServRec.getService <- forkGet;
      forkServRec.freeService <- forkFree;
      insert forkServRec into forkServTable;
      i := i + 1;
end while;

-- create the philosophers and pass them the outports of
-- get and free services of the fork processes
n := 0;
while n < 5 repeat
      philoCreate := create of process philo;
      new forkServRecl;
      new forkServRecr;
      --j <- (n + 1) mod 5;
      j <- (n + 1);
      forkServRecl := forkServTable[n];
      forkServRecr := forkServTable[j];
      leftForkGet := forkServRecl.getService;
      leftForkFree := forkServRecl.freeService;
      rightForkGet := forkServRecr.getService;
      rightForkFree := forkServRecr.freeService;
      k := n;
      m := n;
      call philoCreate("init", m, k, j, leftForkGet, leftForkFree,
                       rightForkGet, rightForkFree, terminal);
      n := n + 1;
end while;
```

The execution of this version revealed that only four philosophers have succeeded in eating, but we have five philosophers. Examination of the code may or may not reveal the bug. Obviously the first step in debugging this particular problem will be to check whether the philosophers have been informed about the correct forks they are allowed to access, that is, their left and right forks. Queries *define21* and *define22* in window *SD4.1* in Figure 4 defines the left and right fork of a philo process. The result in window *Result 4.2* of the filter query *show43* in window *SD4.1* shows a hole in the left-right fork cycle. The visualization shows that the last philosopher process (philo(4)) has not been initialized properly with the forks it is allowed to access. Knowing where in the program this is done, the bug can be

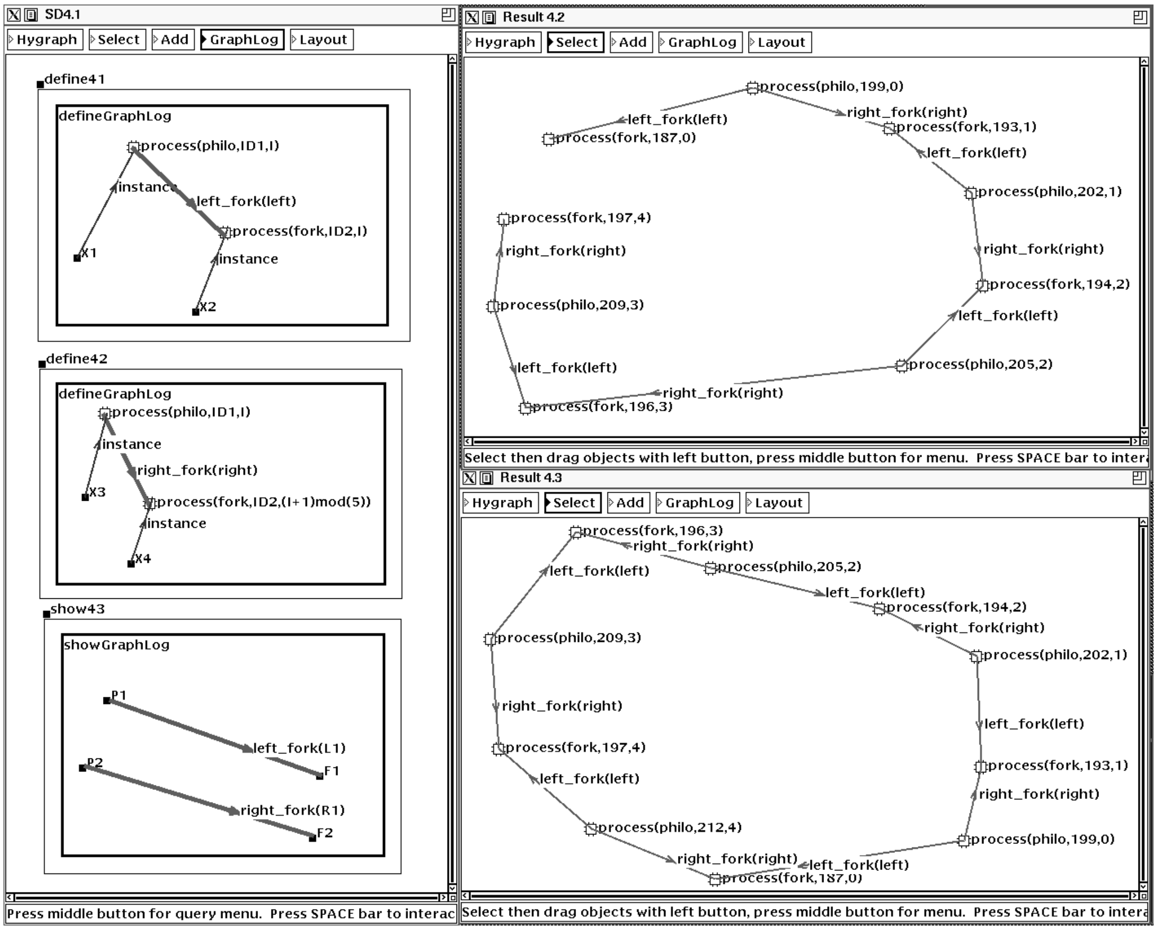

Figure 4: Left-right fork cycle.

detected.

After the bug was fixed and the program was executed again, we detect the correct left and right fork assignment pattern in *Result 4.3*. This is an example of how we create new patterns from the trace to observe the behavior of the program.

4.1.2 Version II

A portion of the Hermes code for the philo and fork processes of this version is shown below.

```
-- philo process - version II
while (eaten = "no") repeat
    -- call the left fork to get fork
    call lfg(gservice, phid, lfid);
    -- call the right fork to get fork
    call rfg(gservice, phid, rfid);
    call eatCall("eat", phid);
    --free both the forks
```

```
        call lff(fservice, phid, lfid);
        call rff(fservice, phid, rfid);
        eaten := "yes";
end while;

-- fork process - version II
while ('true') repeat
        select
                event get_fork
                        receive get_fork_msg from get_fork;
                        if available = "yes" then
                                available := "no";
                                return get_fork_msg;
                        end if;
                        -- else block

                event free_fork
                        receive free_fork_msg from free_fork;
                        available := "yes";
                        return free_fork_msg;
                otherwise
                        end select;
end while;
```

The execution of this version in some run yielded correct result, that is, all philosophers succeeded in eating and in some run this version simply halted without producing any result. We hypothesize that in the last cases deadlock has occurred. To detect whether there is a *wait-for* cycle in the trace we proceed in the following manner.

To reason about the semantics of the program at a higher level we introduce abstract event through the use of the abstraction capabilities provided by $\mathbf{Hy}^+$. The abstract event of interest is the get_fork and free_fork events. These two events consist of four primitive events: call, receive, return and receive_impl. Query *define51* in window *SD5.1* in Figure 5 creates a *blob*: a new blob relation abs_event is defined that relates the newly created abstract event node abs_ev with the four primitive event nodes. Node abs_ev is defined as a *functor* with six arguments: the name of the get_fork or free_fork event (GFFF), pdid of the philo process (PDID1) that made the request and the pdid of the fork process (PDID2), sequence number of the last primitive event in the blob (S4), information about to which of the left or right forks the request was made (LR) and what message has been returned (YN). In the case of the third version of the program, the last field of the get_fork event will contain either yes or no. We assume that the programmer knows the message contents that have been kept in the trace. Message contents are defined as a functor (msg) with appropriate fields. Note that in one single query both the get_fork and free_fork high level abstract events are defined.

Query *define52* in window *SD5.1* defines the abstract precedence abs_precedes between abstract get_fork or free_fork events. This query states that an abstract event abs_ev1 immediately precedes (abs_precedes) another abstract event abs_ev2 if the last event (receive_impl)

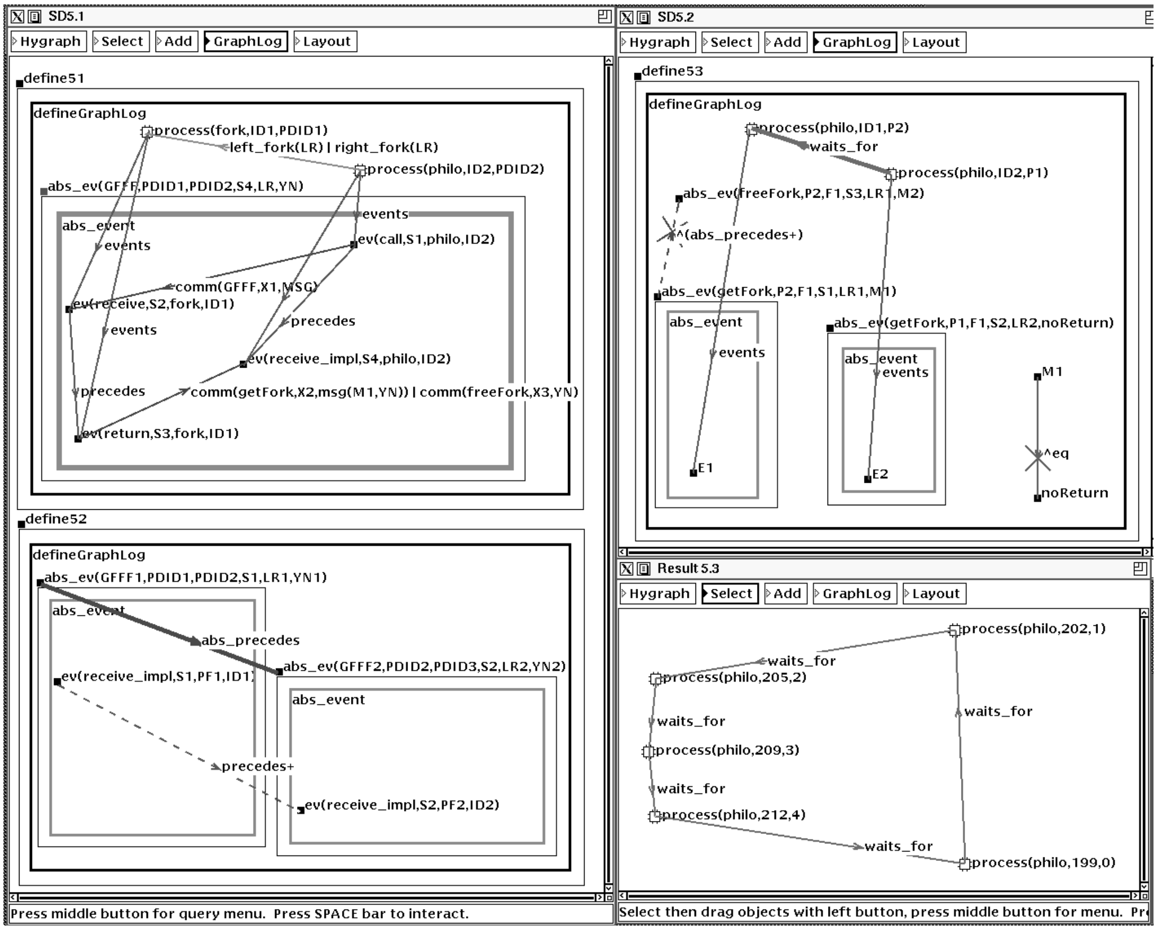

Figure 5: Detecting deadlock.

of abs_ev1 transitively precedes the last event of abs_ev2. Note that, the precedence between the abstract events is defined by the user based on her intuition of the program.

We now define the deadlock cycle. If a process P1 makes a request for a fork F1, but the request was not returned, since another process P2 succeeded in getting F1 and P2 never returned F1, then process P1 waits_for process P2 for the fork F1. If as a result of this query a waits_for cycle is created, then that indicates a deadlock. Query *define53* in window *SD5.2* defines the waits_for relation, the result (deadlock cycle) is shown in Result 5.3. That F1 was never returned is specified by the edge ^(abs_precedes+) connecting the get_fork and free_fork (which never happened) abstract events. The negation (^) is shown visually by crossing out the edge. The plus sign is the transitive closure literal (shown dashed to emphasize that the edge in the query graph matches a path in the input graph). We took transitive closure of the abs_precedes, because a free_fork event may not immediately follow a get_fork. A note about the noReturn term: If to a call, receive pair no return and receive_impl pair is found (or matched), then in the trace we introduce a dummy pair of these events and tag the message in both the traces with noReturn. Given a large and complex program with huge trace data, creating pattern like this and observing it visually definitely helps. It may not be possible to detect such pattern just looking at the causality graph.

4.1.3 Version III

We will now operate on the trace of the third version of the program and observe and assert interesting program behavior.

A portion of the Hermes code for the philo and fork processes of this version is shown below.

```
-- philo process - version III
while (eaten = "no") repeat
      -- call the left fork to get fork
      if(lavailable = "no") then
            call lfg(gservice, phid, lfid, lavailable);
            if lavailable = "yes" then
                  gotfork <- gotfork + 1;
            end if;
      end if;

      -- call the right fork to get fork
      if(ravailable = "no") then
            call rfg(gservice, phid, rfid, ravailable);
            if ravailable = "yes" then
                  gotfork <- gotfork + 1;
            end if;
      end if;

      -- got both the forks? eat then
      if gotfork = 2 then
            call eatCall("eat", phid);
            --free both the forks
            call lff(fservice, phid, lfid);
            call rff(fservice, phid, rfid);
            eaten := "yes";
      else
            if gotfork = 1 then
                  -- if only the left fork was acquired, free it
                  if lavailable = "yes" then
                        call lff(fservice, phid, lfid);
                  -- if only the right fork was acquired, free it
                  else
                        call rff(fservice, phid, rfid);
                  end if;
            end if;
      end if;
      gotfork := 0;
      lavailable := "no";
      ravailable := "no";
```

```
end while;

-- fork process - version III
while ('true') repeat
      select
            event get_fork
                  receive get_fork_msg from get_fork;
                  if available = "yes" then
                        get_fork_msg.available := "yes";
                        available := "no";
                  else
                        get_fork_msg.available := "no";
                  end if;
                  -- return the message with "yes" or "no" answer
                  return get_fork_msg;

            event free_fork
                  receive free_fork_msg from free_fork;
                  available := "yes";
                  return free_fork_msg;

            otherwise
                  end select;
end while;
```

First we show how the events in a process can be clustered into a single process blob. This has been discussed and shown in Figure 1 in Section 2. Processes in turn may be clustered into hierarchical process clusters.

The contents of a process blob not of interest, is hidden (shaded boxes). The blobs whose contents are shown can also be filtered out through a filter query and shown in a separate window. The figure in *Result 1.2* in Figure 1 serves a number of purposes: the communication pattern can be observed visually, for example the communication pattern in this figure matches with that of in *Result 4.3* in Figure 4, indicating a correct communication pattern. That the fair policy is followed can also be observed.

Query *show61* in window *SD6.1* in Figure 6 is used to observe (assert) how the successful eating events (should) take place. This query is capable of matching against any *interleaved* patterns. The query is based on the successful eating events pattern described in the previous section. In this query the successful eating event is defined as a pattern of two successful get_fork events happening in strict sequence which is followed by two successive free_fork events. The result is shown in *Result 6.2* in Figure 6. This query again shows how the **Hy**$^{+}$ system allows us to reason about the program at a very high level.

We have already seen that all the philosophers have successfully eaten. We now assert that this has happened even in the face of a *deadlock situation*, due to the introduction of the

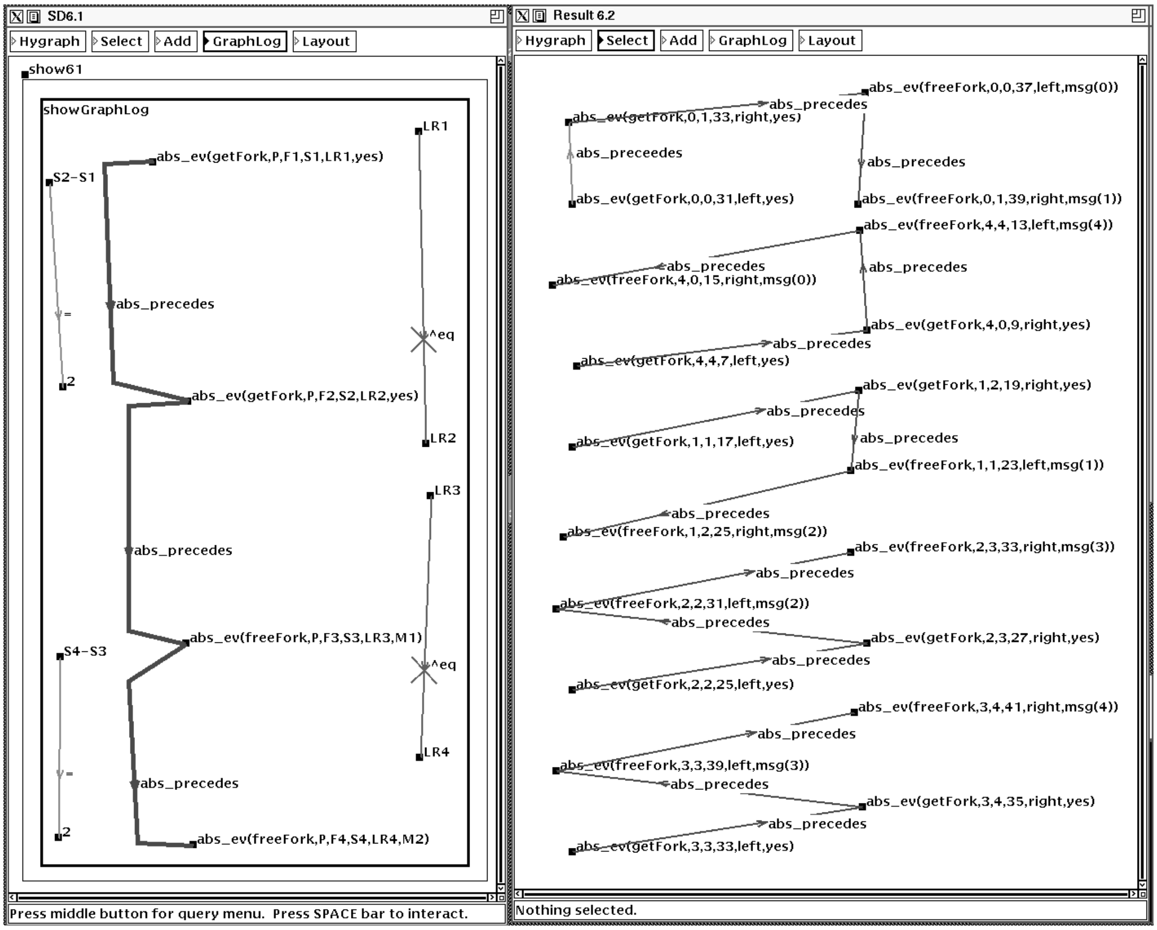

Figure 6: Successful eating events.

fairness policy. If we can find in the execution trace instances of five (number of philosophers and forks) successful get_fork(left) or get_fork(right) events happening concurrently, then that indicates a deadlock situation in certain point in time in the execution of the program. To check this we first find out the successful get_fork events that are mutually concurrent. Query *define71* in window *SD7.1* in Figure 7 performs this. This query can be described as follows: if two distinct get_fork abstract events does not transitively precede each other, then create a conc_events edge between them. Query *define72* defines the *degree* of each such node (nodes connected by a conc_events edge). The aggregation facility of the **Hy**$^+$ system is used for this purpose. This query counts the number of unique processes P2 (COUNT(P2)) with whose get_fork events a get_fork event in P1 is concurrent. Query *define73* clusters the get_fork events whose count_conc defined in the previous query equals four, into a blob defined by conc_get and conc_get_ev. This query tries to find a *clique* of size four. The result is shown in *Result 7.2*. It shows that at the beginning of the execution there were two instances of deadlock situations. This result also shows the way the dining philophers problem was solved in the third version: the conc_get_ev(right) could never happen, if the request for the right fork is not made after the left fork request has failed. Deadlock situation can also be detected by counting the number of events in the conc_get_ev blobs as shown in *define74* in

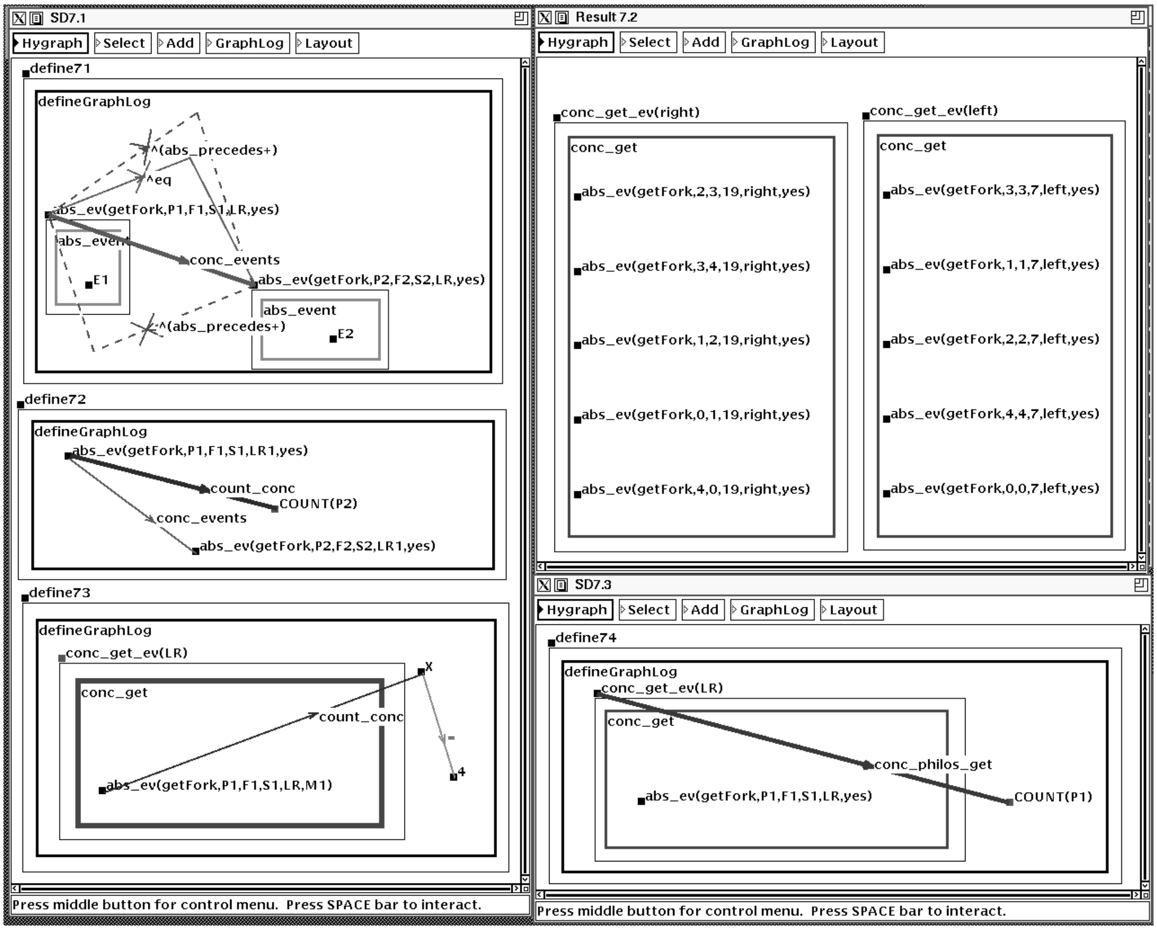

Figure 7: Observing a deadlock situation.

window *SD7.3* and then checking that count equals to five (number of philosophers).

Queries in Figure 8 define the order of eating events by the philosophers. As we know that a successful eating event happens when a philosopher succeeds in getting both the forks immediately one after the another. We define the order of eating events based on these patterns. Query *define8.1* in window *SD8.1* states that a philo process eat_precedes another philo process, if the second successful get_fork event of the first philosopher abs_precedes transitively the second get_fork event of another philosopher. The result of this query is shown in *Result 8.2*. It shows that there is no contradiction in the order of eating events: no neighboring philosophers eat concurrently. The result also shows that in the execution of the program philosopher 1 preceded in eating philosopher 3. Query *define82* in window *SD8.1* finds out the mutually concurrent eating events. If this query is run on the result in *Result 8.2*, we can observe the mutually concurrent eating events as shown in window *Result 8.3*.

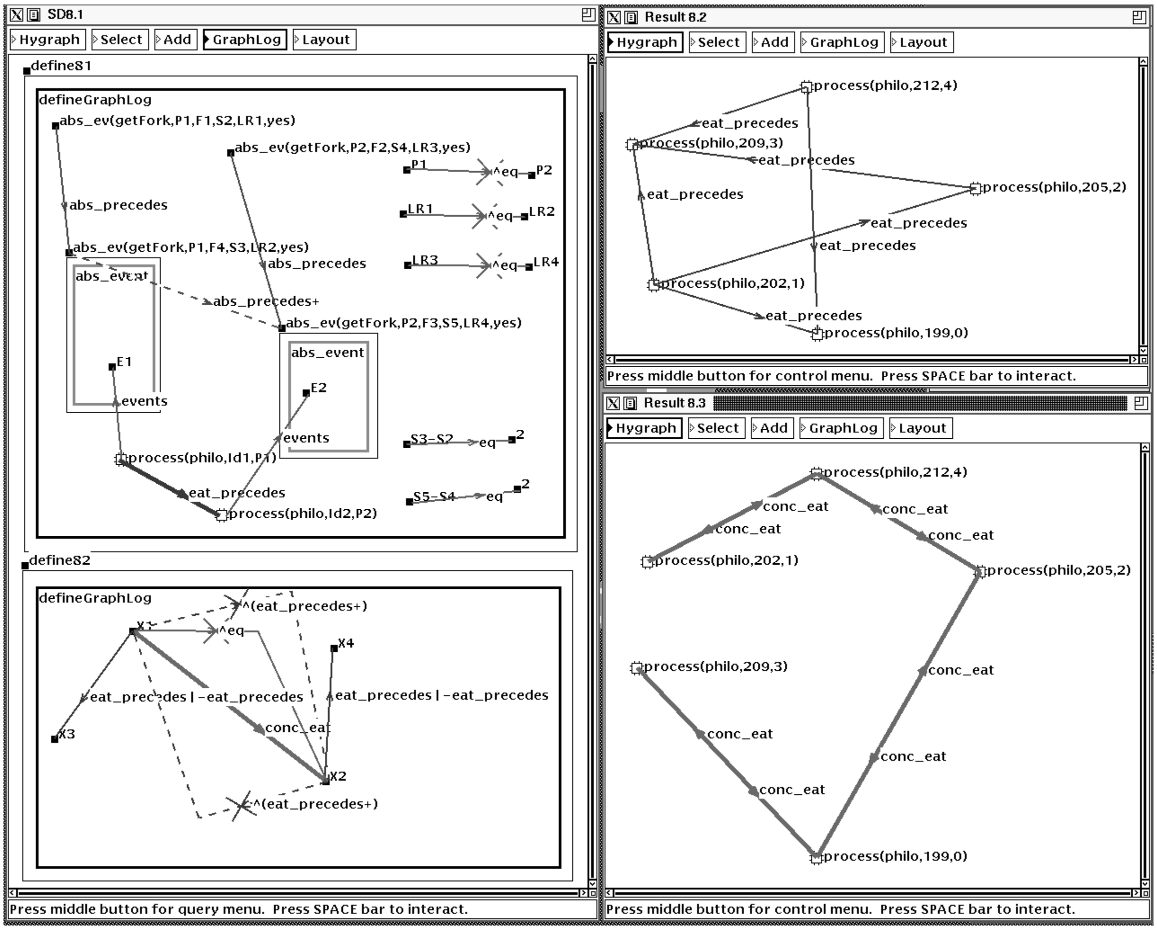

Figure 8: Precedence of eating events.

5 Related Work

An examination of the survey [MH89] reveals that most debugging systems do not provide all the items discussed in subsection 3.2 into one single debugger.

Hy$^+$ debugger combines to certain extent the methods proposed in [ZR91], [Sno88] and [Bat89].

[ZR91] does not provide a querying capability, the abstraction, filtering and visualization capabilities provided by [ZR91] also is limited. The EBBA system [Bat89] provides higher level event abstraction mechanism, but ignores precedence information and lacks visualization capability provided by **Hy**$^+$. The query language TQuel in [Sno88] is less expressive than GraphLog. Furthermore, the visual and graphical nature of GraphLog queries allows the debugger express intuitively appealing queries. To see this let us consider the TQuel query on page 170 of [Sno88], which asks "Which processes were resumed by process P1?" (see [Sno88] for the description of the problem and the relations):

range of S **is** SendMessage
range of W **is** Waiting

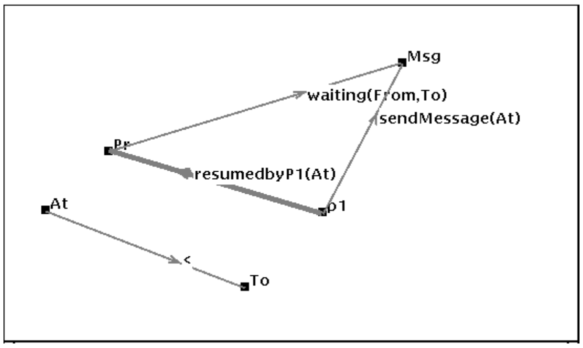

Figure 9: GraphLog query equivalent to the TQuel query.

retrieve ResumedbyP1 (Process = W.Process)
  valid at end of W
  where S.Mailbox = W.Mailbox **and** S.Process = P1
  when S **precede end of** W

The corresponding GraphLog query is shown in Figure 9.

The relations waiting(Pr, Msg, From, to) and sendMessage(p1, Msg, At) are depicted as edges connecting first two attributes which are nodes in the graph. The derived relation resumedbyP1(p1, Pr, At) is shown as a thick edge. Snodgrass's monitor assumes a global clock which does not give an accurate picture of a distributed computation.

6 Conclusion and Future Work

We have described how the **Hy$^+$** data visualization system can be used for postmortem debugging of parallel and distributed programs.

The facilities provided by the **Hy$^+$** system have been shown to be useful for *network management* [CH93]. This suggests that the **Hy$^+$** system can serve as a common front-end tool for the effective development, maintenance and operation of distributed applications. Program behavior can be observed not just in terms of the semantics of the program, but also in terms of the underlying network that the distributed application is running on. 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 1) 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 expressiveness of GraphLog can further be enhanced by adding a temporal dimension to it. GraphLog augmented with temporal operators will be able to express important temporal properties of concurrent programs. We are investigating graphical or GraphLog equivalents of temporal operators capable of reasoning about partially ordered computations. With this addition, in addition to being able to reason about execution traces of a program written in a particular language, we will also be able to model a distributed problem employing some simple formalism. Currently, we are considering Communicating Finite State Machines with bounded buffers as a possible model, since it will be possible to express the model graphically. Within such an extended framework one can prove, for example, the deadlock-freeness and starvation-freeness properties of the dining philosophers problem. The state space generated by the model can be visualized and a variety of interesting queries can be asked, having their result presented visually by the $\mathbf{Hy}^+$ system.

Finally, more work is needed to fully support *on-line* debugging in $\mathbf{Hy}^+$. Ideally, we would like to 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, where the possible pictures have been specified through GraphLog patterns.

Acknowledgments

We are indebted to the other members of the $\mathbf{Hy}^+$ development group: Frank Eigler, Yanni Jew, Emanuel Noik, Dimitra Vista, and Annie Yeung. The CORDS project at the IBM Canada Centre for Advanced Studies, headed by Jacob Slonim, provided a stimulating research environment. We are very much indebted to Willard Korfhage of Polytechnic University, Brooklyn, NY for providing us with Hermes IPC event collector.

References

[Bat89] 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.

[CH93] Mariano Consens and Masum Hasan. Supporting network management through declaratively specified data visualizations. In *Proceedings of the IEEE/IFIP TC6/WG6.6 Third International Symposium on Integrated Network Management, III*. Elsevier North Holland, 1993. To be published.

[Che89] W.H. Cheung. Process and evevent abstraction for debugging distributed programs. Technical report, University of Waterloo, September 1989.

[CM90a] 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.

[CM90b] 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.

[CW89] W.S. Cheng and V.E. Wallentine. DEBL: A knowledge-based language for specifying and debugging distributed programs. *Communications of the ACM*, 32(9):1079–1084, September 1989.

[Fid91] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, pages 28–33, August 1991.

[Fou92] Open Software Foundation. *Introduction to DCE.* Open Software Foundation, 1992.

[GF88] A.J. Gordon and R.A. Finkel. Handling timing errors in distributed programs. *IEEE Transactions on Software Engineering*, 14(10):1525–1535, October 1988.

[Kor92] W. Korfhage. The panorama monitoring system. Technical report, Polytechnic University, Brooklyn, 1992.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[Mil91] 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.

[SBG+91] 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.

[Sno87] R.T. Snodgrass. The temporal query language TQuel. *ACM Transactions on Database Systems*, 12(2):247–298, June 1987.

[Sno88] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

[ZR91] 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.