This is to certify that the

dissertation entitled

The Integration of an On-line Parallel Debugger with a Visualization Methodology for Modeling Expected Behavior

presented by

Joseph L. Sharnowski

has been accepted towards fulfillment
of the requirements for

PhD degree in Computer Science

Betty H.C. Cheng
Major professor

Date 4/21/95

THE INTEGRATION OF AN ON-LINE PARALLEL DEBUGGER WITH A VISUALIZATION METHODOLOGY FOR MODELING EXPECTED BEHAVIOR

By

Joseph L. Sharnowski

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Computer Science Department

1995

ABSTRACT

THE INTEGRATION OF AN ON-LINE PARALLEL DEBUGGER WITH A VISUALIZATION METHODOLOGY FOR MODELING EXPECTED BEHAVIOR

By

Joseph L. Sharnowski

Post-mortem visualizations of program execution are useful for debugging the complex behavior of parallel programs. However, the effectiveness of the visualizations is limited by the characteristics of the information that they present. First, the monitor intrusion and data storage constraints restrict the amount of data that may be recorded during the program execution, and thus restrict the corresponding information presented in the visualizations. Second, the type of information presented by visualizations often limits how well their representations match the programmer's conceptual model of the expected program behavior. The models used for visualizing program behavior typically have focused on depicting low-level events, such as message-passing or procedure calls. These models impose upon the programmer the burden of establishing a match between the graphically-displayed events and the patterns of expected behavior for the program.

This dissertation presents newly developed solutions that address these two

problems. In order to relax the problems associated with the acquisition of data, we use an on-line approach to debugging in contrast to a post-mortem strategy. Based on this approach, the first main contribution of this research is the development of new strategies for applying visualization to the operation of an on-line parallel debugger. In particular, we developed a visualization-based approach for the selection of causal distributed breakpoints. In addition, we developed several techniques for improving the scalability of visualizations that represent process communication and integrated these techniques into a visualization-based debugging environment for supporting top-down examination of program states.

The other main contribution of this research is the development of a methodology for modeling visualizations based on the expected behavior of the program, where the model of expected behavior is generated from the program's formal specification. In particular, we developed a technique for modeling expected behavior in visualizations of communication between processing elements, as well as techniques for illustrating the current location in a program execution in terms of the formal specification. In addition, a technique has been developed for using the formal specifications of a program's data structures to guide the generation of data visualizations.

Dedicated to the memory of my grandmothers,

Helen Lewandowski and Julia Sharnowski,

whose faith in God was an inspiration to everyone around them.

ACKNOWLEDGMENTS

I would sincerely like to thank my advisor, Dr. Betty H. C. Cheng, for all of her help and guidance during my Ph.D. program. Beyond providing direction, she also was an endless source of encouragement. It has indeed been both a pleasure and an honor to have been her student.

I thank the other members of my committee, Dr. Lionel M. Ni, Dr. Diane Thiede Rover, and Dr. David Yen, for their efforts on my behalf. I also wish to thank NSF, as this work has been supported in part by the NSF Grants CCR-9209873 and CCR-9407318.

I wish to thank my family and friends for always being there for me. In particular, I thank my parents for being both supportive and for being ideal role models. I also wish to thank my Goddaughter Sarah, for being a source of both pride and joy. Last, but hardly least, I am deeply grateful to Rana, for everything she has added to my life, and all she will still add.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction

Parallel computing offers a promising approach for achieving the high performance required by computationally-intensive applications. The widespread use of this approach, however, has been limited by the difficulties associated with the development of parallel programs. This dissertation addresses one of the main difficulties, specifically the task of debugging a parallel program.

1.1 Motivations

Debugging a sequential program is a difficult task, as it relies on insight for knowing where to look for the cause of an error. Such insight is often only achieved after years of experience in program development [1]. Debugging a parallel program presents an even more difficult challenge [2], as communication between processing elements complicates the task of locating the cause of an error. In order to simplify the development of parallel programs, support tools must be developed for handling this additional complexity. Several obstacles make the development of such tools difficult, including the nondeterministic behavior of parallel programs, where different executions may result in different sequences of interactions between processing elements. In addition, parallel programs are also susceptible to the *probe*

effect, where an attempt to monitor a program's behavior may actually change the behavior by introducing delays that change the order of synchronizations.

1.1.1 Formal Methods

An indirect solution for addressing the difficulty of parallel debugging is to minimize the number of errors that must be detected and corrected. The application of *formal methods* to the process of software development provides a means for achieving this solution. Formal methods are mathematically-based techniques that are used to describe and reason about properties of software systems. The description of the properties are presented using a notation called a *formal specification language*, and the document in which the properties are described is called a *formal specification* [3]. The formal specification is an abstraction of the software system, where the implementation details are intentionally omitted. Using formal specifications facilitates the early evaluation of a software design through the use of formal reasoning techniques [3, 4, 5]. Additional benefits are that the well-defined syntax and semantics of formal specification languages makes their manipulation amenable to automation.

The use of formal specifications is unfortunately unable to entirely eliminate the possibility of errors in the implementation. For example, even if the specification accurately represents a problem, the process of constructing an implementation for the specification is subject to human coding errors. In particular, the *expected behavior* of the program, as described by the formal specification, may be inconsistent with the *actual behavior*, as revealed during the program's execution. Thus, despite the benefits of using formal methods for program development, we must still address the problem of the additional complexity associated with parallel debugging, as debugging tools are still necessary for eliminating errors that are introduced during the implementation stage.

1.1.2 Using Visualization to Debug Errors

Visualization has been shown to be an effective approach for representing the complex behavior of a parallel program [6, 7, 8, 9, 10, 11]. Large quantities of event data from the program execution can be encapsulated in compact graphical representations. Such visualizations convey program execution from a global perspective, where the communication between processing elements is depicted graphically. These visualizations reveal patterns and discrepancies in the event data more readily than corresponding textual output.

One major difficulty in the use of visualization is that the nondeterministic behavior of parallel programs and the probe effect have both hindered the development of *on-line* visualization-based debugging tools. Such tools are intended to visualize the execution of the program as it is running, but, in the case of the probe effect, in particular, the overhead required to perform the visualization may significantly alter the execution behavior of the program. The typical approach to building visualization-based debugging tools has been, instead, to use a *post-mortem* strategy, where trace data of important events is collected during program execution, but the graphical depiction of the data is performed off-line after execution is complete. By postponing the depiction of the event data until after the program execution completes, the probe effect is not as large as compared to when the visualizations are rendered on-line. The disadvantage to this approach, though, is that the amount of collected data must be limited, else the probe effect will become a significant factor, and, in addition, the generated data may potentially exceed the available storage.

Recent advances have been made in the area of replay techniques, where deterministic re-execution of a parallel program is guaranteed by forcing communication between processing elements to occur in the same order as in the

original execution [12, 13, 14, 15]. Such techniques are useful for the construction of on-line debuggers, where *cyclic debugging* strategies focus on running a program repeatedly, each time making additional progress towards pinpointing the location of an error. Since a cyclic debugging approach is only meaningful when deterministic execution is guaranteed, the use of replay techniques for providing such determinism is beneficial. In addition, since the replay mechanism guarantees that re-execution of the program will be deterministic, the probe effect is prevented from altering the behavior of the program.

Replay techniques facilitate the development of visualization-based, on-line debuggers. Since execution behavior is not affected by the probe effect, on-line visualization is able to produce meaningful results. An additional benefit of this approach is that data storage constraints are no longer a factor since the data is being visualized as it is generated, in contrast to a post-mortem strategy where the data is first stored and then visualized after program execution is complete. Currently, however, the use of visualization in on-line debugging environments has been limited to the depiction of communication between processing elements [16, 17, 18, 19, 20], mainly because replay-based techniques have only recently become well-established.

Another major difficulty in the use of visualization is finding a graphical representation for the event data that fits the programmer's conceptual model of the problem at hand [21]. Event data often consists of low-level events such as message-passing or procedure calls, yet visualizations based directly on these events are often difficult to correlate with the expected behavior of the program. Such visualizations lack any use of abstraction to model the low-level events in terms of high-level behavior, such as stages in the algorithm. In order to debug any errors, the programmer must manually establish a match between the graphically-displayed events and the patterns of expected behavior for the program.

1.2 Research Contributions

Thesis Statement: *Visualizations and formal specifications may be applied in new ways to facilitate the difficult task of parallel debugging. In particular, on-line, replay-based debugging techniques offer new opportunities for applying visualization to support typical debugging operations, such as setting breakpoints and examining a program's state. In addition, formal specifications may be applied to parallel debugging for modeling a program's expected behavior, facilitating a comparison between expected and actual behavior.*

This dissertation presents two major contributions to parallel debugging technology:

1. The development of new visualization techniques for use in an on-line parallel debugger.

2. The development of a methodology for using the formal specification of a program as a basis for visualizing the program execution.

The research discussed in this dissertation specifically applies visualization to the following tasks of an on-line parallel debugger:

- *Visualization-based selection of distributed breakpoints*: Many of the typical on-line debugging operations are more complicated in a parallel debugger as compared to a sequential debugger since multiple processes must be considered instead of a single one. For example, *causal distributed breakpoints* [22] is one approach that has been used for stopping a parallel program at a particular point in its execution. In this approach, a sequential breakpoint in a single process initiates the stop, where the remainder of the processes are halted and restored to the earliest state reflecting all events that occurred before the

sequential breakpoint. A graphical representation for this type of breakpoint scheme is superior to a textual representation, as the logical ordering of events between processes is depicted well by a two-dimensional plot. This type of two-dimensional plot is known as a *space-time diagram* [17, 23, 24], where processes are on one axis and time is on the other, with communication shown as lines between the relevant (process, time) pairs. This dissertation presents a visualization-based approach for setting distributed breakpoints, where a space-time diagram is used to graphically illustrate where each process is located in its execution when a distributed breakpoint stops the program [9, 11]. The programmer may use this interface to insert distributed breakpoints, simplifying the otherwise difficult operation.

- *Scalable techniques for visualization of communication events*: For the cases of programs that use a large number of processes or have a large degree of process communication, visualizations may become cluttered, thereby limiting their benefit. The effectiveness of visualizations is determined by how well they reveal patterns and discrepancies, regardless of how much information is represented. This dissertation describes GUI-based filtering techniques for scaling visualizations of communication [10, 11]. A unique characteristic of these techniques is that they are integrated in a top-down debugging framework, where filtering the degree of information presented in visualizations of communication is the first step when converging towards the source of an error.

- *Top-down use of visualizations for examination of program states*: The advantages of visualizing multiple aspects of program execution have been discussed previously [23]. A replay-based, on-line debugging strategy is well-suited for providing multiple visualizations, as the large volume of data required

to render the visualizations can be acquired without being limited by the probe effect and data storage constraints. A top-down approach for using a variety of visualizations to debug incorrect program behavior may thus be constructed [9, 11, 25]. For example, in a case where incorrect communication between processes is suspected, a visualization of communication events would be useful for initially inspecting the message-passing behavior, where filtering may be applied as described above. As the location of the suspected error is narrowed to a single process or a small subset of processes, the execution of those particular processes may individually be visualized using a (procedure) call graph or a source code listing. Finally, if a particular data structure must be inspected, data visualizations may be used. In the case of data visualizations, filtering may be useful for highlighting particular instances of data. Several filtering possibilities are provided, including choices for filtering values that exceed some maximum value, that fall below some minimum value, that are equal to some critical value, or that are within some critical range.

The task of debugging a parallel program is facilitated by an environment that supports comparisons between the actual behavior of the program and its expected behavior. The second contribution of this dissertation is a new approach to visualizing the execution of a parallel program in the context of the program's formal specification, where the formal specification is used to model the expected behavior of the program. The following items were addressed for this research contribution.

- *Modeling expected behavior in program visualizations*: Visualizations that depict various aspects of the program state are useful for identifying inconsistencies that may indicate the source of an error. In order for these visualizations to be useful for debugging purposes, however, the programmer must be able to find a match between the depicted behavior and the expected program behavior. To

facilitate this task, we developed an approach [8, 26] for visualizing the execution of a parallel program in the context of the program's formal specification, written in the specification language LOTOS (Language of Temporal Ordering Specifications) [27, 28]. A LOTOS specification is used to represent an abstraction of the program, thus providing a model of the program's expected behavior. In the case of communication visualizations, each LOTOS process may be used to represent a phase in the computation, and thus the correlation of message-passing events with computation phases is established by overlapping a representation of the active LOTOS processes onto the diagram. In the case of graphically representing the current location in a single process, the location may be represented within a visualization based on the formal specification. In this latter case, a call graph visualization may be used to depict call relations between the LOTOS processes, while a specification listing may be used to portray the execution flow.

- *Selection of Data Visualizations Using Formal Specifications*: Similar to the item discussed above, visualizations of program data may also be generated based on the formal specification of the program. In this case, however, the formal specification is used to guide the selection of an appropriate data visualization, rather than modeling the visualization in terms of items from the formal specification. Specifically, a method for deriving appropriate parametric data visualizations from formal specifications is presented, where this method maps formal specifications of a program's data structures to a graphical unit termed a *cell model* [7, 29]. The cell model consists of several components used to characterize a data structure. A rule-based system is used for selecting appropriate debugging visualizations depending on the contents of the cell model.

For both areas of research described above, we specifically considered the case of programs written for distributed-memory systems, where communication between nodes is via message-passing [30].

1.3 Organization of Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 gives background information on the LOTOS specification language and the Cholesky factorization program used to generate most of the visualizations shown in this dissertation. Chapter 3 presents an overview of the Graphical On-Line Debugger (GOLD), a prototype debugging environment that was developed as a testbed for several of the research contributions discussed in this dissertation. The discussion of research results is divided into several chapters, where each chapter presents results targeted towards accomplishing one of the research contributions discussed above. In particular, Chapter 4 describes a technique for using visualizations to guide the operation of setting a causal distributed breakpoint for a set of processes. Chapter 5 presents a scalable approach for using visualizations to represent process communication, where this approach focuses on using simple, GUI-based filtering techniques. Chapter 6 introduces a strategy for coordinating the use of different types of visualizations when examining a program state. Chapter 7 discusses a technique for modeling expected behavior in program visualizations. Chapter 8 discusses a method for selecting data visualizations based on the formal specifications of the program's data structures. Related work is presented in Chapter 9. Finally, conclusions and future investigations are discussed in Chapter 10.

CHAPTER 2

Background Information

This chapter discusses background information relevant to concepts that appear later in this dissertation. First, an introduction to the LOTOS specification language is presented, in particular demonstrating how it may be used for the specification of parallel programs. Second, a brief introduction to the Cholesky factorization program is provided, where this program was used to generate most of the visualizations shown in this dissertation.

2.1 Specifying Parallel Programs with LOTOS

LOTOS [27, 28] is a formal specification language specifically designed to specify protocols and services. The concepts of LOTOS are general in nature, however, thus making the language useful for a wide variety of other tasks, including the specification of parallel programs. Several features of LOTOS influenced its use for our research: it has been internationally standardized, thus promising a stable definition; it is relatively user-friendly compared to other specification languages, as provided through the usage of keywords and hierarchically-structured specifications; and there exist public-domain tools for working with LOTOS, including syntax and semantics checkers. In the following discussion, we present a brief introduction to

LOTOS, followed by a description of how the language may be used to specify a parallel computing environment. We also include the specification of a simple parallel program.

2.1.1 Introduction to LOTOS

The following is an abbreviated introduction to the LOTOS specification language. We limit our overview to the concepts and notation relevant to the specification of parallel and distributed systems. For a more thorough introduction to LOTOS, the reader may refer to the tutorials provided in [28, 31].

LOTOS consists of two components:

- A *data type* component, based on the algebraic specification language ACT ONE [32].
- A *control* component, based on the process algebras CCS [33] and CSP [34].

The data type component supports the specification of data structures. The definition of a data type includes a *sort* name used to identify instances of the data type, a list of *operations* that may be performed on the data type, and a set of *equations* that describe properties of the operations. In order to facilitate the task of constructing specifications, LOTOS supports a library of standard data types, such as boolean and natural number.

The control component of LOTOS permits a system to be described in terms of *process definitions*, where a process[1] is an entity that may execute either external *actions* (events) or internal, unobservable actions. In the former case, synchronization with the other processes in the system is required, taking place at synchronization points called *gates*. The system as a whole is viewed as a single process, possibly

[1]The meaning of a LOTOS process is distinguished from the meaning of an operating system process, where, in the latter case, a process refers to the execution of a sequential program.

composed of several subprocesses. Each subprocess is a process in itself, and thus the entire system may be viewed as a hierarchy of processes.

The general form of a LOTOS action consists of three components:

- A gate name
- A list of attributes, where each attribute can either be a value offered at the gate (labeled with "!") or a variable accepted at the gate (labeled with "?")
- An optional predicate that establishes a condition on the variables that can be accepted

An example action is the following (where "Nat" is the natural numbers sort):

g !3 ?V:Nat [V < 2]

The predicate for this action asserts that the variable V can only accept values that are less than 2, namely 0 or 1 since V is of sort natural number. Thus, the action is equivalent to one that provides a choice between the two actions "g !3 !0" and "g !3 !1", offering the values 3 and 0 or 3 and 1, respectively, at gate g.

Processes communicate by synchronizing on common actions. For example, if a process P is ready to execute the action "g !3" at the same time that a process Q is ready to execute the action "g ?Y:Nat [Y < 7]", then synchronization may occur between processes P and Q on the common action "g !3".

Two special processes are defined in LOTOS: **stop** and **exit**. The process **stop**, representing a completely inactive process, is invoked by an active process in order to complete a transition to an inactive state. The process **exit** is a process with the sole purpose of executing a successful termination action, after which it invokes the process **stop**.

A *behavior expression* models the activity of a process. Behavior expressions are built from actions and other behavior expressions by using a predefined set of operators. The operators relevant to the specifications in this dissertation are the following (where c and d are actions, and B is a behavior expression):

- *Action prefix* operator: written as a semicolon ";"
  This operator supports the sequential composition of actions and behavior expressions. For example, "c ; d ; B" means that action c is followed by action d, which, in turn, is followed by behavior expression B.

- *Choice* operator: written as "[ ]"
  This operator denotes the choice between two or more actions or behavior expressions. For example, "(c ; B) [ ] (d ; B)" represents a choice between either action c followed by behavior expression B, or action d followed by behavior expression B.

- *Parallel composition* operators: written as "|[G]|", |||, and ||
  The *selective parallel* operator "|[G]|" is used to express parallelism between two behavior expressions, where G is the set of gates on which synchronization must occur. Any actions in the two behavior expressions occurring at gates not in the set G may interleave. The special case in which G is empty is represented by the *interleaving* operator |||, and the special case in which G contains the set of all gates is represented by the *full synchronization* operator ||. In order to illustrate the use of the parallel composition operators, consider the case where c denotes the action "g !3" and d denotes the action "g !2". Then "(c ; **exit**) |[g]| (d ; **exit**)" is equivalent to **stop**, since synchronization cannot occur between "g !3" and "g !2". However, "(c ; **exit**) ||| (d ; **exit**)" is equivalent to "(c ; d ; **exit**) [ ] (d ; c ; **exit**)" since the interleaving operator requires no synchronization.

A behavior expression may be preceded by a *guard*, where the guard must be true in order for the expression to be invoked. The guard has the form of a predicate followed by an arrow. For example, in the following guarded expression, behavior expression B will only be invoked if the value of X is equal to 2:

$$[\,X = 2\,] \rightarrow B$$

A typical structure for a LOTOS specification is of the following form, where *functionality* is either **exit** or **noexit**, depending on whether the process is able to successfully terminate [28]:

specification *name_of_spec* [*gate list*](*parameter list*): *functionality*
    ⟨ *type definitions* ⟩
behavior
    ⟨*behavior expression for the overall specification*⟩
where

⟨*process definitions*⟩
endspec

where each process definition has the form:

process *name_of_process* [*gate list*](*parameter list*): *functionality* :=
⟨*behavior expression for the process*⟩
where
⟨*subprocess definitions*⟩
endproc

2.1.2 Specification of the Parallel Computing Environment

A LOTOS specification for a parallel program requires the specification of the entire environment of the program, where the environment consists of both the processing elements where the computation occurs and the medium through which the processing elements can communicate. For the research discussed in this dissertation, we consider the case of parallel programs written for distributed-memory systems, where communication channels interconnect the nodes to provide a medium for message-passing. A general technique for the LOTOS specification of such environments has been developed [35], from which we adapted our particular approach. We outline the key aspects of our approach below.

A simple example of a parallel computing environment is two processing elements $P1$ and $P2$ connected via two unidirectional channels $C1$ and $C2$, as shown in Figure 2.1. The synchronization points between the processing elements and the channels are the gates labeled *send* and *recv*. In accordance with previously developed notation [35], an action that sends a message *Mesg* from a processing element labeled *Sender* to a processing element labeled *Rcvr* may be formatted as:

send !*Sender* !*Rcvr* !*Mesg*

Similarly, the action that receives the message that was sent as described above may be formatted as:

recv !*Sender* !*Rcvr* ?*Mesg:Message*

where *Mesg* is a variable whose value is set through synchronization.

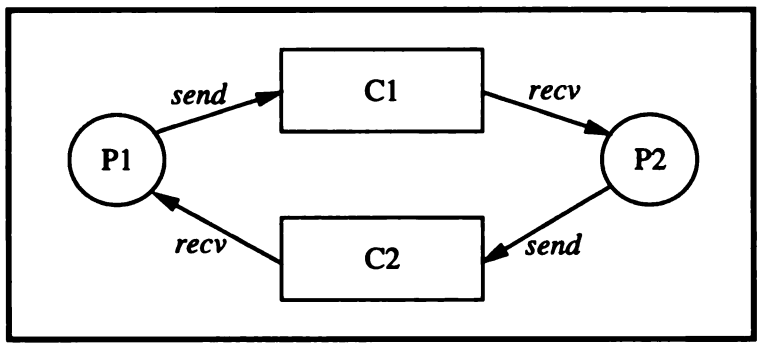

Figure 2.1. Simple parallel computing environment

Consider a program for the computing environment given in Figure 2.1, whose purpose is to send a natural number from $P1$ to $P2$, and then send the same number back again from $P2$ to $P1$. The specification for this program is shown in Figure 2.2. The behavior expression for the overall specification states that the actions of the processing elements ($P1$ and $P2$) may interleave, and the actions of the channels ($C1$ and $C2$) may also interleave, but the actions between the processing elements and the channels must fully synchronize (i.e., synchronize on both the *send* and *recv* gates). By examining the behavior expressions in the four process definitions of the specification, we observe that the first synchronization occurs between $P1$ and $C1$ at the *send* gate. This synchronization causes the variable *Msg* in the action "send !P1 !P2 ?Msg:Nat" to accept the value *Some_Num* offered by the action "send !P1 !P2 !Some_Num", effectively representing the passing of a message from the processing element $P1$ to the channel $C1$. Similarly, the next three synchronizations that occur pass the message from $C1$ to $P2$, then from $P2$ to $C2$, and finally from $C2$ to $P1$, at which point all four of the LOTOS processes $P1$, $P2$, $C1$, and $C2$ have successfully

```
specification simple_send_and_return(Some_Num:Nat): exit
library NaturalNumber endlib
behavior
   (
      ( P1[send,recv](Some_Num) ||| P2[send,recv] )
   ||
      ( C1[send,recv] ||| C2[send,recv] )
   )
where
   process P1[send,recv](Some_Num:Nat):exit :=
      send !P1 !P2 !Some_Num ; recv !P2 !P1 ?ReturnMsg:Nat ; exit
   endproc
   process P2[send,recv]:exit :=
      recv !P1 !P2 ?Received_Num:Nat ; send !P2 !P1 !Received_Num ; exit
   endproc
   process C1[send,recv]:exit :=
      send !P1 !P2 ?Msg:Nat ; recv !P1 !P2 !Msg ; exit
   endproc
   process C2[send,recv]:exit :=
      send !P2 !P1 ?Msg:Nat ; recv !P2 !P1 !Msg ; exit
   endproc
endspec
```

Figure 2.2. Specification for transmitting a number between $P1$ and $P2$

terminated, such that the specification may also successfully terminate.

The model that we use for a general parallel computing environment supports an unlimited number of processing elements, where each pair of processing elements is connected by two unidirectional channels, as shown in Figure 2.1 for the case of $P1$ and $P2$. We assign each processing element with a natural number for its name, where the advantage of this naming scheme is discussed below. The number zero is assigned to a special processing element called the *host node*, whose purpose is to handle management duties for the set of processing elements, such as assigning tasks or data

sets to the other processing elements. We refer to the other processing elements in the environment as *worker nodes*, assigning them the names $1, \ldots,$ *TOTAL_WORKERS*, where *TOTAL_WORKERS* refers to the number of worker nodes in the environment. The purpose of worker nodes is to collectively complete the main computation for a problem by each performing a portion of it. For clarity purposes, we distinguish between the following types of channels: those that connect the host to the worker nodes and those that connect pairs of worker nodes. We assume for simplicity that processing elements do not fail, that messages are not lost, and that unbounded message size is supported by the environment.

The behavior expression for the overall specification of the computing environment indicates that the actions of the host and worker nodes interleave, and it also indicates that the actions of the two types of channels interleave. But it requires the actions between the nodes and the channels to fully synchronize, as was the case for the specification shown in Figure 2.2. The behavior expression for the overall specification is shown in Figure 2.3.

The individual instances of worker nodes and the two types of channels are created using recursion, based on the naming scheme for the processing elements, as shown in Figure 2.4. The *NodeCtr* variable is used to recursively count in descending order through all the "names" of the worker nodes, from *TOTAL_WORKERS* to one (the process definitions for creating the instances of the channels are similar in nature).

The required behavior of a channel is to forward a message from a sender to a receiver, and then make a recursive call to itself in order to wait for the next message that needs to be forwarded. An example of the process definition for a channel for the case of a channel from a host to a worker is shown in Figure 2.5.

The specification of the behavior of the *Host_Node* and *Worker_Node* processes depends on the particular program that is being specified.

The full specification for the parallel computing environment is presented in the

```
(
      Host_Node[send,recv](TOTAL_WORKERS)
         |||
      All_Worker_Nodes[send,recv](TOTAL_WORKERS,TOTAL_WORKERS)
)
  ||
(
      All_Host_to_Worker_Channels[send,recv](TOTAL_WORKERS)
         |||
      All_Worker_to_Worker_Channels[send,recv](TOTAL_WORKERS,
                                    TOTAL_WORKERS)
)
```

Figure 2.3. Behavior expression for the overall specification

Appendix.

2.1.3 Specification of a Simple Program

The specification discussed above for a parallel computing environment is generic in nature, such that it may be used for specifying any parallel program that communicates via message-passing. Each particular program will have unique process definitions for the host and worker nodes, but the remainder of the specification will be approximately the same.

A typical structure for a message-passing parallel program is for the host node to distribute work to each of the worker nodes, where the worker nodes then perform their assigned duties, perhaps communicating among themselves to do so. Finally, the worker nodes return the results to the host nodes, thus completing that stage of computation. In the case of multiple stages of computation, the host node may then assign a new set of tasks to the worker nodes, and the cycle continues.

```
process All_Worker_Nodes[send,recv](NUM_WRKRS,NodeCtr:Nat):
                                        noexit :=
    [NodeCtr > 0] →
        Worker_Node[send,recv](NUM_WRKRS,NodeCtr)
            |||
        All_Worker_Nodes[send,recv](NUM_WRKRS,NodeCtr - 1)
endproc (* All_Worker_Nodes *)
```

Figure 2.4. Recursive creation of worker nodes

```
process Host_to_Worker_Channel[send,recv](Sender,Rcvr:Nat): noexit :=
    send !Sender !Rcvr ?Msg:Message;
    recv !Sender !Rcvr !Msg;
    Host_to_Worker_Channel[send,recv](Sender,Rcvr);
endproc (* Host_to_Worker_Channel *)
```

Figure 2.5. Process definition for a channel

A simple example of a message-passing parallel program that fits the above description is one in which the host node sends each worker node a natural number equal to the value of the worker node's name, where the worker node then doubles the received number and returns the result to the host. The specifications for the host and worker node processes of this number-doubling program are shown in Figure 2.6. The behavior expression for the host node illustrates the use of process decomposition, where the subprocesses *Send_Numbers* and *Receive_Replies* perform the tasks of sending messages to the worker nodes and receiving the results,

```
process Host_Node[send,recv](NUM_WRKRS:Nat): exit :=
(
   Send_Numbers[send,recv](NUM_WRKRS);
   Receive_Replies[send,recv](NUM_WRKRS);
   exit
)
where
   (* Send_Numbers uses recursion to send a number to each worker node *)
   process Send_Numbers[send,recv](NodeCtr:Nat): exit :=
      [NodeCtr > 0] →
      (
         send !0 !NodeCtr !NodeCtr ;
         Send_Numbers[send,recv](NodeCtr - 1)
      )
      [ ]
      [NodeCtr = 0] → exit
   endproc (* Send_Numbers *)

   (* Receive_Replies uses recursion to receive numbers from worker nodes *)
   process Receive_Replies[send,recv](NodeCtr:Nat): exit :=
      [NodeCtr > 0] →
      (
         (* Since Sender is a variable, receive the replies in any order *)
         recv ?Sender:Nat !0 ?Reply:Nat ;
         Receive_Replies[send,recv](NodeCtr - 1)
      )
      [ ]
      [NodeCtr = 0] → exit
   endproc (* Receive_Replies *)
endproc (* Host_Node *)

process Worker_Node[send,recv](NUM_WRKRS,MY_NUM:Nat): exit :=
   (* Receive a number from the host node *)
   recv !0 !MY_NUM ?Value:Nat ;
   (* Send twice the value of the number back to the host node *)
   send !MY_NUM !0 !(Value+Value) ;
   exit
endproc (* Worker_Node *)
```

Figure 2.6. Specification of nodes for the number-doubling program

respectively. The *Send_Numbers* process requires the messages to be sent to the worker nodes in decreasing order of the value of the worker nodes' names, but the *Receive_Replies* process receives the results in any order.

2.2 Cholesky Factorization Program

The application we used to generate most of the visualizations shown in this dissertation is the Cholesky factorization program supplied with the PVM 2.4 [36] distribution, modified to run under PVM 3.3. The executions occurred on clusters of identical, ethernet-connected SUN SPARCstation 10 workstations. In the brief background discussion below, we present an informal description of the parallelized program, followed by LOTOS specifications of relevant portions of the program.

2.2.1 Informal Description of the Program

Cholesky factorization considers the special case in which a matrix A is both symmetric and positive definite. In this case, matrix A has a factorization of the form $A = LL^T$, where L is a lower triangular matrix. This factorization is known as the *Cholesky* factorization.

The program that we use to compute the Cholesky factorization is a Column-Cholesky [37] implementation, in which the processing elements are each assigned an approximately equal number of columns for the computational tasks, although not necessarily consecutive. The implementation consists of three phases: synchronous Cholesky factorization, forward substitution, and backward substitution. A full discussion of these phases is beyond the scope of this dissertation, although the interested reader may refer to other references [37, 38]. In the following presentation, we focus on the message-passing events for each of the phases. We do not discuss the contents of the messages, but rather focus on the patterns in which they are sent. The

matrix we consider is of size $n \times n$, where the columns are numbered $0, \ldots, n-1$. The number of available worker node processes is represented by *TOTAL_WORKERS*.

All three phases of the program contain message-passing events within loops that use the column number as the index variable. In the case of the synchronous Cholesky factorization phase, the iteration is in order of increasing column number, from 0 to $n-1$. During each iteration of the loop, the processing element checks if it is assigned the column corresponding to the value of the index variable. If so, the processing element sends a broadcast message to the other processing elements. Otherwise, it receives the message broadcast from the processing element that actually is assigned the column corresponding to the value of the index variable.

The message-passing behavior in the backward substitution phase is similar to that described above for the synchronous Cholesky factorization phase, except that the iteration is in order of decreasing column number, from $n-1$ to 1 (column zero is skipped).

The forward substitution phase iterates in order of increasing column number, from 1 to $n-1$ (column zero is again skipped), but the message-passing behavior is different from that of the other two phases. In this case, if the processing element determines that it is assigned the column corresponding to the value of the index variable, then it waits to receive values sent from each of the other processing elements. Otherwise, it sends a message to the processing element that is assigned the column.

2.2.2 LOTOS Specifications for the Program

In this example, the host node process participates in the computation only during initialization, where the worker node processes perform the majority of the work. The behavior expression for the worker node processes consists of the sequential composition of an initialization process with three processes corresponding to the three stages of the program:

Initialization[send,recv](⟨List of Local Variables⟩);
Synchronous_Cholesky_Factorization[send,recv](⟨List of Local Variables⟩);
Forward_Substitution[send,recv](⟨List of Local Variables⟩) ;
Backward_Substitution[send,recv](⟨List of Local Variables⟩)

The error we consider below is located within the *Forward_Substitution* process, and so we focus our discussion on that process. Its behavior expression consists of a set of initialization operations, followed by the main loop that iterates in order of increasing column number:

For_Sub_Pre-loop[send,recv](⟨List of Local Variables⟩);
For_Sub_Main_Loop[send,recv](1,n-1,⟨List of Local Variables⟩)

where the header and behavior expression for the *For_Sub_Main_Loop* process is shown in Figure 2.7. The *CurrentCol* variable within the *For_Sub_Main_Loop* process is used to iterate in order of increasing column number through recursive calls to the *For_Sub_Main_Loop* process.

```
process For_Sub_Main_Loop[send,recv](CurrentCol:Nat,MAX_COL:Nat,
                                ⟨List of Local Variables⟩): exit :=
[ CurrentCol ≤ MAX_COL] →
(
   For_Sub_Pre-communication[send,recv](⟨List of Local Variables⟩);
   For_Sub_Communication[send,recv](CurrentCol,⟨List of Local Variables⟩);
   For_Sub_Post-communication[send,recv](⟨List of Local Variables⟩);
   For_Sub_Main_Loop[send,recv](CurrentCol+1,MAX_COL,
                                ⟨List of Local Variables⟩)
)
[ ]
[CurrentCol > MAX_COL] → exit
```

Figure 2.7. Header and behavior expression for *For_Sub_Main_Loop*

Finally, the specification of the *For_Sub_Communication* process is shown in

Figure 2.8. The behavior expression for this process is divided into two choices

```
process For_Sub_Communication[send,recv](CurrentCol:Nat,
                                        ⟨List of Local Variables⟩): exit :=
    [CurrentCol ∈ MY_COL_SET] →
        For_Sub_Receive_Messages[send,recv](0,TOTAL_WORKERS);
    [ ]
    (   [CurrentCol ∉ MY_COL_SET] →
            send !MY_NUM !Owner(CurrentCol) !Msg:Message ;
            exit    )
    where
        process For_Sub_Receive_Messages[send,recv](NodeCtr:Nat,
                                        TOTAL_WORKERS:Nat): exit :=
            [NodeCtr < (TOTAL_WORKERS - 1)] →
            (* Since Sender is a variable, receive messages in any order *)
            (   recv ?Sender:Nat !MY_NUM ?Msg:Message ;
                For_Sub_Receive_Messages[send,recv](NodeCtr + 1,
                                        TOTAL_WORKERS)    )
            [ ]
            [NodeCtr = (TOTAL_WORKERS - 1)] → exit
        endproc (* For_Sub_Receive_Messages *)
endproc (* For_Sub_Communication *)
```

Figure 2.8. Specification of the *For_Sub_Communication* process

that determine whether the worker node should participate in send or receive actions. The selection between the two choices is based on whether the *CurrentCol* variable belongs to the set of columns assigned to the worker node that has invoked the process. In the case where the column is assigned to a worker node, the recursive *For_Sub_Receive_Messages* process is used to receive *TOTAL_WORKERS-1* messages sent by the other worker nodes.

CHAPTER 3

Overview of the GOLD Debugging Framework

In order to provide a context for the presentation of the research results, this chapter describes GOLD [11], a graphically-based debugger that we developed to give a proof of concept demonstration of our research. GOLD may be used to debug parallel and distributed programs written in C and compiled using the GNU compiler gcc, and has currently been ported to Sun workstations running either SunOS 4.1.x or Solaris. GOLD supports programs that have been written using PVM 3.3 [39] message-passing primitives, where the programs consist of a single host process that spawns a collection of worker processes. A PVM program is typically designed to parallelize a computation by having the host process distribute work to a collection of worker processes. In this chapter, the top-down procedure for debugging using GOLD is presented. Next, the general architecture of GOLD, which facilitates top-down debugging, is discussed.

3.1 A Top-down Debugging Procedure

GOLD is designed to support a top-down procedure for locating the source of an error. The steps in the top-down procedure are as follows.

1. Construct a hypothesis regarding the potential location of the source of the error.

2. Establish one or more program states to examine in order to prove or disprove the hypothesis.

3. For each program state from Step 2:

   (a) Execute the program until the desired program state is reached.

   (b) Examine the program state using a top-down approach, such that the state is investigated from a broad scope initially, but the scope narrows as debugging proceeds in order to examine the potential source of an error in greater detail.

4. If the source of the error has been located, then quit; Else, construct a new hypothesis based on what has been learned so far, and return to Step 2.

Therefore, the procedure for debugging is top-down in two respects. First, the overall procedure uses a top-down strategy for deriving hypotheses that incrementally make progress towards the source of an error. Second, Step 3(b) of the procedure uses a top-down strategy for examining a particular program state.

The visualization-based features of the GOLD debugging environment are designed to support the top-down debugging procedure. In particular, the visualization-based approach to breakpointing supports the task of stopping the program in a specific state (Step 2), while the strategy for coordinating the application of different types of visualizations supports the examination of a program state (Step 3(b)). The GUI-based techniques for filtering communication events are useful when

a hypothesis concerns incorrect message-passing behavior, where it is helpful to depict only the relevant communication activity during Step 3(b).

3.2 The General Architecture of GOLD

The top-down procedure described above defines an iterative approach to debugging. The basis for iterative debugging is to execute a program repeatedly, each time gathering more information to be used towards identifying the source of an error. The difficulty in using an iterative approach for parallel debugging is that any attempt to monitor the behavior of a parallel program may actually change its behavior. Referred to as the *probe effect* [40], the debugging routines could potentially cause the order of message-passing transactions between processes to change such that the erroneous behavior may be difficult to reproduce.

Replay strategies [12] have been devised for providing deterministic re-executions of parallel programs. The basis of a replay strategy is to record the communication order between processes during an initial program run. No additional debugging operations are performed during the initial run, thus minimizing the possibility that the erroneous behavior will be concealed. Any additional program runs are guaranteed to be equivalent by forcing the communication to occur in the same order as in the initial run. Replay strategies thus facilitate an iterative approach to parallel debugging.

The basic operation of GOLD is based on a replay strategy. In order to debug a PVM program, the program is recompiled with an instrumented version of the PVM library, where the instrumentation handles all details of the replay-based operation. An initial run of the recompiled program is first used to collect the necessary data for supporting program replay, where the program is run using its normal command-line parameters. During the re-executions, the program is executed by typing `gold`

followed by the name of the program and its command-line parameters. For example, if the program is typically invoked by typing "host datafile", then the following line would be used to invoke the re-execution:

```
gold host datafile
```

The replay-based operation of GOLD ensures deterministic re-execution, and so GOLD has been constructed as an on-line debugger, where debugging operations are performed during the re-executions. This approach is in contrast to a post-mortem strategy for debugging, where important events are recorded during a program execution, but the examination of the event data is performed off-line after execution is complete. Although the overhead required by GOLD to perform on-line debugging operations may slow the execution of the program to a small degree, the replay-based operation guarantees that the behavior will maintain its equivalence with the initial run.

A Central Operations Module (COM) is used to control the global activity during the execution of a parallel program. The interface for this module is given in Figure 3.1. Menu items for setting or removing a breakpoint distributed across

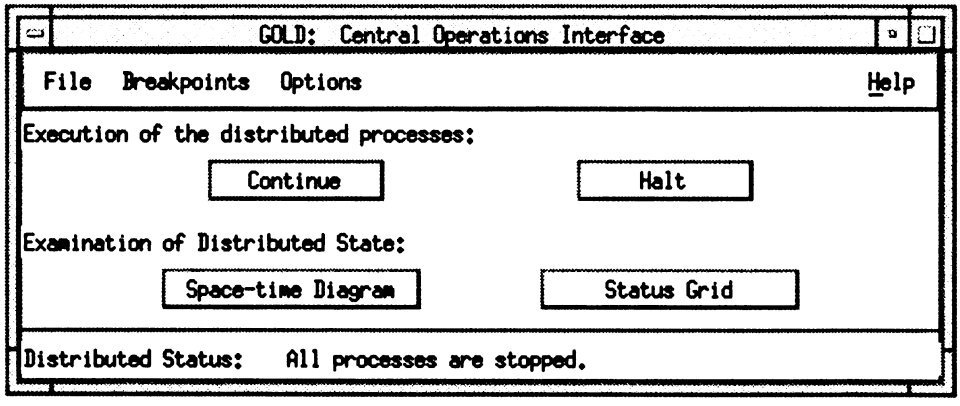

Figure 3.1. The central operations interface

the set of processes are accessible through the Central Operations Interface (COI), and options are available for activating visualizations to examine the state of the set of processes. The interface also provides options for controlling the execution of the parallel program, including **Continue** for restarting the execution after the processes have all stopped and **Halt** for terminating execution. (A "start" option is not applicable for the case of PVM programs, since the host process is started when **gold** is initially activated, and the worker processes are started during the execution of the host process via a **pvm_spawn** system call.) A **Status** line indicates the current execution status for the set of processes.

The execution of individual processes is managed by Local Operations Modules (LOMs), one per process. An overview of the architecture for the GOLD debugging environment is shown in Figure 3.2, where the relationship between the COM and

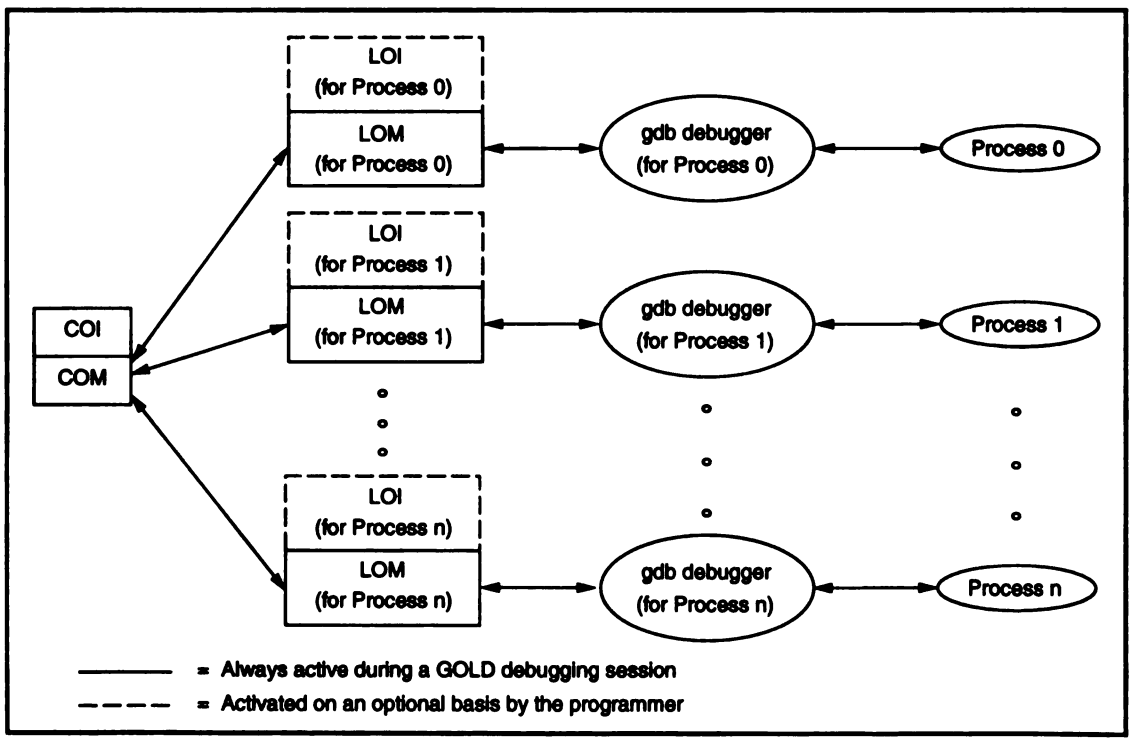

Figure 3.2. The architecture of GOLD

the LOMs is illustrated. The LOM uses the GNU debugger **gdb** to perform low-level debugging operations, such as inserting breakpoints and determining the values of particular variables. As such, a gdb process directly controls the execution of the PVM process, and the LOM controls the execution of the gdb process. The LOM modules communicate with the COM in order to coordinate the execution of the parallel program. For example, in order to continue execution after the processes have all stopped, the programmer would select the **Continue** button from the COI (Figure 3.1), and the COM will then send an appropriate message to each of the LOMs. Each LOM will then request the local debugger to continue execution of its PVM process.

The top-down procedure for debugging facilitates the programmer's ability to move towards the source of an error. As such, the programmer will often narrow the search to the execution of a particular process. In order to examine this local execution more closely, a local operations interface (LOI) is supplied, as shown in Figure 3.3. Various options are provided for controlling the execution of the single process, including **Start** for initiating execution, **Continue** for restarting execution after the process has been stopped at a local breakpoint, and **Step** and **Next** for continuing execution one line at a time, where **Step** sequences through the individual statements of a called function and **Next** treats a function call as a single line. A **Status** line indicates the execution status for the process, and additional fields indicate the current point of execution in terms of the source code, including the function, the source code file, and the line number within the source code file. The LOI also offers options for managing breakpoints and for examining the state of the local process. In particular, menu items are available for setting or removing a local breakpoint, and call graphs and source code listings may be activated to examine the current state of the process. The bottom portion of the LOI displays messages that are generated by the local debugger or by the execution of the program (e.g., output

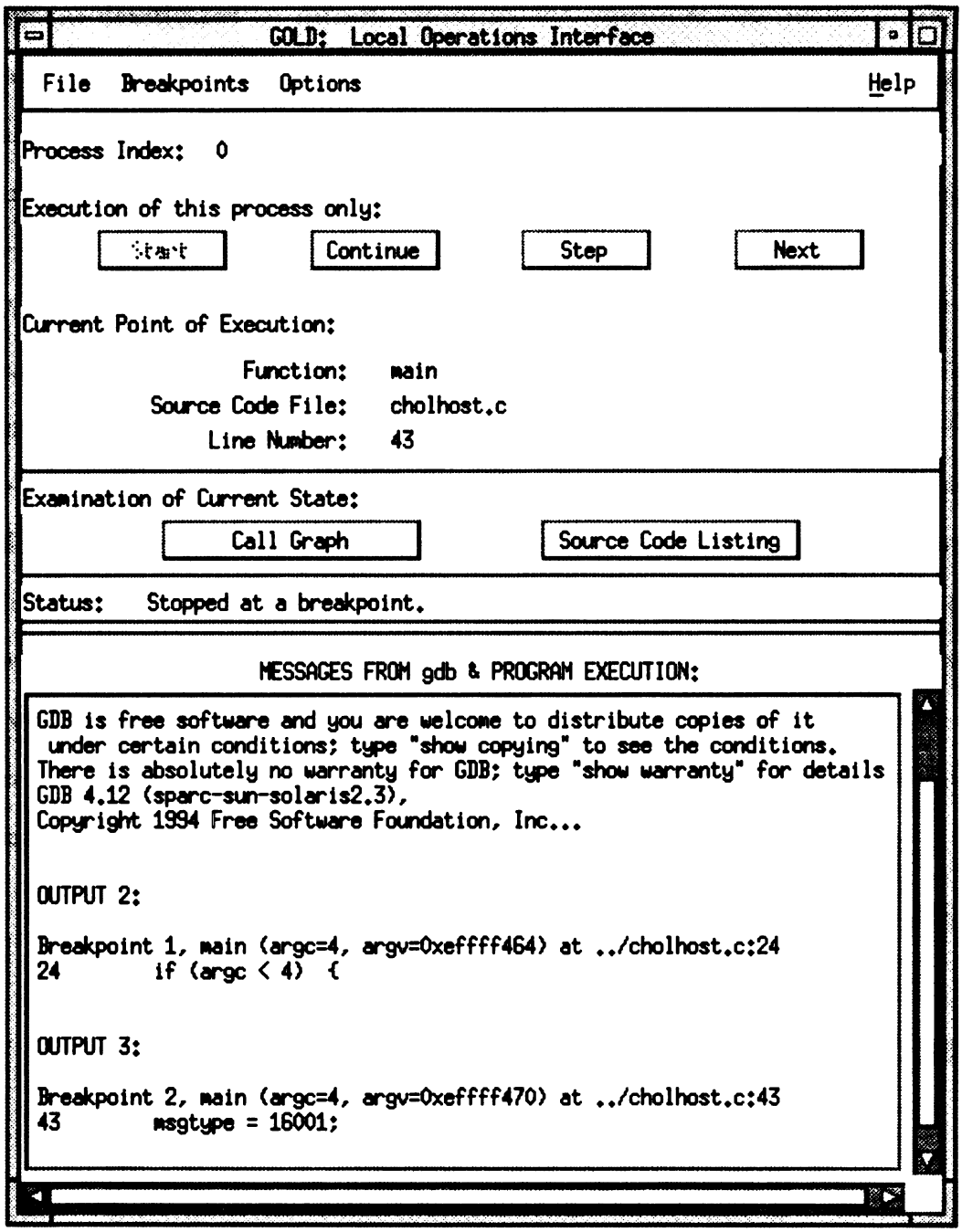

Figure 3.3. The local operations interface

from a call to `printf`).

The LOI for a process is activated as an option by the programmer, such that it remains deactivated by default. By using this approach, the potential problem of flooding the windowing environment with a large number of active interfaces (windows) is avoided. In contrast, however, the COI remains active throughout the debugging session, as it is required for controlling the overall execution of the parallel program.

A status grid window is provided via the COI for displaying the current status of each of the PVM processes used in a particular program execution, where different color shadings represent different status values (e.g., running, stopped by a breakpoint, etc.). A LOM notifies the COM whenever a change in the status of its local process occurs, and the status grid is then updated accordingly. An example of the status grid for an execution using a host process and eight worker processes is shown in Figure 3.4. (The host process is always assigned the node number 0, while the worker processes are assigned the node numbers $1, \ldots, n$ for the case of n worker processes.) In the case shown in Figure 3.4, process 6 is currently running, while the remainder of processes are each stopped at a breakpoint.

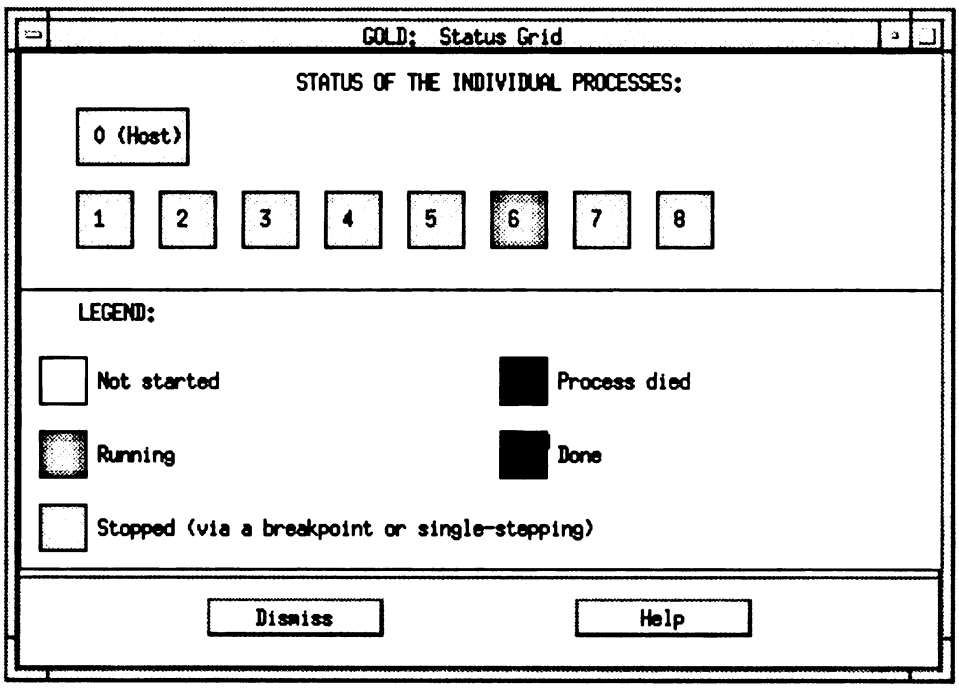

Figure 3.4. The status grid window

CHAPTER 4

Visualization-based Breakpointing

When attempting to determine the source of an error, the programmer marks a particular program state for examination by placing a breakpoint at that state. This chapter discusses the graphically-based approaches for establishing breakpoints using GOLD [9, 11].

4.1 Distributed Breakpoints

The task of setting a distributed breakpoint for a set of processes has been considered previously [22, 41, 42, 43]. The novelty of our approach is the use of visualization for supporting the operation of setting a breakpoint. The discussion of distributed breakpoints is presented in two parts, where we first present background information, and then discuss the visualization-based approach used by GOLD.

4.1.1 Background Information

Process communication in a message-passing parallel program occurs via *send* and *receive* events. Other types of events that occur local to a single process are referred to as *internal* events [42]. The entire collection of events that occur during a program execution is partially ordered by the "happened before" relation, denoted "$\rightarrow$" [44].

This relation is defined as follows.

1. If a and b are events in the same process, and b follows a, then $a \rightarrow b$.

2. If a is a send event in one process, and b is the corresponding receive event in another process, then $a \rightarrow b$.

3. If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$.

If $a \not\rightarrow b$, $b \not\rightarrow a$, and $a \neq b$, then a and b are referred to as *concurrent* events.

The state of a process (or a collection of processes) that has been stopped is referred to as the *breakpoint state.* A common technique for stopping the execution of a group of processes is to have a breakpoint located in a single process trigger the operation. We refer to this particular process as the *triggering process*, and the relevant breakpoint is known as the *triggering breakpoint.* When the triggering breakpoint is reached, the remaining processes may be stopped using a variety of strategies. One such strategy is to have the triggering process send an appropriate message to each of the other processes requesting them to stop. However, this strategy is unfavorable for debugging purposes, as it may hide the source of an error. For example, consider the case of the two processes P and Q shown in Figure 4.1. For this case, a triggering breakpoint occurs in process P at the location marked with an "X", where a message (indicated by a dotted arrow) is then sent to process Q in order to stop it at time t_{Q_2}. However, in order to determine if process communication led to an error at point "X" in process P, process Q should be examined at time t_{Q_1}, as that was its state when it last had a causal effect on process P. By waiting until t_{Q_2} to examine process Q, any state information regarding the message that was sent at t_{Q_1} may change in the intervening time.

Another strategy by which a process may stop the global execution when it reaches a triggering breakpoint is to simply let the other processes run until they block waiting

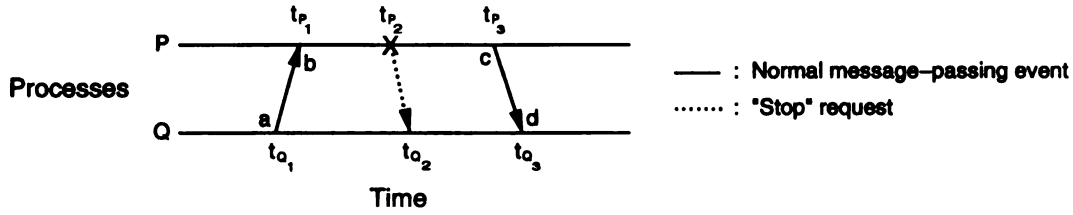

Figure 4.1. Stopping global execution using a triggering breakpoint

for some event to occur in the triggering process. For example, in the case shown in Figure 4.1, the triggering breakpoint at point "X" will eventually cause process Q to block when it attempts the receive event d, since the send event c will not have occurred yet. We can also observe from this example that process Q is again allowed to proceed beyond the last state that had a causal effect on process P, and thus this strategy for stopping the global execution has the same disadvantage as the previous one.

In contrast, the strategy we use for stopping the global execution of a group of processes is based on a *causal distributed breakpoint* [22]. This type of breakpoint is one in which a single process stops at a triggering breakpoint and the remaining processes are stopped at the last event that "happened before" the triggering breakpoint. This last event is referred to as a *causal breakpoint event.* A causal distributed breakpoint stops each process at the last point that had a causal effect on the execution of the triggering process. An example of a causal distributed breakpoint is shown in Figure 4.2. The triggering breakpoint is located at point "X" of process P, and the dashed line indicates where each process is stopped by the causal distributed breakpoint.

GOLD computes a causal distributed breakpoint by first executing the initial program run. At the end of the normal execution of the initial run, each process completes a post-processing stage before it terminates, where a *dependency vector* is

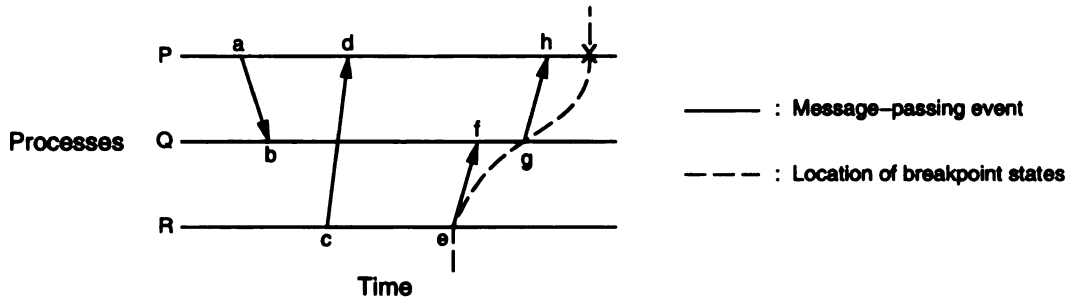

Figure 4.2. Example of a causal distributed breakpoint

recorded for each send and receive event performed by that process. The send and receive events for a process are assigned consecutively-numbered event indices, such that a dependency vector for event ϵ_i of process i is defined as follows [22]:

$$DV_i^{\epsilon} = \langle \delta_1, \delta_2, \ldots, \delta_n \rangle$$

where n is the number of processes. Component δ_i of the vector is set equal to ϵ_i. Each message is appended with the value of the current event index from the sender, such that the value of component δ_j, $j \neq i$, is set equal to the value of the event index from the last message received by process i from process j. For the case in which no message has been passed from process j to process i yet, the value of δ_j is set equal to $\perp$, a value less than all possible event indices.

At the end of the post-processing stage, the dependency vectors from each process are copied to a common location accessible by the COM. The combined collection of dependency vectors is used for computing the causal distributed breakpoints. A dependency vector, however, only records the direct causal dependencies that exist at a particular event position. In order to compute a causal distributed breakpoint, transitive dependencies must also be considered. The visualization-based operation for setting a causal distributed breakpoint uses an algorithm that was specifically designed to consider transitive dependencies during the computation of a causal

distributed breakpoint [22].

4.1.2 Using Visualizations to Set Causal Distributed Breakpoints

GOLD uses a visualization-based approach to support the operation of setting a causal distributed breakpoint [9, 11]. We refer to the operation of setting a breakpoint as the *insertion* of the breakpoint. The **breakpoint** menu from the COI (shown in Figure 3.1) may be used to activate the breakpoint insertion interface shown in Figure 4.3. A primary component of this interface is the communication graph, referred to as a *space-time diagram* [44]. A space-time diagram plots processes along one axis and time along the other, where messages are shown as lines between the relevant (process, time) pairs. In this particular case, the displayed communication events are those that were recorded in the replay data during the initial program run. The outcome is a graph of communication patterns that is used to assist the programmer in appropriately placing causal distributed breakpoints.

Only event indices are recorded during the collection of replay data. In particular, the time that each event occurs is not recorded. The relative placement of the events in the space-time diagram is thus based solely on the "happened before" relation, where the "time" of an event is based on its relative time. The time between a send event and its corresponding receive event is one time unit, and concurrent events are plotted at the same position in time. An alternative technique for the collection of replay data would be to record a timestamp with each send or receive event, such that the space-time diagram could more accurately represent the relative occurrence of the events. However, the disadvantage of capturing timestamps during the initial program run is that the level of monitoring intrusion is increased, likewise increasing the possibility of a probe effect, and thus this technique is not used.

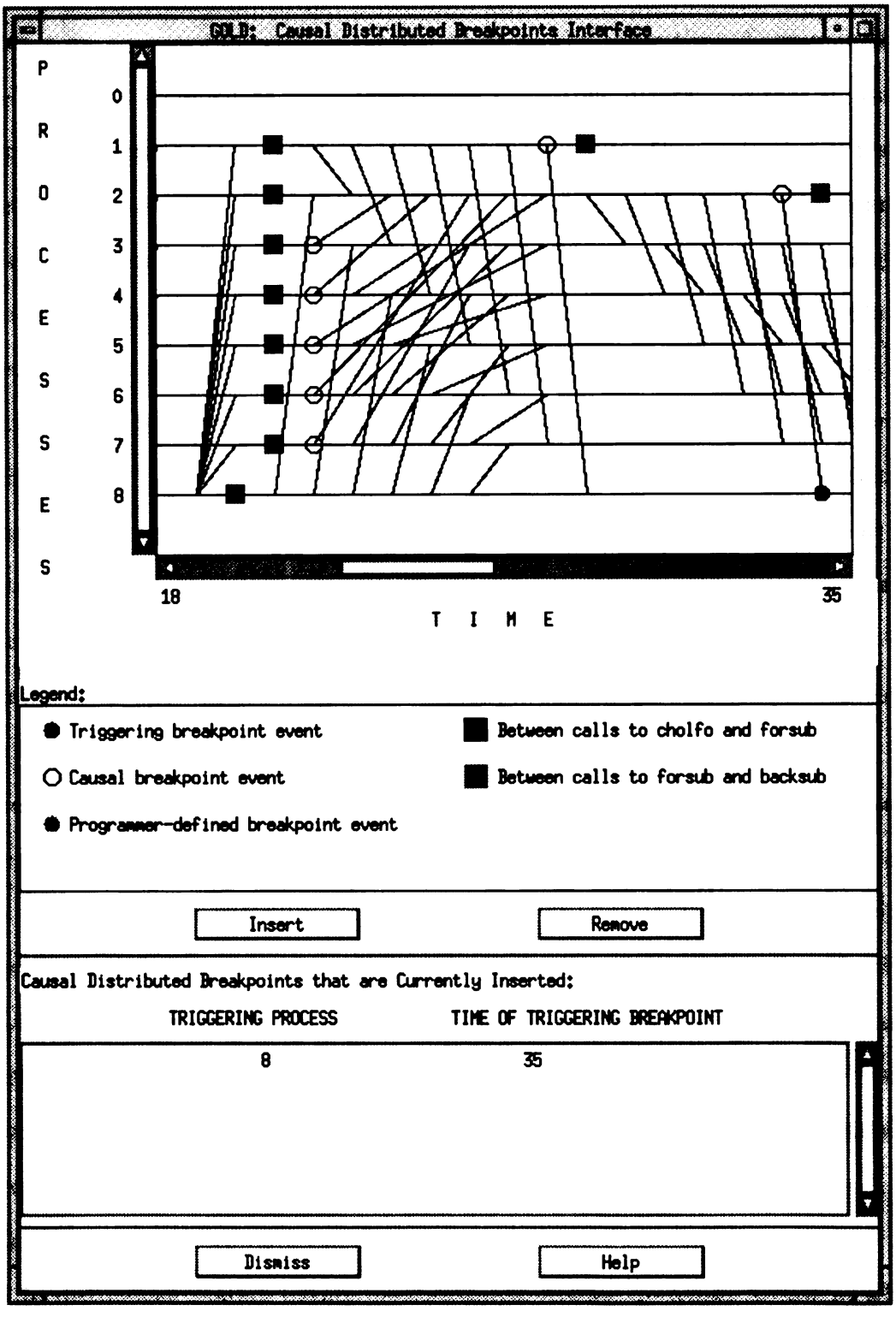

Figure 4.3. Interface for insertion of causal distributed breakpoints

A potential location for a causal distributed breakpoint may be evaluated by selecting an appropriate send or receive event in the space-time diagram. Scroll bars have been included on both axes of the space-time diagram so that it can be completely examined for potential locations of breakpoints. Upon selection of a send or receive event, a dark-colored, filled circle is placed at the position of the event to mark it as the location of the triggering breakpoint. The algorithm for computing the causal distributed breakpoint is then executed, where the result of this computation is depicted by placing an empty circle along each of the other process lines at the location of the last event that had a causal effect on the execution of the triggering process. (The lighter-colored, filled circles represent the location of programmer-defined breakpoint events, which will be discussed below.) For example, in Figure 4.3, the triggering breakpoint is at a receive event for process 8, and empty circles for depicting the causal breakpoint events have been drawn accordingly along the other process lines. (The causal breakpoint event for process 0 is located at an event earlier than the displayed time frame.)

Setting causal distributed breakpoints based solely on a visualization of message-passing events may be difficult, as it depends on the ability to locate relevant message-passing patterns. In order to simplify the task, we offer an option for generating programmer-defined *marker* events when the replay data is captured. Specifically, a GOLD-supported system call may be added to the source code in appropriate places, where this system call includes a textual description of the marker event as one of its parameters. During the initial program run, the corresponding marker events are collected along with the replay data. When the space-time diagram is rendered, the marker events are depicted using shaded boxes, where different colors shadings are used to distinguish marker events. For example, Figure 4.3 shows a case in which two marker events were used, highlighting when the program is between the Cholesky factorization and forward substitution phases, and between the forward substitution

and backward substitution phases. A legend of the different marker events is depicted below the space-time diagram, where the textual descriptions of the marker events are used to label the corresponding shaded boxes. The marker events may also be selected as the triggering breakpoint event for a causal distributed breakpoint.

The selection of an event in the space-time diagram gives the programmer an opportunity to evaluate the causal dependencies for a triggering breakpoint, but does not actually set the causal distributed breakpoint, such that it would stop the program during an execution. In order to complete the insertion of the breakpoint, the programmer may select the **Insert** button. At that time, the list of inserted breakpoints will be updated. For example, in the case of Figure 4.3, the causal distributed breakpoint depicted in the space-time diagram has already been inserted, as indicated by the entry in the list (lower window of Figure 4.3). When the **Insert** button is selected, the COM also records the event number at which each process should stop in order to satisfy the causal distributed breakpoint. As the program is running, each LOM maintains a counter that tallies the number of message-passing and marker events that have occurred for the particular PVM process, and will stop the execution of the process when the value of the counter matches the event count for a distributed breakpoint. If the programmer decides an inserted breakpoint is no longer necessary, then the appropriate entry in the list of inserted breakpoints may be selected and the **Remove** option chosen.

One of the special cases addressed by GOLD is when causal distributed breakpoints overlap. During an execution where multiple breakpoints are set, the program stops at the breakpoints according to the relative order of the triggering breakpoints, where this order is computed according to the "happened before" relation for events recorded in the replay data. In a case where causal distributed breakpoints overlap, however, particular processes may not be able to stop at the appropriate location for a later breakpoint after the stopping location of an initial breakpoint has

been reached. For example, consider the two causal distributed breakpoints shown in Figure 4.4, where the triggering breakpoints are located at points "X" and "O". If the execution initially stops at the causal distributed breakpoint triggered at point "X" (i.e., process P stops at point "X", process Q stops after executing event e, and process R stops after executing event a), process P has already passed its stopping location for the second breakpoint triggered at point "O" (i.e., event c), and so the requirements of that causal distributed breakpoint cannot be satisfied. A similar conflict occurs for process R if the program initially stops at the breakpoint triggered at point "O". In order to avoid this type of conflict, GOLD checks for overlapping causal distributed breakpoints when the **Insert** button is selected, and prevents the insertion of a breakpoint that it determines to be overlapping with a breakpoint that was inserted earlier.

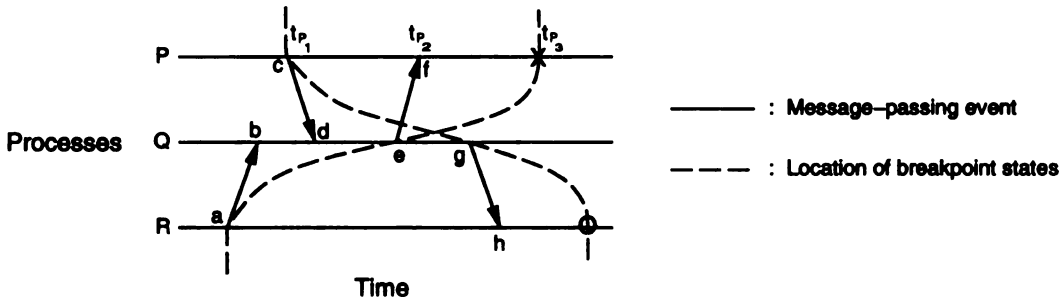

Figure 4.4. Overlapping causal distributed breakpoints

GOLD provides the programmer with an option for stopping the execution of a process at an event time different from that computed by the causal distributed breakpoint algorithm.[1] This option is useful for stopping the program at non-causal

[1]The interface for inserting causal distributed breakpoint supports this option by using the different mouse buttons for separate purposes. Specifically, the left mouse button is used to select the location of a triggering breakpoint, and the middle mouse button is used to mark an alternative breakpoint event.

distributed breakpoints. For example, Figure 4.5 illustrates a case where processes 1–8 are each stopped after the first message-passing event (initiated by process 8) in the backward substitution phase. (The causal breakpoint event for process 0 is again located at an event earlier than the displayed time frame.) The location of programmer-defined breakpoint events are marked by lighter-colored, filled circles.

4.2 Local Breakpoints

An operation for inserting a local breakpoint is provided by the LOI, accessible from the **breakpoint** menu. The interface that is activated for inserting a local breakpoint consists of three main regions, as shown in Figure 4.6. The top region contains a call graph that the programmer uses to select the appropriate function in which the breakpoint should be placed. When the function is selected, the source code for that function is displayed in the middle window. The programmer may set a breakpoint by selecting the appropriate location in the source code listing, and then choosing the **Insert** button. The bottom region of the interface is a list of the local breakpoints that have been inserted for the process. For example, in the case shown in Figure 4.6, a local breakpoint has been inserted for process 0 (i.e., the host process) immediately after the call to **pvm_spawn**. An option for removing a breakpoint is also provided.

A particular case in which setting local breakpoints may be useful is when a process has stopped at a distributed breakpoint event, but it is then desirable to examine the sequence of states immediately following that event. In that case, local breakpoints may be inserted for controlling the execution of the process. The process may then be run separately from the other processes in the system, using the execution options accessible from the LOI. The process may be run on an individual basis in this manner until it blocks, waiting to receive a message from another process, or until completion if no other receive events occur.

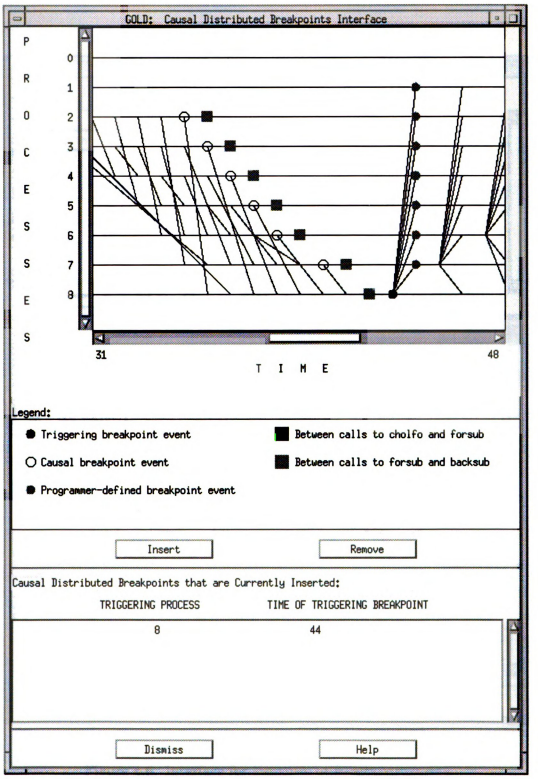

Figure 4.5. Example of a non-causal distributed breakpoint

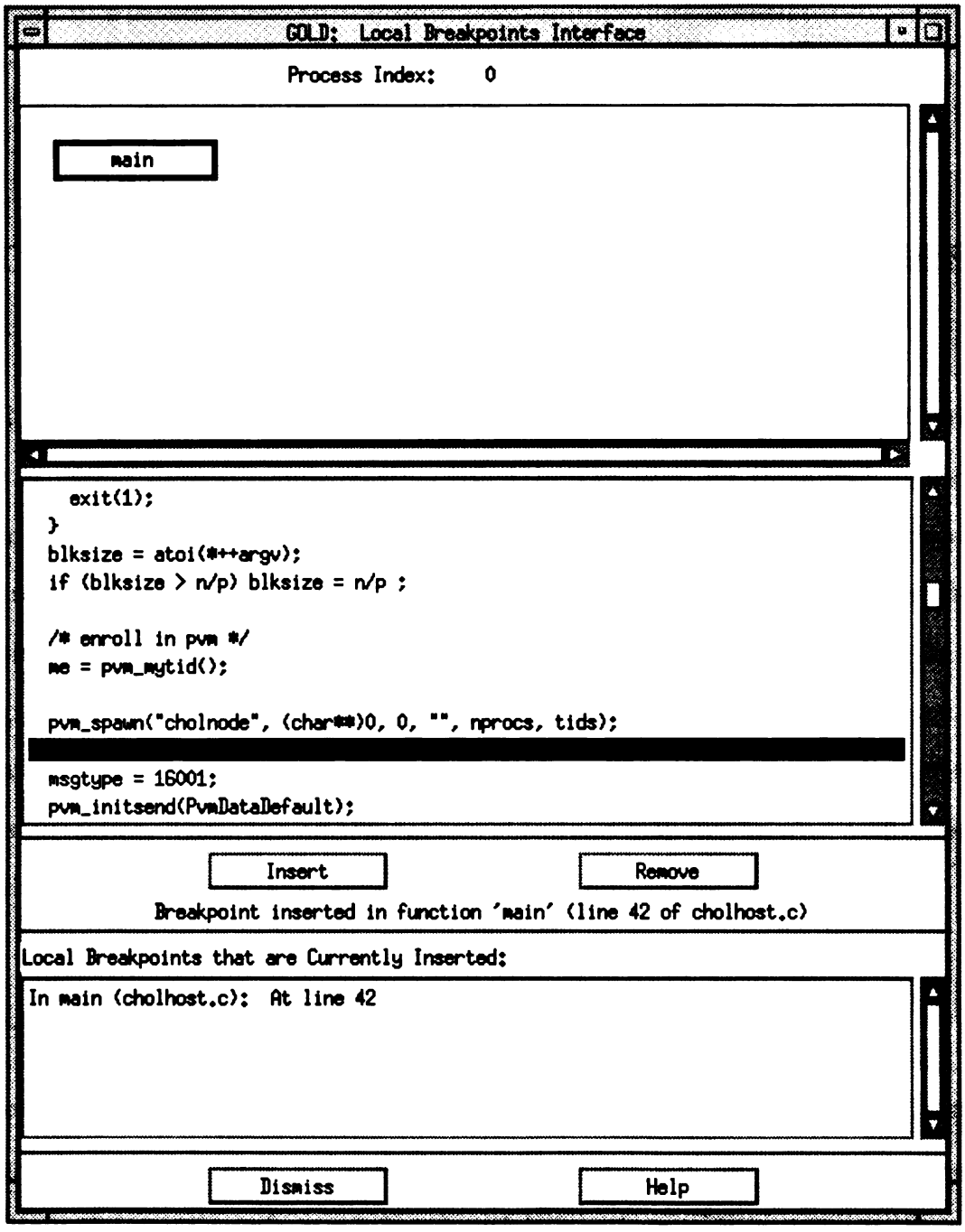

Figure 4.6. Interface for insertion of local breakpoints

CHAPTER 5

Scalable Visualization Techniques

We have developed several techniques for improving the scalability of visualizations that represent process communication [10, 11]. The basis for these techniques are filtering mechanisms, where the programmer is provided with a simple interface for controlling the information that is depicted. Two cases in particular have been considered. First, we developed a set of visualization-based features to handle programs that make use of a large number of processes. Second, we developed an approach for visualizing program executions that incur a large degree of process communication. These two orthogonal sets of features may also be used cooperatively for the case of programs that grow in size in both dimensions.

5.1 Handling Large Numbers of Processes

Space-time diagrams are used by GOLD to depict process communication during a program execution. Previously, we described how a space-time diagram is used to facilitate the insertion of causal distributed breakpoints. The space-time diagram that depicts communication during the program execution is similar, except that the time of an event is based on actual time, in contrast to a relative time. Each LOM sends a message to the COM whenever a send or receive event occurs. The COM

orders these messages as they are received, waiting to record receive events until the corresponding send events arrive, and ensuring that the events at each node are recorded in the order that they occur. When an event is able to be recorded (i.e., all events that "happened before" the event are already recorded), a timestamp is recorded with the event, where this timestamp is used as the time of the event when constructing the space-time diagram. This timestamping mechanism ensures accurate event ordering, which is critical for debugging.

Programs that use large numbers of processes particularly influence the effectiveness of a space-time diagram, as it becomes difficult to detect patterns and inconsistencies in a space-time diagram when the number of elements plotted along the process axis grows large. In order to handle this shortcoming of space-time diagrams, we take advantage of a strategy used by many programmers when debugging. That is, in most cases, the programmer has a general idea of the location of the source of an error, which means that typically the behavior of only a subset of the relevant processes is of interest during debugging. When this case occurs, the "Process Filter" window shown in Figure 5.1 may be activated for filtering what processes are depicted

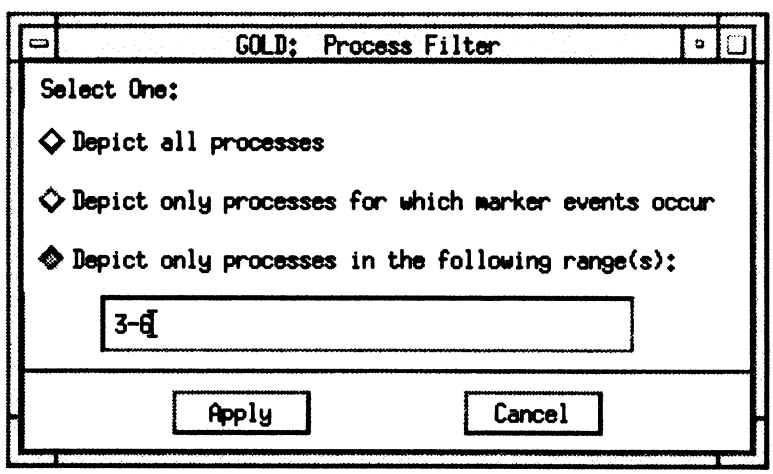

Figure 5.1. Popup used to filter the processes

along the process axis. One of the selections shown in the window is an option to specify one or more ranges of processes that should be depicted along the process axis. For the case shown in Figure 5.1, processes 3–6 were selected as the range of processes to be depicted, where the scenario was for an execution of the Cholesky factorization program that used 16 worker processes. The unfiltered version of the space-time diagram is shown in Figure 5.2, while the space-time diagram that has had filtering applied is shown in Figure 5.3. (The depicted time frame is the same in both figures.) Clearly, the visualization shown in Figure 5.2 is too congested to be useful, where, in contrast, the programmer may readily locate patterns of communication in Figure 5.3.

An additional option shown in the popup in Figure 5.1 is one to depict only the processes for which a programmer-defined marker event occurs. This option is useful when only a subset of processes participate in some phase of the program execution, where a marker event has been added to record participation in the phase. As in the above case, the number of processes along the process axis of the space-time diagram may be significantly reduced by using this option.

5.2 Handling a Large Degree of Process Communication

Similar to the case above, the effectiveness of space-time diagrams is affected when a program incurs a large degree of process communication. In this case, the detection of patterns and inconsistencies is complicated since the space-time diagram becomes "cluttered" with a large quantity of activity. Filtering options are again useful for handling this situation, where, for this case, the filters are designed to eliminate the depiction of all but a subset of messages. In particular, a single process is selected as the process of interest, and a visualization referred to as a "Single Point-

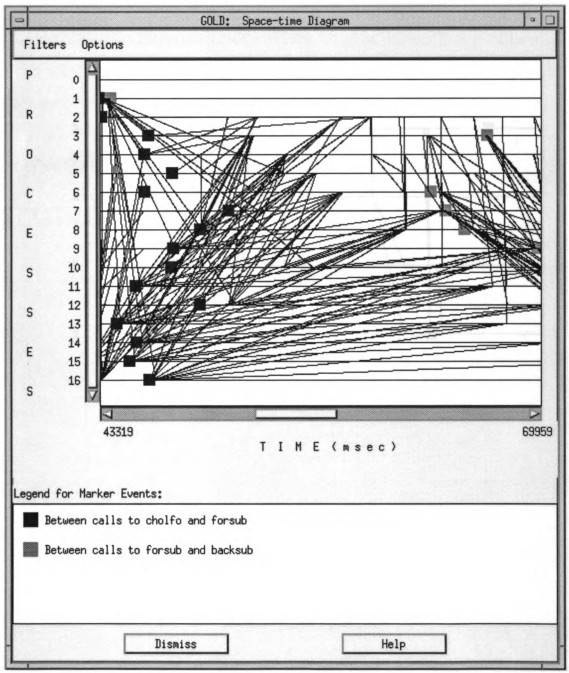

Figure 5.2. Space-time diagram before process-filtering has been applied

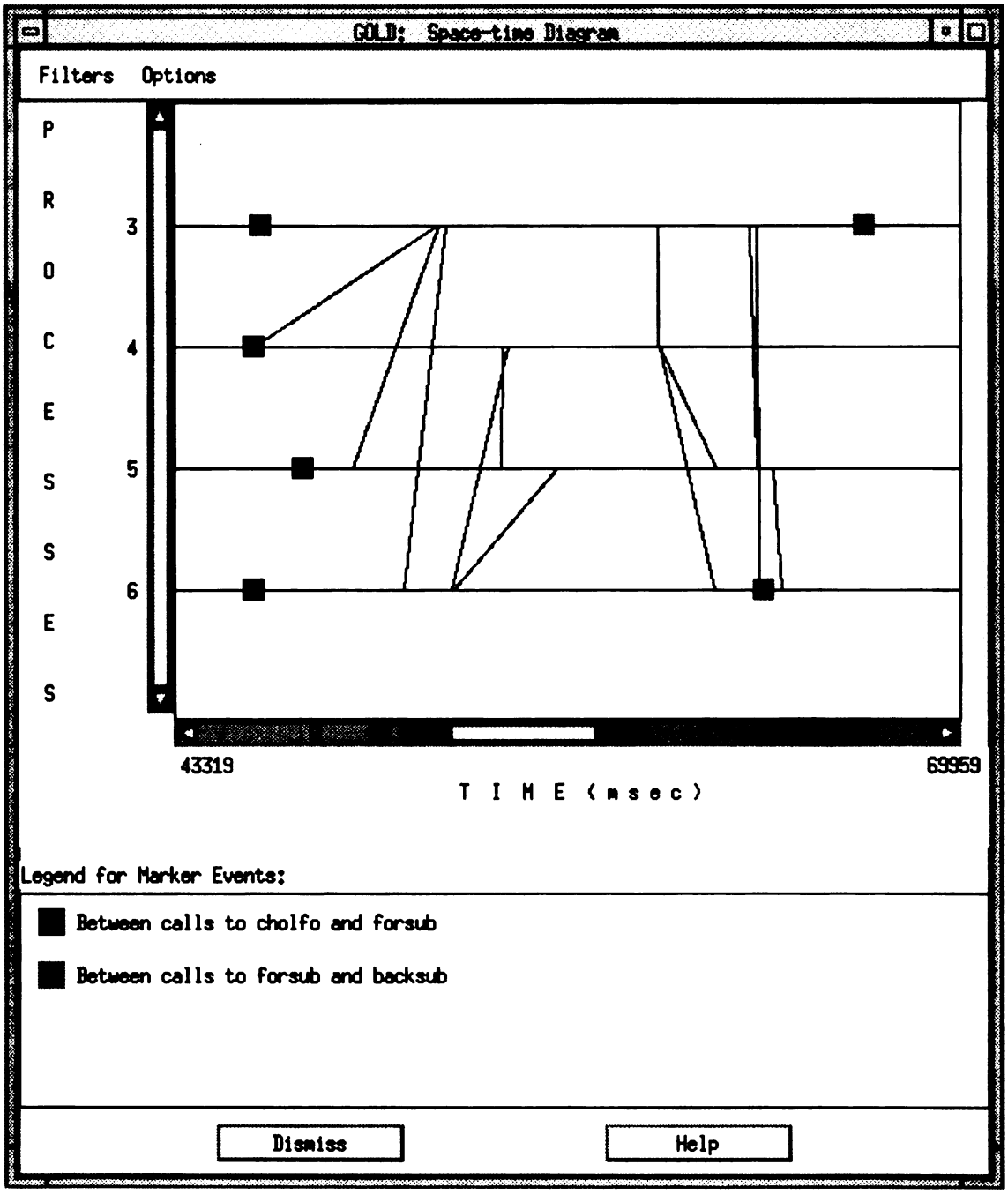

Figure 5.3. Space-time diagram after process-filtering has been applied

of-View (SPV) communication graph" is activated to depict only the messages that are either received at or sent from that particular process. An example of this type of visualization is shown in Figure 5.4, where process 10 has been selected as the process of interest. (The event data for Figure 5.4 is the same as that used to render the visualizations shown in Figures 5.2 and 5.3, where approximately the same time frame is depicted.)

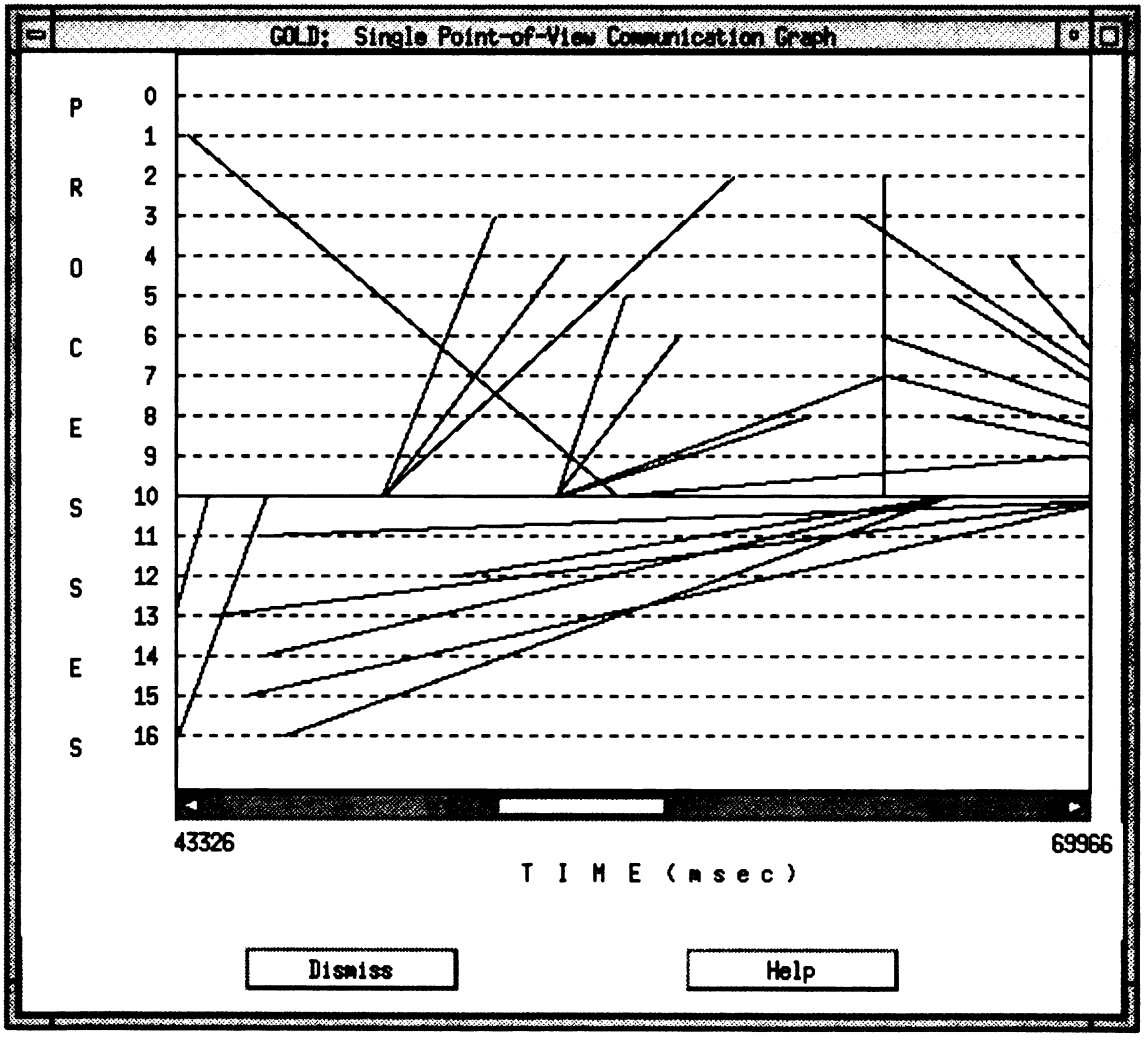

Figure 5.4. Example of a single point-of-view communication graph

One of the features of an SPV graph is the simple approach for activating it. An SPV graph may initially be activated by selecting the appropriate process axis in a space-time diagram with the middle mouse button. In order to change the process of interest, the applicable process axis may be selected in the SPV graph, and the SPV graph will then be updated accordingly. This visualization-based approach for filtering messages provides the programmer with a simple, efficient mechanism for examining process communication when a large degree of communication occurs.

Additional options are offered for changing the scale of either the process or time axes. These options are useful for either the case of a large number of processes or a large degree of process communication. The programmer may use these options to "zoom in" towards a particular region of a space-time diagram. For example, Figure 5.5 illustrates a case in which the time axis scale for the space-time diagram shown in Figure 5.3 was modified. By changing the time axis scale, the message-passing events immediately before the backward substitution phase could be examined more closely.

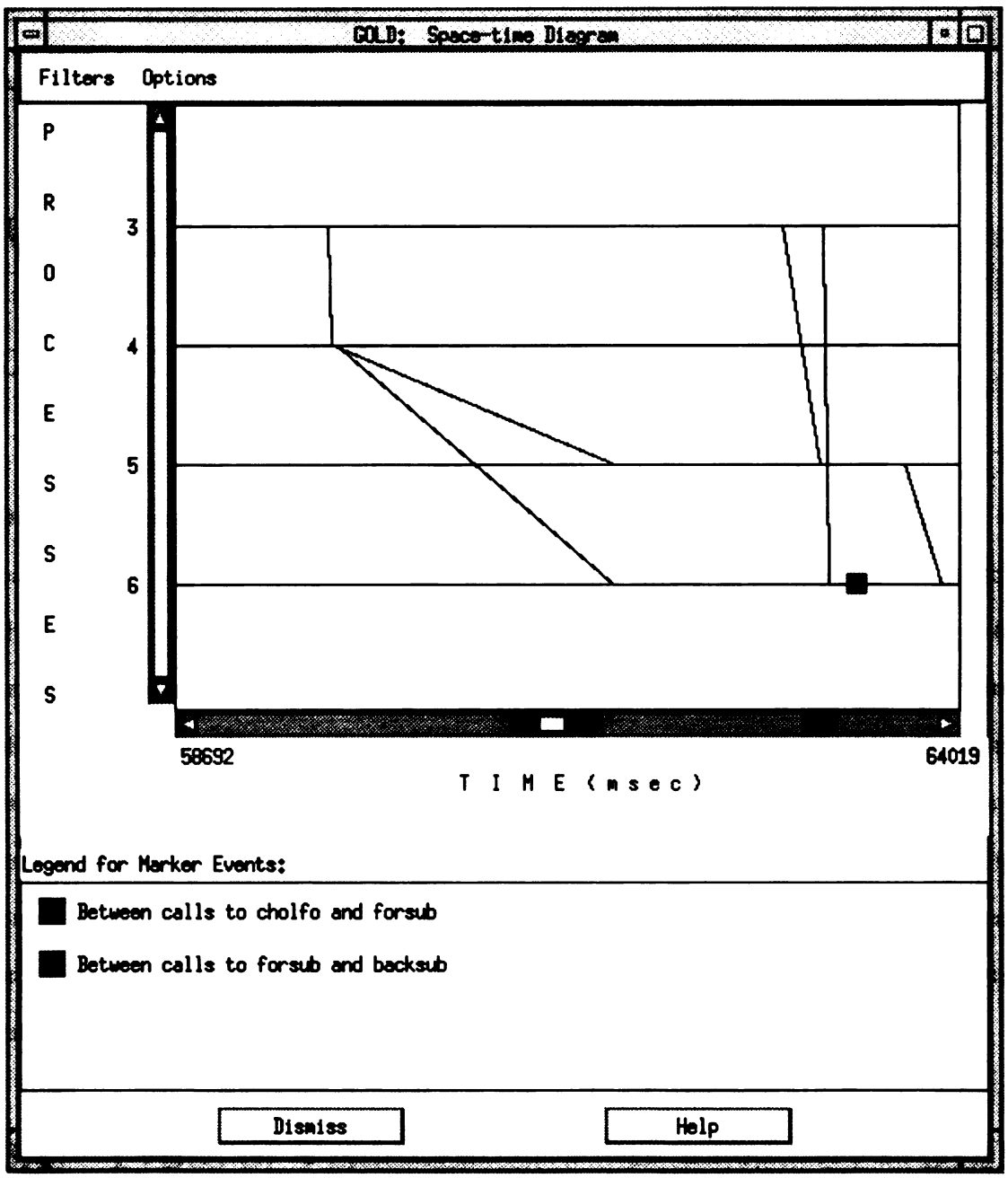

Figure 5.5. Modifying the time axis scale to "zoom in"

CHAPTER 6

Visualization-based Examination of Program States

Distinct types of visualizations are used to represent different characteristics of a parallel program execution. For example, space-time diagrams are designed to depict the communication between processes across time, whereas other visualizations are targeted towards representing the execution in terms of the source code structure [45, 46]. These different types of visualizations each have individual advantages, and so may be applied for different purposes during debugging. In this chapter, we present guidelines for coordinating the use of the different visualizations for examining program states [9, 11, 25]. We first introduce the different types of visualizations supported by GOLD, and we then discuss how they are applied in a top-down, integrated manner.

6.1 Types of Visualizations Supported by GOLD

The initial execution of a program using GOLD is dedicated to the collection of replay data, but the programmer may then run the program any additional number of times in order to examine particular states. The different visualizations used for

examining program states may be categorized into four types, described as follows. First, communication graphs are used to examine communication patterns during a program execution. Both space-time diagrams and SPV graphs are supported for depicting message-passing events as they occur. In the case of space-time diagrams, any programmer-defined marker events are also depicted.

Second, a *call graph* is used to depict the current location in the execution of a particular process. The function hierarchy (i.e., a tree structure) is used to graphically represent the location, where the path from the root node to the node representing the current function is highlighted. An example of a call graph is shown in Figure 6.1, where the current location in the execution is within the function `forsub`.

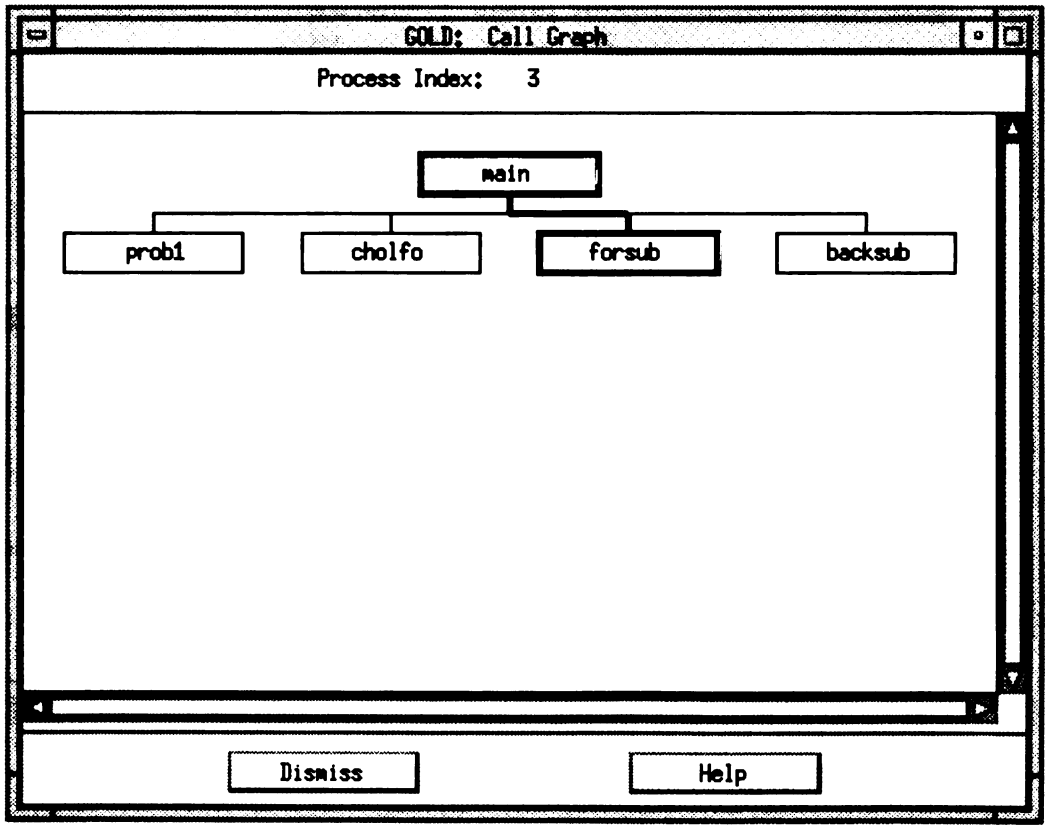

Figure 6.1. Example of a call graph

A third type of visualization supported by GOLD is a *source code listing.* This type of visualization is also used to depict the current location in the execution of a particular process. In this case, however, the location is displayed via a listing of the relevant source code, where the current location is highlighted. Although source code listings have a textual format, as opposed to a graphical format, they are handled as a visualization since their use is consistent with the use of the graphically-based visualizations. An example of a source code listing is illustrated in Figure 6.2, where the current location within the function **forsub** is shown.

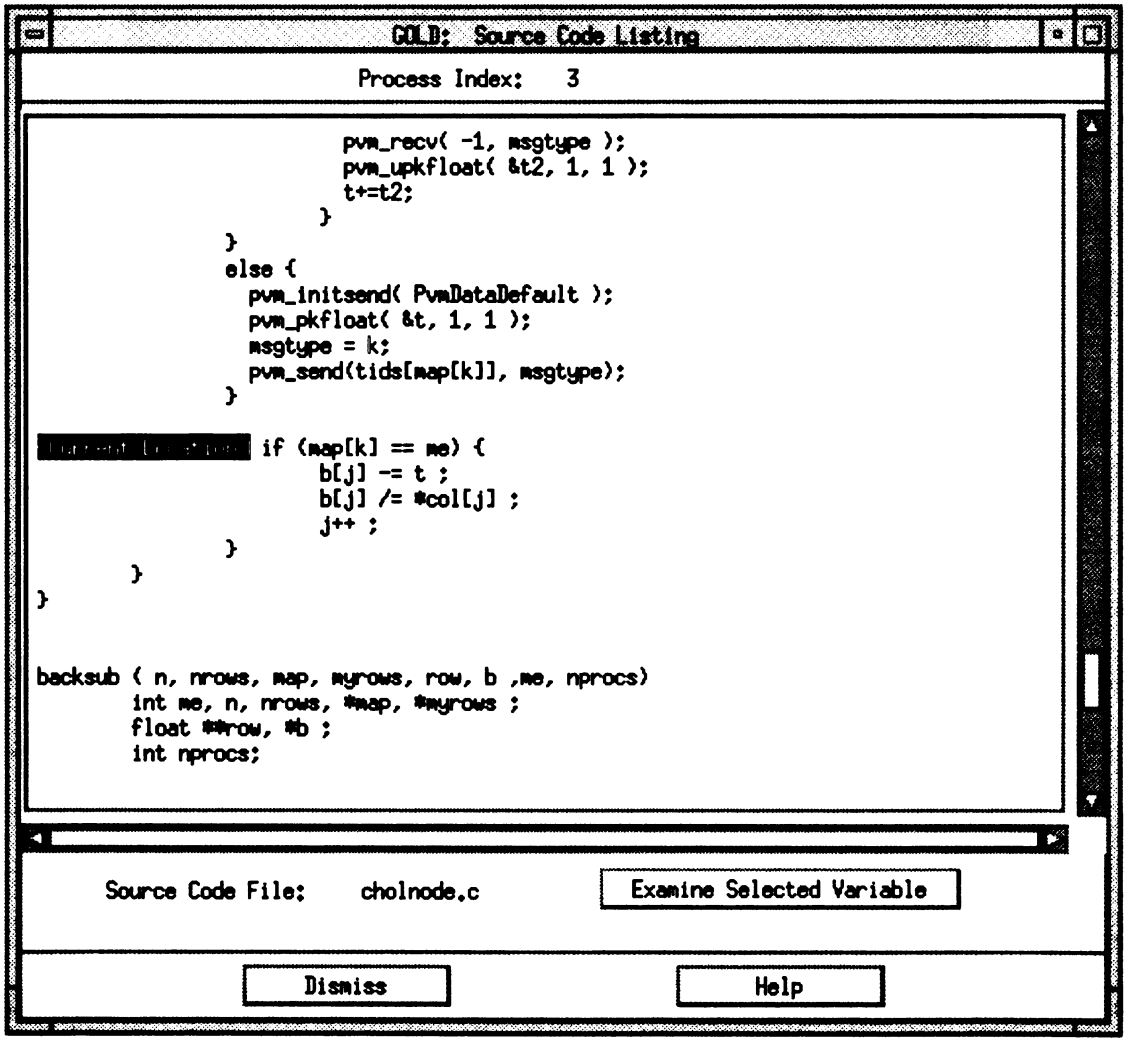

Figure 6.2. Example of a source code listing

Finally, *data visualizations* are used to inspect the contents of data structures. They are especially useful for the case in which the other types of visualizations have been used to narrow the source of a suspected error to the operations on a particular data structure. The values of one- and two-dimensional arrays are depicted graphically, while the values of other types of data structures are printed textually. For the latter case, the contents of data structures located in the same process are displayed in a common window. For example, Figure 6.3 shows a case in which the values of several of the variables from the function `forsub` are displayed.

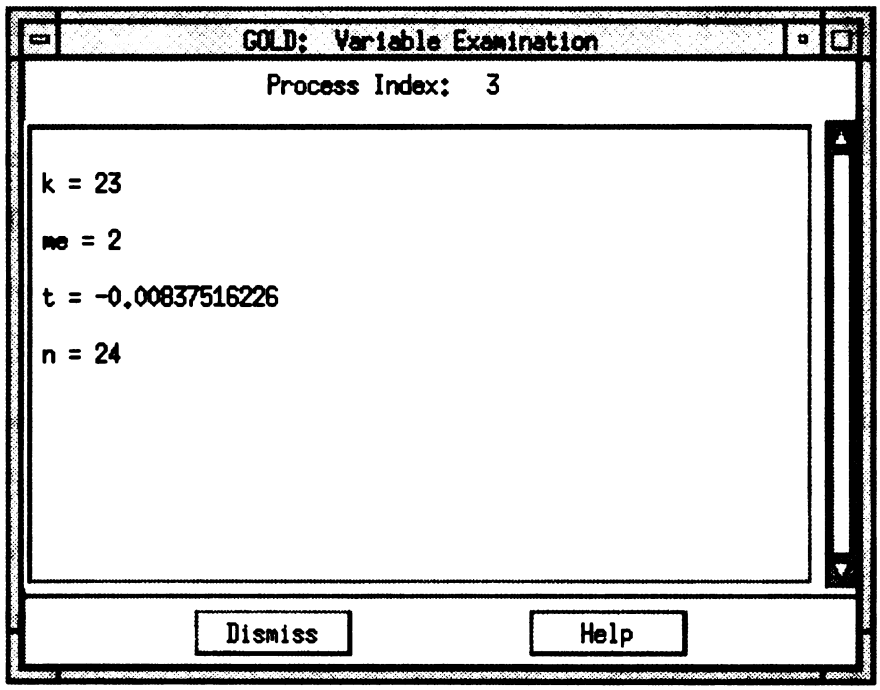

Figure 6.3. Textual display of data structures

For the depiction of one- or two-dimensional arrays, each array is displayed in a separate window. If the selected array has been declared with static dimension(s) in the current function, then the size of that array is accessible via local debugger commands. However, if the array is dynamically allocated, or passed as a pointer to

the current function, the size is not readily accessible, and so a separate interface is activated to obtain additional information from the programmer. For example, in the case of the source code listing in Figure 6.2, the variable **map** was passed as a pointer to the function **forsub**. Upon selecting that variable for examination, the "Additional Variable Information" interface shown in Figure 6.4 was activated. This interface lets the programmer indicate whether the selected variable (i.e., a pointer) points to a single element, a one-dimensional array, a two-dimensional array, or to some other data structure. If the selection for a one-dimensional array or a two-dimensional array is chosen, then additional size information must also be entered in the appropriate textfield. In the case shown in Figure 6.4, the selection for a one-dimensional array has been chosen for the variable **map**, and the size has been set equal to the variable **n**. (The variable **n** is also passed as a variable to the function **forsub**.) By entering this size information in the appropriate textfield and selecting the **Apply** button, the LOM has enough information to query the local debugger for the values of the array elements.

The one-dimensional array visualization corresponding to the variable **map** is shown in Figure 6.5. In this case, the array values are plotted as a bar chart, where the length of a bar represents its relative magnitude. The precise values of particular elements may be examined by selecting the corresponding bar (thus highlighting the bar), such that a key benefit of the array visualization is that it provides a simple format for probing the values of the array elements. In the case shown in Figure 6.5, the value of element 21 has just been probed, where the bar for element 21 is highlighted and its value is displayed.

An example of a two-dimensional array is shown in Figure 6.6. In this case, the array values for the variable **output** are plotted in a two-dimensional grid, where different color shadings represent different ranges of values. (In the particular case shown in Figure 6.6, all the array values fall in either the top or the bottom

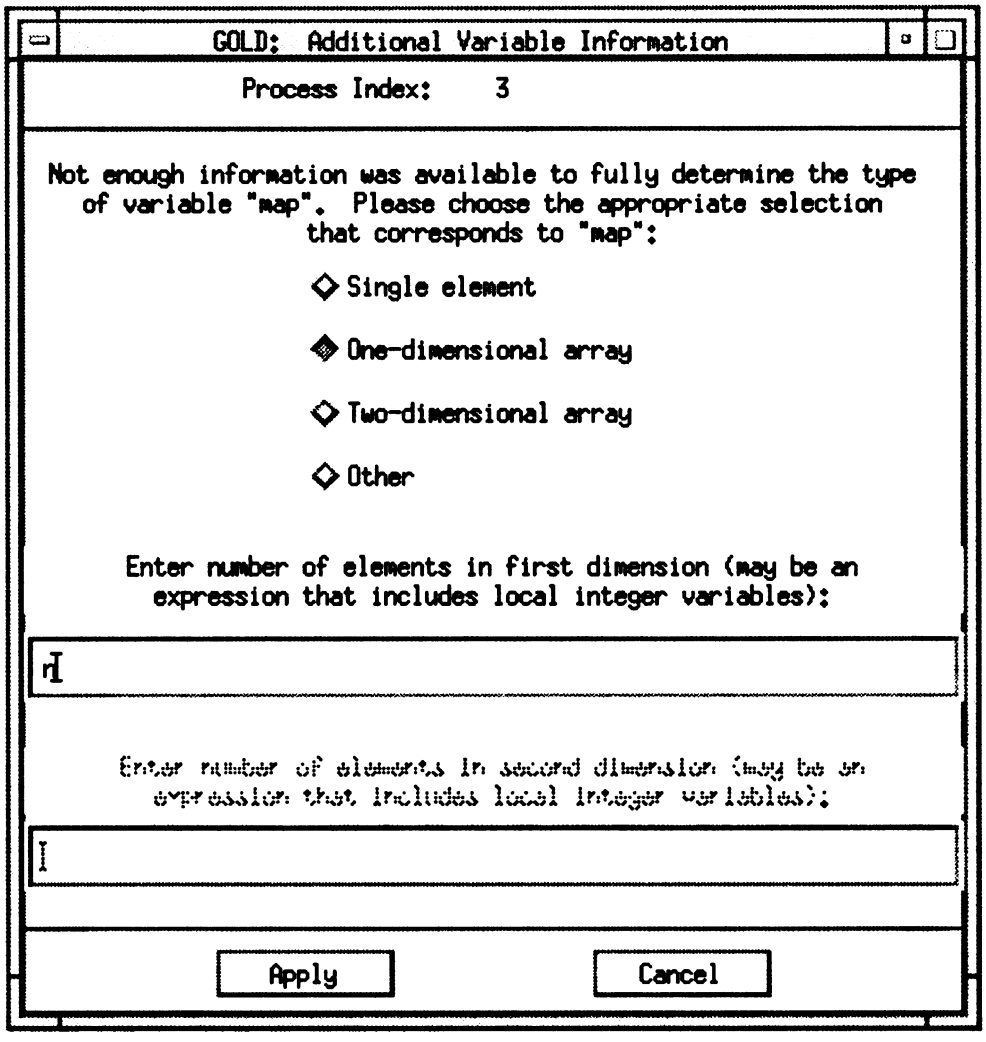

Figure 6.4. Interface for obtaining additional variable information

range of the four possibilities listed in the legend.) Similar to the case of a one-dimensional array, the precise value of a particular element may be examined by selecting the corresponding rectangle, where the rectangle is then highlighted and the corresponding value is displayed.

One of the features of the array visualizations is an option for filtering the values in order to highlight particular instances of data. Several filtering possibilities are provided, including choices for filtering values that exceed some maximum value, that fall below some minimum value, that are equal to some critical value, or that

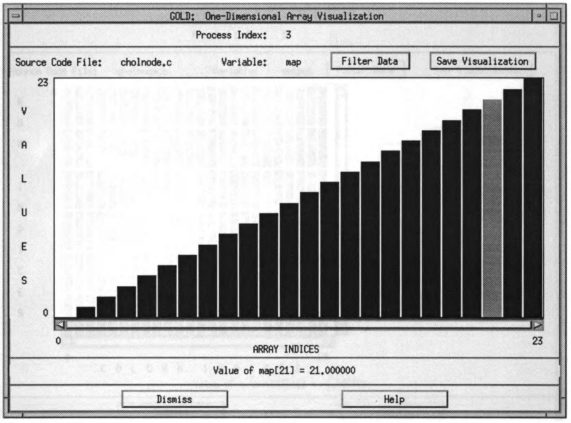

Figure 6.5. Graphical depiction of a one-dimensional array

are within some critical range. The interface for filtering data values is shown in Figure 6.7, where a filter has been selected for highlighting negative data values in the two-dimensional array visualization that was shown in Figure 6.6. The filtered version of the two-dimensional array visualization is shown in Figure 6.8.

An additional feature of the array visualizations is an option for saving the current contents of a visualization. This option is useful for examining changes in an array as the program executes. The saved version of the array visualization may be compared to an array visualization that is activated at a later point in the program execution. For example, Figure 6.9 illustrates a case in which the array visualization shown in Figure 6.8 was saved, and then compared to an array visualization for the variable `output` at a later point in the program execution.

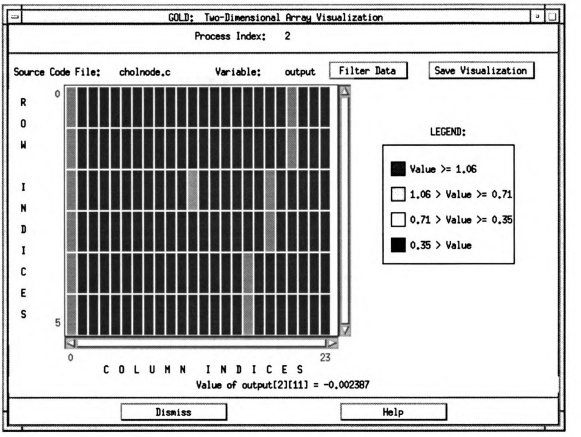

Figure 6.6. Graphical depiction of a two-dimensional array

The depiction of the space-time diagram used for state examination is handled by the COM. The LOMs notify the COM whenever a message-passing or marker event occurs, such that the space-time diagram may be updated. In contrast, the LOMs handle the rendering of the other three types of visualizations. The information that is contained in call graphs, source code listings, and data visualizations is confined to a single process, and therefore communication with the COM is unnecessary.

The communication graphs are continuously updated during program execution. When the program stops at a causal distributed breakpoint, scroll bars along the time axes of the space-time diagram and the SPV graph may be used to examine the communication events throughout the execution history. As such, the COM stores an

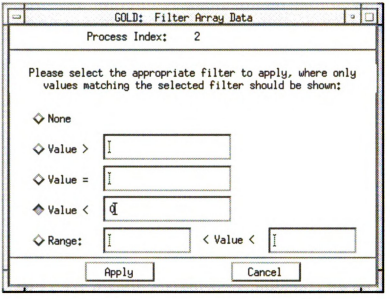

Figure 6.7. Interface for filtering data values

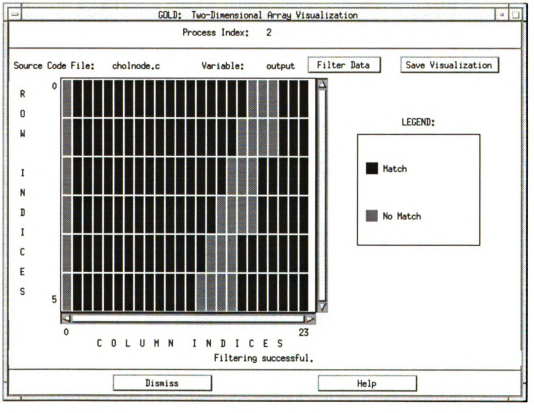

Figure 6.8. Filtered version of a two-dimensional array

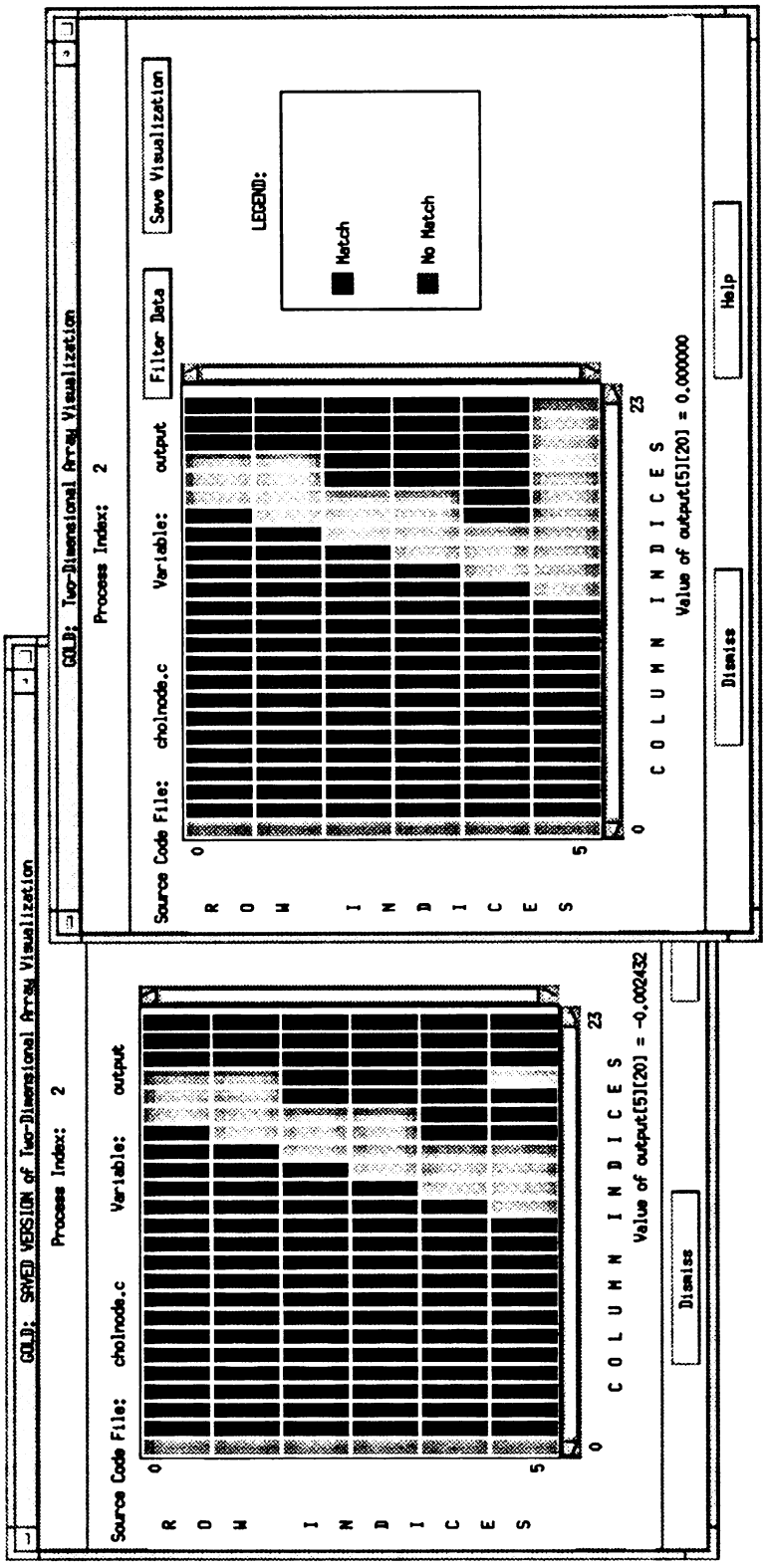

Figure 6.9. Examination of changing values for a two-dimensional array

entry for each communication event that occurs in order to enable the programmer to browse through the entire collection at any point during the execution of the program. In contrast, call graphs, source code listings, and data visualizations are only updated when the execution reaches a breakpoint (either a causal distributed breakpoint or a local breakpoint). These visualizations reflect the current state of the program, and so do not require the storage of execution data across time, yielding a savings in data storage requirements. This feature is a side-effect of using an on-line approach to visualization as opposed to a post-mortem approach. In the case of a post-mortem strategy, any data that is potentially of interest must be stored during the program execution, such that it may later be visualized after execution is complete.

6.2 Top-down Coordination of Visualizations

The different types of visualizations are applied in a top-down manner for examining a program state. That is, the communication graphs and status grid are used as the top-level visualizations, where the application of the other types of visualizations is based on the selection of process axes or events from a communication graph. Visualizations may be applied according to the hierarchy shown in Figure 6.10. Upon selection of a process axis in a communication graph, the programmer is given the options listed in the "Process Selection Window" shown in Figure 6.11, where the program is currently stopped at the causal distributed breakpoint that was illustrated in Figure 4.5. In particular, options are provided for activating an SPV graph, an LOI, an LOI and a call graph, or an LOI and a source code listing for the selected process. The advantage of activating the LOI at the same time as a call graph or a source code listing is that the relevant process may be executed independently of the other processes for examining a particular sequence of states. Alternatively, a call graph or source code listing may be activated as an option from the LOI.

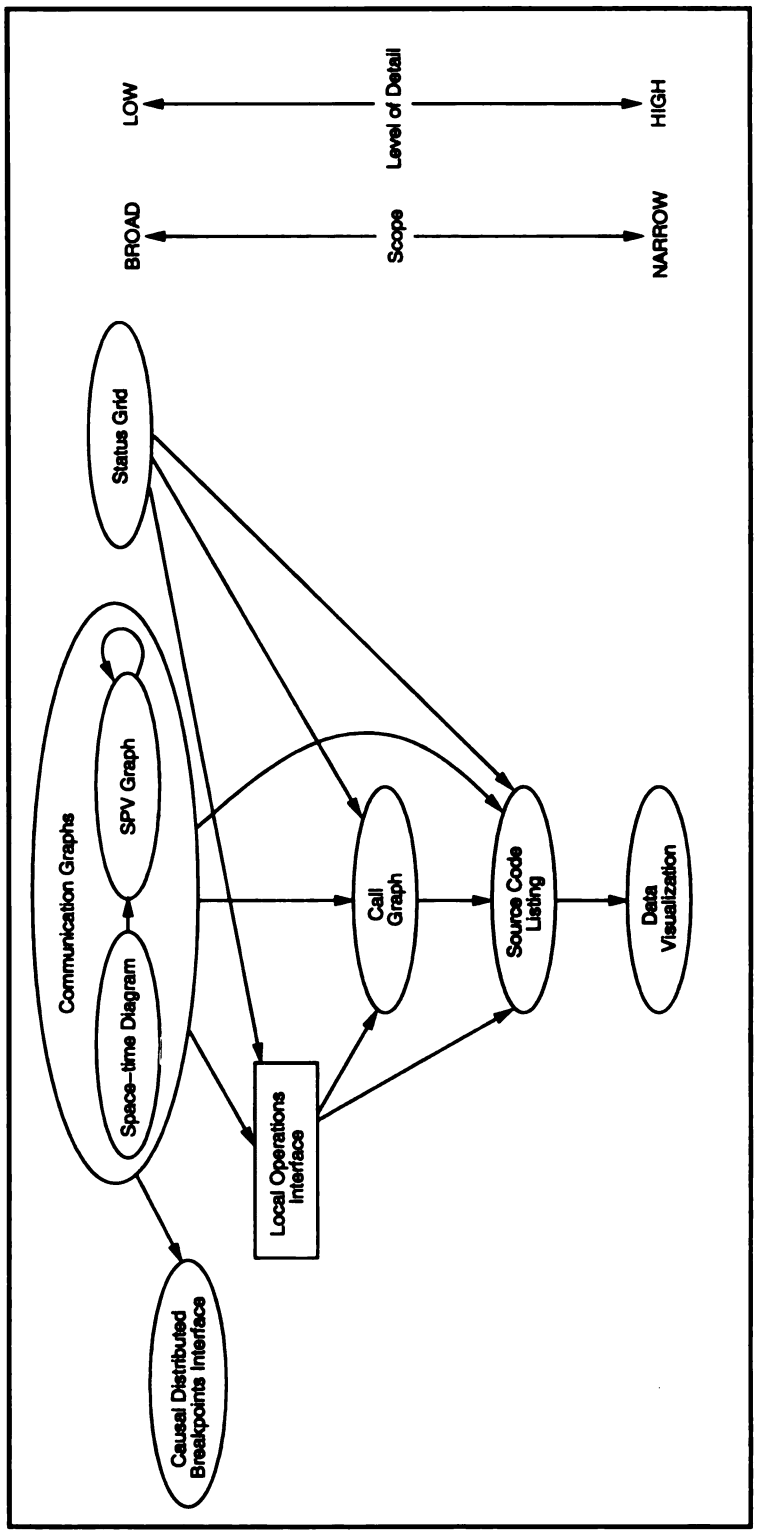

Figure 6.10. Visualization hierarchy for examining a breakpoint state

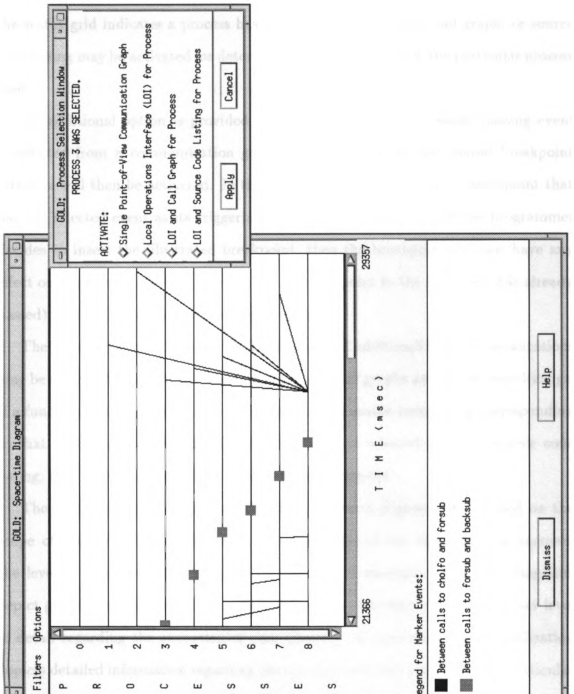

Figure 6.11. Options for examining a breakpoint state from a space-time diagram

The "Process Selection Window" may also be activated by selecting the appropriate process from the status grid. This option may be useful, for example, if the status grid indicates a process has died. The relevant LOI, call graph, or source code listing may be activated for determining the point at which the particular process died.

An additional option is provided when an endpoint of a message-passing event is selected from a communication graph, where the causal distributed breakpoint interface will then be activated. In that case, the causal distributed breakpoint that has the selected event as its triggering breakpoint is illustrated. If the programmer decides to insert the illustrated breakpoint, then the breakpoint will not have any effect on the current execution (since the relevant point in the execution has already passed), but will apply for future re-executions.

The hierarchy shown in Figure 6.10 indicates that additional levels of visualizations may be activated based on the selection of items in call graphs and source code listings. If a function node is selected in a call graph, then a source code listing corresponding to that function is displayed. If a variable name is selected from the source code listing, then an appropriate data visualization is depicted.

The hierarchy for applying visualizations shown in Figure 6.10 is based on the scope of the different visualizations. As the scope of the visualizations narrows, the level of detail revealed by them increases. For example, space-time diagrams depict global program activity (i.e., a broad scope), but they reveal only a low level of detail regarding the execution of each process. In contrast, a data visualization depicts detailed information regarding the contents of a data structure for a particular process, but its scope is consequently limited to that data structure.

Therefore, we coordinate the use of numerous types of visualizations in order to obtain a top-down approach for the examination of a program state, where the examination initially proceeds from a broad scope, but where the scope narrows as

the potential source of an error is investigated. Specifically, a communication graph is initially used to depict the overall message-passing activity in the system, where increasingly greater detail is revealed by examining lower-level visualizations, such as call graphs, source code listings, and data visualizations, all of which can be activated at the local process level. By using this approach, the programmer may systematically converge towards the source of an error.

CHAPTER 7

The Modeling of Expected Behavior in Program Visualizations

This chapter discusses a method for using the LOTOS specification of a parallel program to model expected behavior in visualizations of the program's execution [8, 26]. For the case of visualizing communication between processing elements[1], each LOTOS process represents a phase in the computation, where we overlap a graphical representation of the phases onto a space-time diagram. For the case in which a program is stopped at a breakpoint, we have developed techniques for visualizing the current location in terms of the formal specification, either via a visualization of the LOTOS process hierarchy or via a specification listing. This overall methodology facilitates the task of debugging a parallel program by supporting graphical comparisons between the actual behavior of the program and its expected behavior. Prior to the development of GOLD, we developed a prototype debugging environment, PANORAMA, to implement the expected behavior methodology. An

[1]Throughout this chapter, "processing element" refers to an operating system process, while "process" refers to a LOTOS process.

overview of PANORAMA is presented in Section 7.1. Section 7.2 discusses how PANORAMA supports the collection of data, and Section 7.3 discusses how it supports the modeling of expected behavior in program visualizations. Section 7.4 considers a practical example where we demonstrate how PANORAMA can be used to detect message-passing errors. Section 7.5 discusses the integration of the expected behavior modeling techniques into the GOLD debugging environment, including improvements and extensions to the original methodology.

7.1 Overview of Panorama

PANORAMA is a tool for visualizing the execution of a message-passing parallel program within a model of the program's expected behavior. There are two main steps required to visualize program execution using PANORAMA: data collection and graphical depiction. Below, we present an overview of how these two steps may be used to generate program visualizations, while the details regarding each of the steps are discussed in turn in the subsequent subsections.

The general execution flow for PANORAMA is shown in Figure 7.1. PANORAMA incorporates a post-mortem visualization strategy, where trace data of important events is collected during program execution, while the graphical depiction of the data takes place off-line after execution is complete. Post-mortem approaches such as this one have been widely applied by visualization tools [24, 47, 48, 49, 50].

The first step towards visualizing program execution is to collect the appropriate data for rendering the visualizations. Both the LOTOS specification and the source code of the program are required as input for accomplishing this task. The specification is analyzed during the data collection step to derive a model of the expected behavior for the program, where this model is recorded in the form of expected behavior data. This expected behavior model is not only used in the

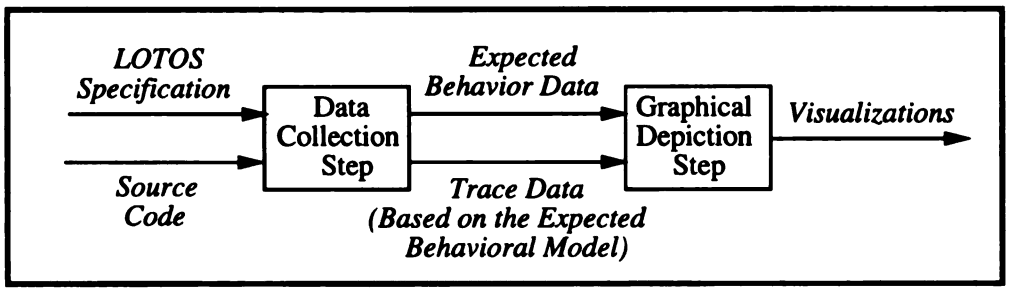

Figure 7.1. General execution flow of PANORAMA

graphical depiction step, but it also guides the instrumentation of relevant events in the source code. The instrumented version of the source code is then compiled and executed, where the instrumentation generates trace data during the program execution.

Both the expected behavior data and the trace data generated by the data collection step are used as input to the graphical depiction step. The graphical depiction step uses the two types of data to render a visualization of the program execution that models the expected behavior of the program. Specifically, a mapping is maintained between the expected behavior model and the trace data, thus providing a means for visualizing the trace data in terms of high-level events from the LOTOS specification. Such a visualization strategy is useful for debugging communication errors, which we illustrate by example in Section 7.4.

7.2 Data Collection Step

The collection of data is the first step towards visualizing program execution using PANORAMA. In this subsection, we discuss the collection of the two types of data used by PANORAMA's graphical depiction step: the expected behavior data and the trace data. In particular, we discuss how these two types of data are generated using

the LOTOS specification and source code of the program as input.

7.2.1 Collection of Expected Behavior Data

A LOTOS specification is an abstraction of a program, from which a model of the program's expected behavior may be derived. In order to facilitate the task of debugging message-passing errors, PANORAMA's expected behavior model focuses on capturing the occurrences of the message-passing events in a program. By concentrating on the aspects of a specification related to message-passing events, the expected behavior model forms an abbreviated representation of the specification, while retaining a level of abstraction in the specification.

PANORAMA generates the expected behavior model by calculating the hierarchy of processes within the LOTOS specification. The hierarchy forms a tree structure, where the name of the specification is at the root of the tree. As each process is added to the tree, the message-passing actions that occur in the behavior expression of that process are recorded with it. For example, Figure 7.2 shows the subtree of processes for the specification of the host node in the number-doubling program from Section 2.1.3, where the message-passing actions that occur in the specification are listed with their corresponding processes. After finishing the calculation of the expected behavior model (i.e., the tree structure), PANORAMA stores the model, where the stored version is known as *expected behavior data.*

Recording the message-passing actions according to where they appear in the process hierarchy structure enables message-passing actions to be categorized. This approach effectively treats the message-passing actions that occur in separate process definitions as being of different types. For example, in the specification of the number-doubling program that was shown in Figure 2.6, PANORAMA categorizes the action to send a message located in the *Send_Numbers* process differently from the action to send a message located in the *Worker_Node* process, where the names of the

process definitions are used to identify these different categories of message-sending actions. Later, we will show how this approach allows us to visualize the different types of message-passing events that occur during a program execution in terms of their corresponding locations in the expected behavior model.

The strategy for generating the expected behavior model is based primarily on the nesting of process definitions, where this nesting structure is provided by default in a LOTOS specification. One aspect of LOTOS specifications that PANORAMA considers during the generation of the model is that there are no reserved names for the gates used in message-passing actions. Because LOTOS does not have reserved keywords for this case, the programmer is allowed to explicitly declare gate names, or, by default, *send* and *recv* will be used. Given the names of the gates, PANORAMA is then able to identify the message-passing actions in the behavior expression of each process definition.

7.2.2 Collection of Trace Data

In order to visualize a program's execution, *instrumentation* must be used to identify relevant events for the program to record when it is executed. The trace data

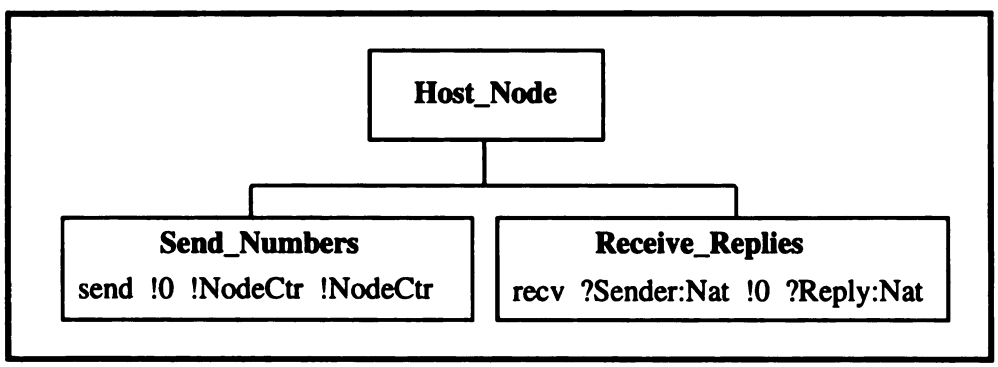

Figure 7.2. Subtree of processes for the specification of the host node in the number-doubling program

generated via instrumentation may later be graphically depicted in the form of visualizations. In the following discussion, we explain how PANORAMA supports each of the three stages of trace data collection: instrumentation, program execution (during which trace data is generated), and post-processing.

Instrumentation

PANORAMA uses a *software instrumentation* approach for the collection of trace data, where appropriate statements are added to the source code in order to generate the relevant trace data during program execution. The method for adding these data collection statements is based directly on the expected behavior model. Specifically, the method allows the programmer to map items from the expected behavior model to their corresponding instrumentation points in the source code via a graphical interface. The benefit of this approach is that the generated trace data may be graphically depicted in terms of the expected behavior model, which will be discussed in detail in Section 7.3. The programmer begins the mapping procedure by selecting an entry from a nested listing of the processes, where the selection of the particular process is based on the programmer's decision to instrument the source code events corresponding to that process. The programmer is then requested by PANORAMA to indicate instrumentation points in the source code for each of the following events:

1. Entry point of the process: The location in the source code at which point the implementation of the process' behavior expression begins
2. Exit point of the process: The location in the source code at which point the implementation of the process' behavior expression ends
3. Message-sending actions for that process: The message-sending statements in the source code that correspond to each of the message-sending actions in the behavior expression of the process
4. Message-receiving actions for that process: The message-receiving statements in the source code that correspond to each of the message-receiving actions in the behavior expression of the process

Given that a LOTOS specification is an abstract representation of the implementation, there may not be a direct correspondence between the specification and the source code. Therefore, for each of the events listed above, the programmer may indicate an approximate location in the source code if an exact location is not available. For example, broadcasts are often implemented within a parallel programming environment as a single procedure call, but LOTOS requires each message that is sent to be specified as a separate action. In this case, PANORAMA provides the programmer with a special option for indicating a many-to-one mapping.

After selecting instrumentation points for each of these events, the programmer may then start the mapping procedure again for a new process from the expected behavior model. An example of this mapping procedure is shown in Figure 7.3, where the *Receive_Replies* process from the number-doubling program is being handled. The upper window lists the events to be instrumented for the *Receive_Replies* process, while the lower window displays the source code for the node program of the number-doubling program. The button labeled "Instrument Program" is used to perform a specific event instrumentation after items have been selected from both the upper and lower windows. The button labeled "Select new process" allows a new process to be chosen for instrumentation from a nested listing of the processes.

After the programmer has performed the mapping procedure for each process of interest, PANORAMA handles the underlying details of adding the data collection statements to the source code. Currently, PANORAMA provides software instrumentation support for programs that use PVM (Parallel Virtual Machine) version 2.4 [36] message-passing primitives. Both C and Fortran coding options are handled by PANORAMA.

A significant advantage to PANORAMA's support of a software instrumentation approach is that the data collection statements are capable of generating auxiliary information as part of a traced event, where the auxiliary information is used to map

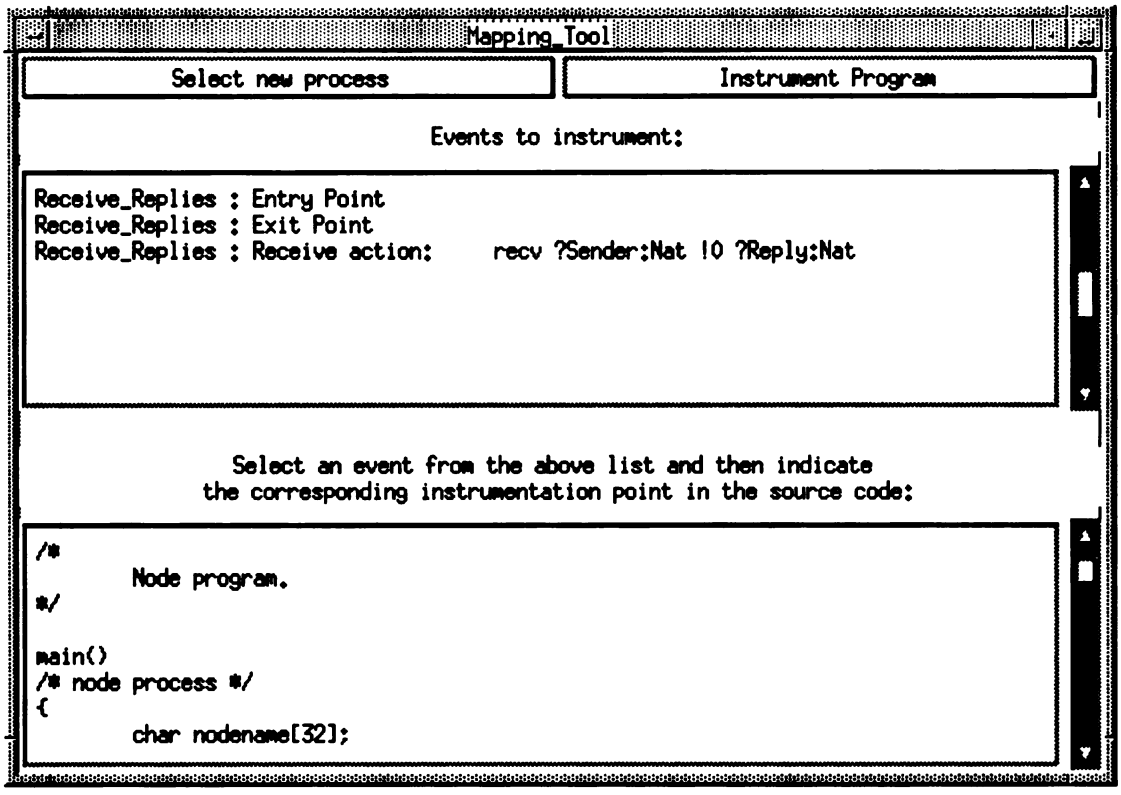

Figure 7.3. Example of the mapping procedure

the event to its corresponding location in the source code and formal specification. However, the disadvantage of software instrumentation is that it shares the processor with the application program, and therefore may produce interference in terms of both execution time and the ordering of message-passing events between processors.

Less-intrusive forms of program instrumentation are possible, such as hardware instrumentation where specialized components are used to collect trace data. The disadvantage of using hardware instrumentation is the difficulty of establishing a mapping between the low-level form of the collected data and the high-level form of the source code and program specification. Our visualization strategy relies on the existence of a mapping between the collected trace data and the expected behavior model, and thus a software instrumentation approach has been chosen for

PANORAMA. Although this approach may potentially alter the execution behavior of the program, PANORAMA attempts to avoid this possibility by allowing the programmer to selectively add instrumentation for only the processes in the expected behavior model that are of interest.

Program Execution

The instrumented version of the source code may be compiled and executed, where each processing element produces a file of trace data during the program execution. The merging of these trace files is performed by PANORAMA in the post-processing stage.

Post-processing of the Trace Data

The post-processing stage performs a time-ordered merge of the trace files that were generated by the processing elements during the program execution stage. Since distributed-memory systems generally lack a synchronized global clock, additional analysis is then performed to adjust the ordering to be consistent with the happened-before relation. In the PANORAMA framework, it is guaranteed that no message-passing events are listed in the time-ordering as being received before they are sent.

7.3 Graphical Depiction Step

The graphical depiction step of PANORAMA uses both the expected behavior data and the trace data to render a visualization of the program execution that models the expected behavior of the program. In this subsection, we discuss the visualization strategy used by PANORAMA, including a description of the major features available for generating visualizations.

Space-time diagrams display communication events between processing elements

across time. One axis of the diagram represents the processing elements, while the other axis represents time. PANORAMA uses an enhanced version of a space-time diagram for graphically depicting program execution, where the enhancement is an overlapping of the active LOTOS processes onto the diagram. (A process is considered "active" between its recorded entry and exit times.) Since both message-passing events and process entry/exit times are depicted by the diagram, it effectively displays each of the items from the expected behavior model across time as the corresponding events appear in the trace data. By graphically depicting the trace data in terms of items from the expected behavior model, the diagram facilitates a comparison between the actual behavior and expected behavior of the program. Thus, we call this diagram a *Behavior Comparison graph*, or *BC-graph* for short.

The BC-graph is constructed by using arcs to represent message-passing events and shaded rectangles to represent the active processes. (The shaded portion of a rectangle represents the interval between the entry and exit times of the corresponding process.) Different shading patterns are used to distinguish between active processes. The graphical depiction of the events is performed by a playback strategy, where the programmer may choose either to sequence through the events in a step mode or have PANORAMA provide a simulated replay. For example, a BC-graph of the execution of the number-doubling program is illustrated in Figure 7.4. This visualization shows that six processing elements were involved in the computation, consisting of one host (by default, labeled "0") and five worker nodes (labeled "1-5"). The shaded rectangles represent the intervals of time throughout the program execution during which the *Send_Numbers*, *Receive_Replies*, and *Worker_Node* processes were active. The message-passing events that occurred during the program execution are depicted in the model of active processes, thus facilitating a visual mapping of the message-passing events to their corresponding location in the expected behavioral model.

PANORAMA's visualization strategy offers several features that facilitate the

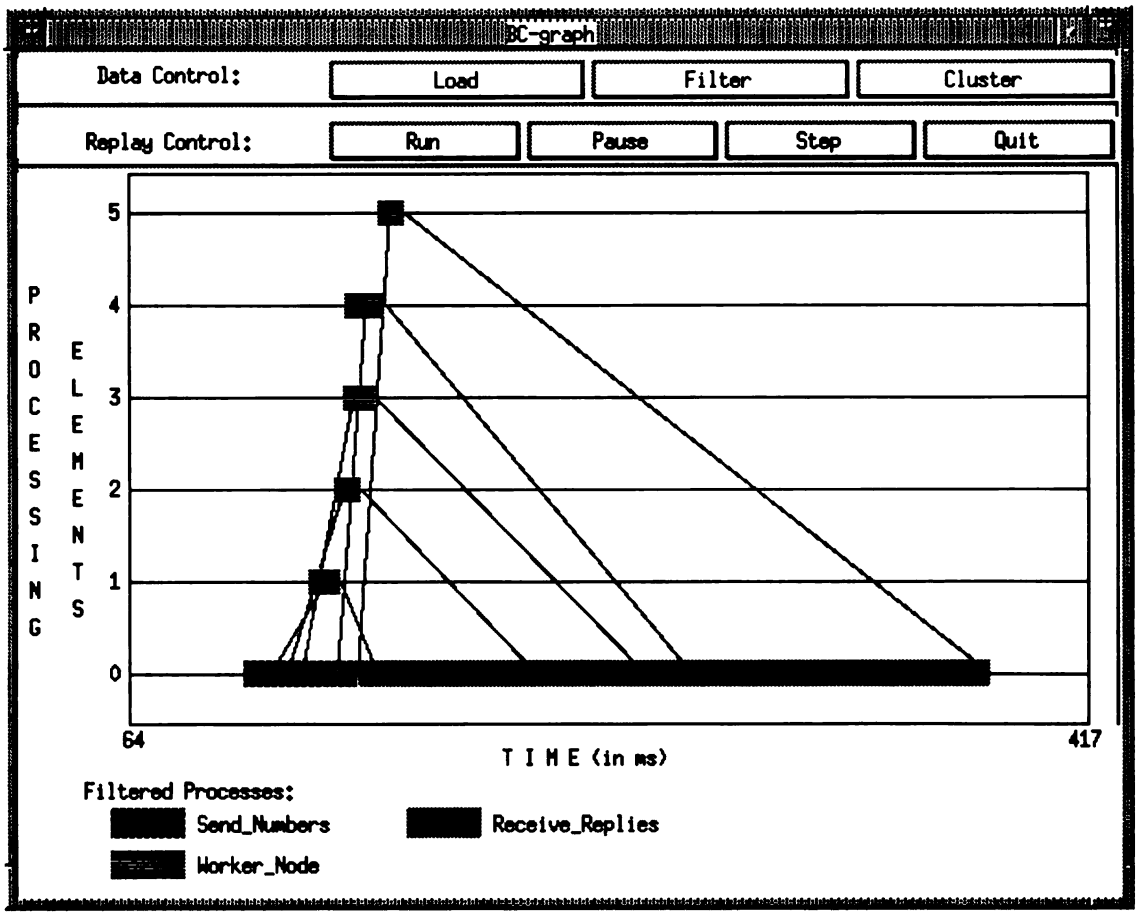

Figure 7.4. BC-graph of the execution of the number-doubling program

debugging of message-passing errors. One of the key features is that the graphical elements that represent the items from the expected behavior model may be selected using the cursor, at which time windows are activated that display the portions of the source code and LOTOS specification that correspond to the selected element. This feature provides a simple method for the programmer to investigate a potential error that was detected in the visualization. Another feature is the ability to perform selective filtering of the events to be depicted, where the processes in the expected behavioral model are used as the basis for the selection. PANORAMA's default graphical depiction strategy is to display the entire set of events recorded in the trace data, potentially leading to congestion. In order to reduce the congestion, the

programmer may choose relevant processes from a listing of the hierarchy of processes, such that the events will be displayed exclusively for the chosen processes. Filtering options are also provided for displaying only the active processes or only the message-passing events. Finally, a BC-graph provides an abstraction (clustering) mechanism for displaying a subtree of active processes by their parent process, thus reducing the congestion that may be caused by displaying the individual active processes (as depicted by the shaded rectangles). This last feature may be used in combination with the selective filtering option to provide a top-down approach to debugging. Examples of several of these debugging features are given in the following section.

7.4 Debugging Example: Cholesky Factorization

Space-time diagrams are useful for locating errors in simple message-passing patterns. However, for cases in which the message-passing patterns between processing elements are either complex or nonexistent, the benefit of space-time diagrams is limited. In this section, we illustrate an example in which PANORAMA facilitates the task of debugging a message-passing error, where the error is not readily detected when viewing space-time diagrams. The application we use is the Cholesky factorization program. The trace data for this example was obtained by running the program on a cluster of eight identical ethernet-connected SUN SPARCstation 1 workstations. In the course of the discussion, we provide several PANORAMA-generated visualizations that demonstrate the debugging features of PANORAMA.

The error that we investigate is located in the communication step of the forward substitution phase. Specifically, the processing element that is assigned the column corresponding to the current value of the index variable must wait to receive messages from *TOTAL_WORKERS - 1* worker node processes, where a looping construct is used to implement the receipt of multiple messages. We consider the case in which

the looping construct is implemented erroneously, such that it waits for messages from *TOTAL_WORKERS* worker node processes instead of *TOTAL_WORKERS* − 1. Since only *TOTAL_WORKERS* − 1 worker nodes send messages, the processing element that is assigned the column enters a deadlock state since it is waiting for a message that will never arrive. The other worker nodes are able to proceed to the next iteration, but each one eventually enters the deadlock state upon executing an iteration in which it is assigned the current column. Since the processing elements enter deadlock states, our discussion below considers partial trace files that were generated by the worker nodes before they entered the deadlock state.

The complexity of the message-passing patterns complicates the task of locating the error in the forward substitution phase. For example, a BC-graph of only message-passing events at the point of deadlock is shown in Figure 7.5. (The host node, represented by processing element "0", does not communicate with the worker nodes after the initialization phase, and, thus, there are no message-passing events shown for processing element "0".) This particular visualization perspective is equivalent to that provided by space-time diagrams.

By using the clustering mechanism, the BC-graph may be used to display the communication behavior in a model of the high-level stages of the program. For example, Figure 7.6 shows the message-passing behavior in the synchronous Cholesky factorization and forward substitution phases of the program, where phases are distinguished by the level of shading. By using this type of visualization perspective for initially examining the program execution, the source of the problem may be narrowed down to the forward substitution phase.

In order to gain a better understanding of the erroneous message-passing behavior in the forward substitution phase, filtering may be used to depict key events. For example, Figure 7.7 shows the case in which filters have been applied to depict only the active *For_Sub_Receive_Messages* processes along with the relevant message-passing

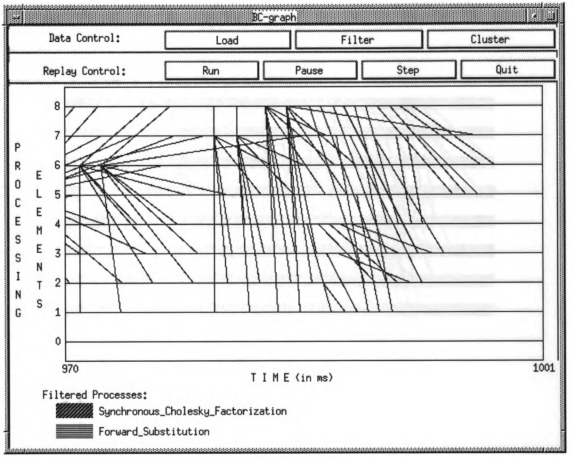

Figure 7.5. BC-graph of only message-passing events

events.

The visualization in Figure 7.7 shows that the processing elements are each waiting in the *For_Sub_Receive_Messages* process at the point where the information in the partial trace files is exhausted. (The depicted events are the last events recorded for each of the processing elements, at which point progress apparently stops since none of the processing elements completed the execution of the program.) The behavior of processing element "1" is of particular interest, as we see that it received one message from each of the other processing elements (i.e., the expected behavior, as defined by the *For_Sub_Receive_Messages* process for a worker node whose *CurrentCol* $\in$ *MY_COL_SET*), yet it failed to exit the *For_Sub_Receive_Messages* process as

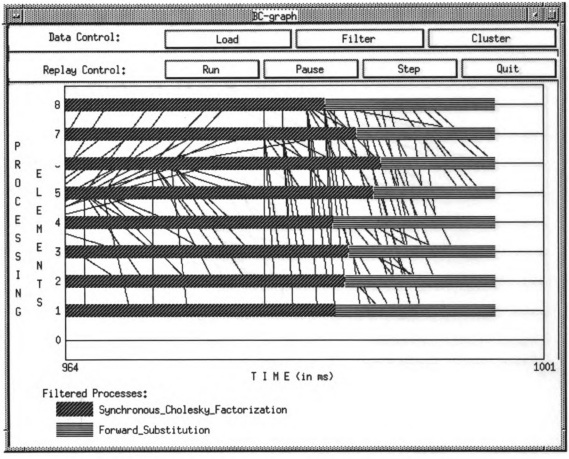

Figure 7.6. BC-graph of both active processes and message-passing events

it should have at approximately the time 994 milliseconds. Upon locating this questionable behavior, the programmer may use the cursor to select any of the graphical elements that represent the process, at which time windows are displayed showing the specification and source code corresponding to the questionable event, as is shown in Figure 7.8. Through a comparison of the contents of the two windows, an inconsistency may be detected in the bound for the number of messages to be received, where the specification states a bound equal to $TOTAL\_WORKERS - 1$, but the source code states a bound equal to $TOTAL\_WORKERS$. This inconsistency explains the cause of the deadlock problem.

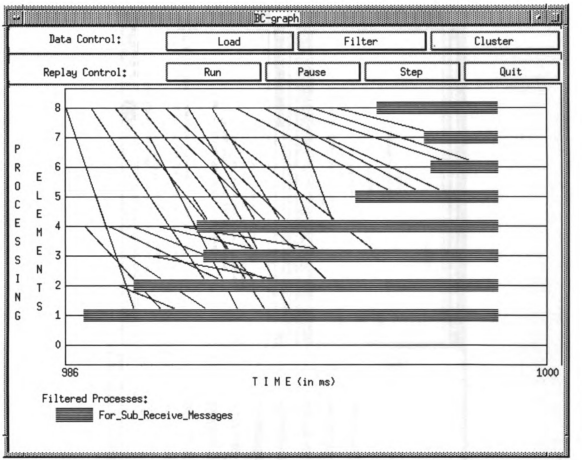

Figure 7.7. BC-graph of the filtered events

7.5 Integration with GOLD

GOLD is designed as on-line debugger, in contrast to PANORAMA, which is a post-mortem debugger. As such, several features were added to the expected behavior modeling approach when it was integrated into GOLD, where these features specifically take advantage of the on-line debugging framework. In addition, some improvements to the interfaces were made, in particular for the mapping tool. These improvements and extensions are discussed below.

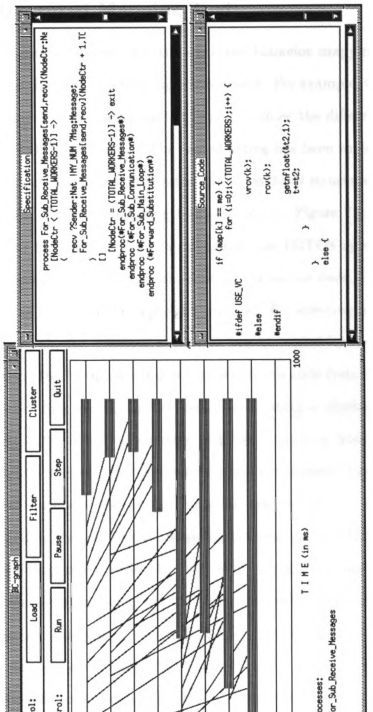

Figure 7.8. Windows for specification and source code corresponding to questionable process

7.5.1 An Improved Mapping Tool

Several improvements were made for the expected behavior mapping tool when it was integrated into the GOLD debugging environment. For example, the PANORAMA version of the mapping tool used a nested listing to show the different levels in the hierarchy of a LOTOS specification. The nested listing has been replaced with a call graph structure, that graphically illustrates the hierarchical structure. For example, the interface for the improved mapping tool is shown in Figure 7.9, where the top left window depicts the hierarchy of processes from the LOTOS specification of the Cholesky factorization program. The structure of the source code is also illustrated via a call graph, as shown in the top right window. By selecting a node from the specification call graph, the events that may be instrumented for the corresponding process are shown in the lower left window. By selecting a node from the program call graph, the corresponding location in the source code listing is displayed in the lower right window. An event is instrumented by making a selection from both the lower left and lower right windows, and then selecting the **Instrument Program** button.

An additional improvement to the mapping tool is that it now assists the programmer in finding the appropriate mapping between an expected behavior event and the corresponding source code. In particular, when the programmer selects a node from the specification call graph, the mapping tool searches for a matching node in the program call graph, and will update the "Program Call Graph" and "Source Code" displays accordingly if it finds a match. If no match is found, the "Program Call Graph" and "Source Code" displays are left unchanged.

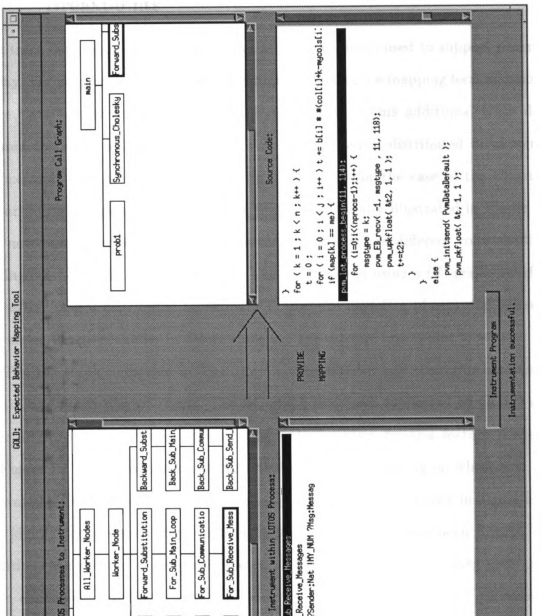

Figure 7.9. The interface for the improved mapping tool

7.5.2 A BC-graph Interface for Causal Distributed Breakpoints

The trace data generated during the initial program run is used to support program replay. For programs that have been instrumented using the mapping tool, additional trace data is generated by each instrumented event. This additional trace data is used to render a BC-graph interface for setting causal distributed breakpoints. An example of this interface is shown in Figure 7.10 for the case of the Cholesky factorization program. The breakpoint that was originally illustrated in Figure 4.3 has now been inserted using the BC-graph interface, where different color shadings distinguish between LOTOS processes. The advantage of using a BC-graph interface for setting causal distributed breakpoints is that the modeling of expected behavior can assist the programmer in determining the appropriate breakpoint location.

The BC-graph supported by PANORAMA offers filtering and clustering options for controlling the display of events. The BC-graph interfaces supported by GOLD also offer filtering and clustering options, where the interface for this feature is shown in Figure 7.11. (The operation of the interface differs depending on whether it was activated via the `Filter` or `Cluster` menu options.) In the case shown in Figure 7.11, the highlighted processes represent those for which events have been filtered. In addition, the subtree of processes rooted at the *Synchronous_Cholesky* node have been clustered, such that events that occur for any of the processes in the subtree will be displayed as occurring for the *Synchronous_Cholesky* process.

7.5.3 Expected Behavior Visualizations

Several visualizations were integrated into the GOLD debugging environment for modeling expected behavior. The BC-graph was one of these, where it is now used to depict communication activity as the events occur, rather than from a post-mortem

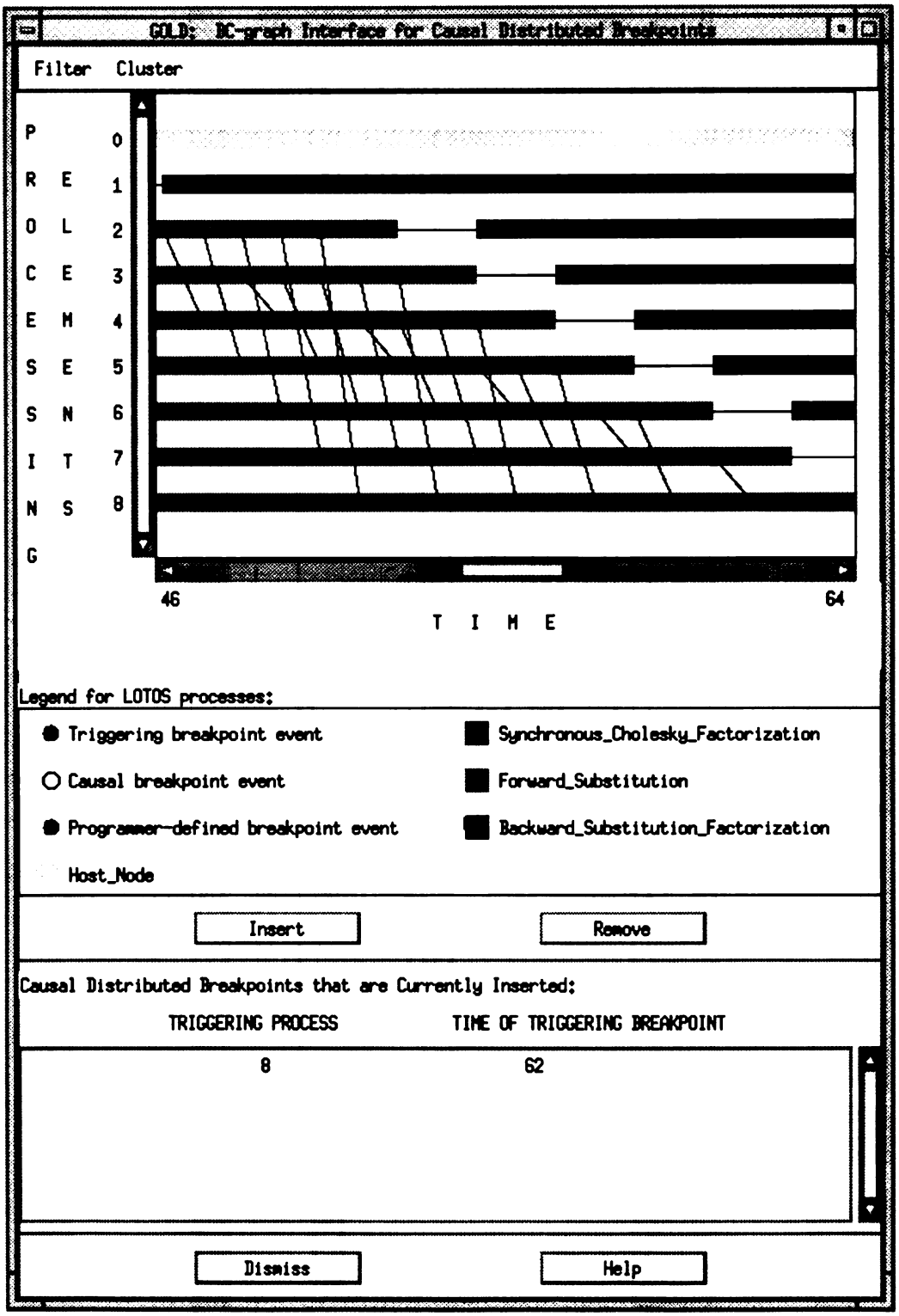

Figure 7.10. BC-graph interface for causal distributed breakpoints

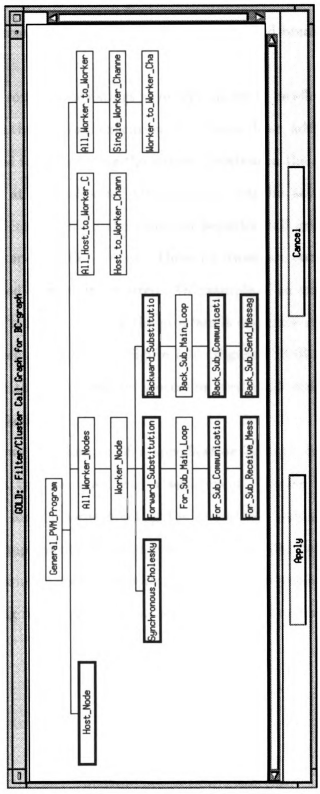

Figure 7.11. The filtering/clustering interface

perspective. An example of the BC-graph interface is shown in Figure 7.12, where the program is currently stopped at the causal distributed breakpoint illustrated in Figure 7.10.

Call graphs and source code listings have been shown to be effective for illustrating the current point in the program execution. Similar to these, additional visualization have been developed for illustrating the current location in the specification. When the program stops at a breakpoint, visualizations may be activated for depicting both the current location within the expected behavior call graph and the current location within the specification listing. These locations are computed according to the last instrumented event that occurred. An example of an expected behavior call graph is shown in Figure 7.13, where the program is currently stopped at the causal distributed breakpoint illustrated in Figure 7.10. Figure 7.13 illustrates that the last instrumented event that occurred for processing element 5 was within the process *For_Sub_Communication.*

Figure 7.14 shows an example of a specification listing. In this case, the last instrumented event that occurred was a send event. Figure 7.14 also shows the source code listing for the same point in the execution, illustrating the portion of the program that implements the send event. This scenario demonstrates a simple approach for comparing expected behavior, as represented by the specification listing, to the corresponding implementation, as represented by the source code listing.

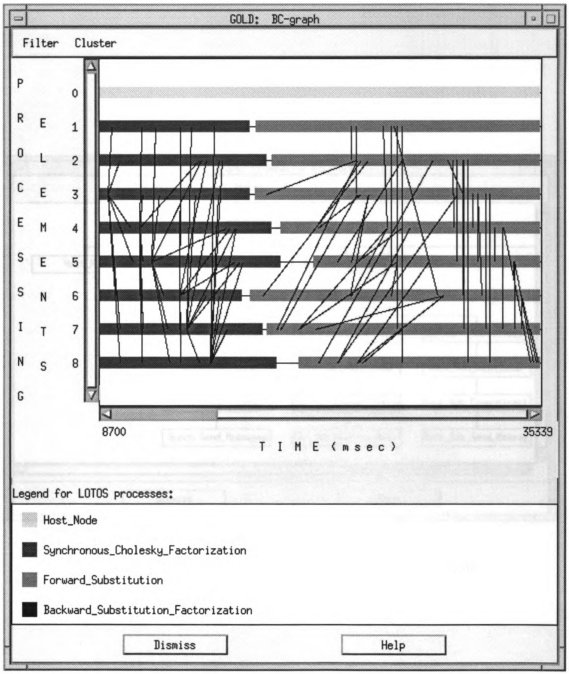

Figure 7.12. BC-graph interface for GOLD

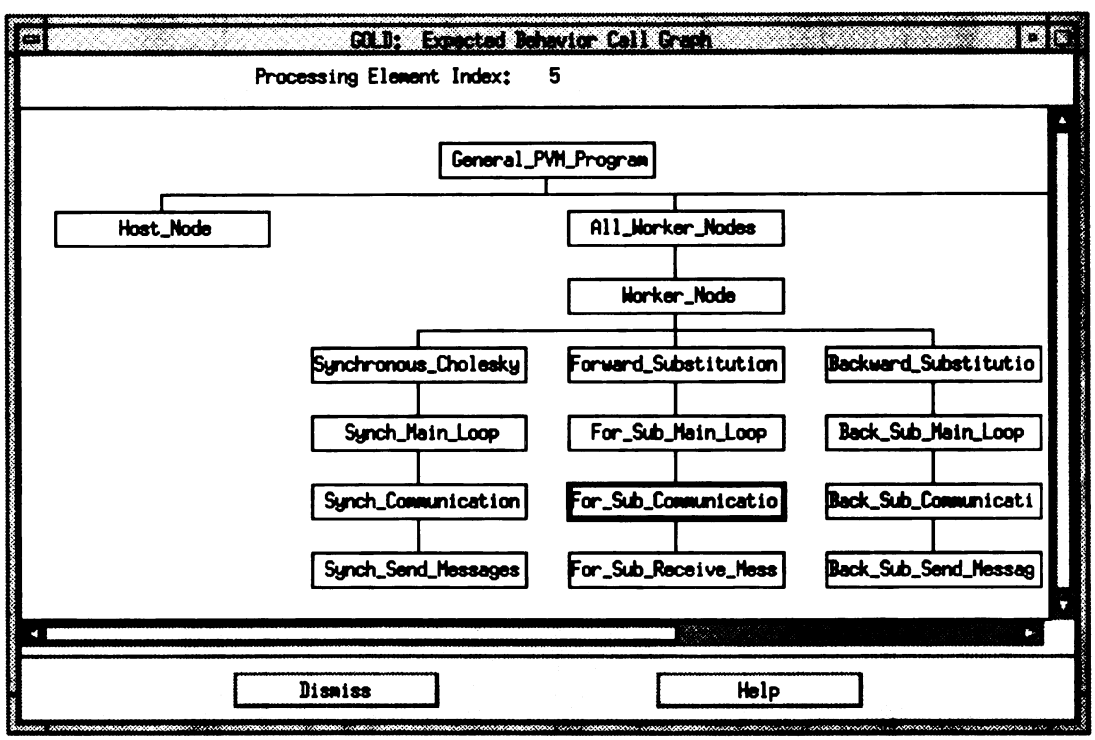

Figure 7.13. Modeling expected behavior in a call graph

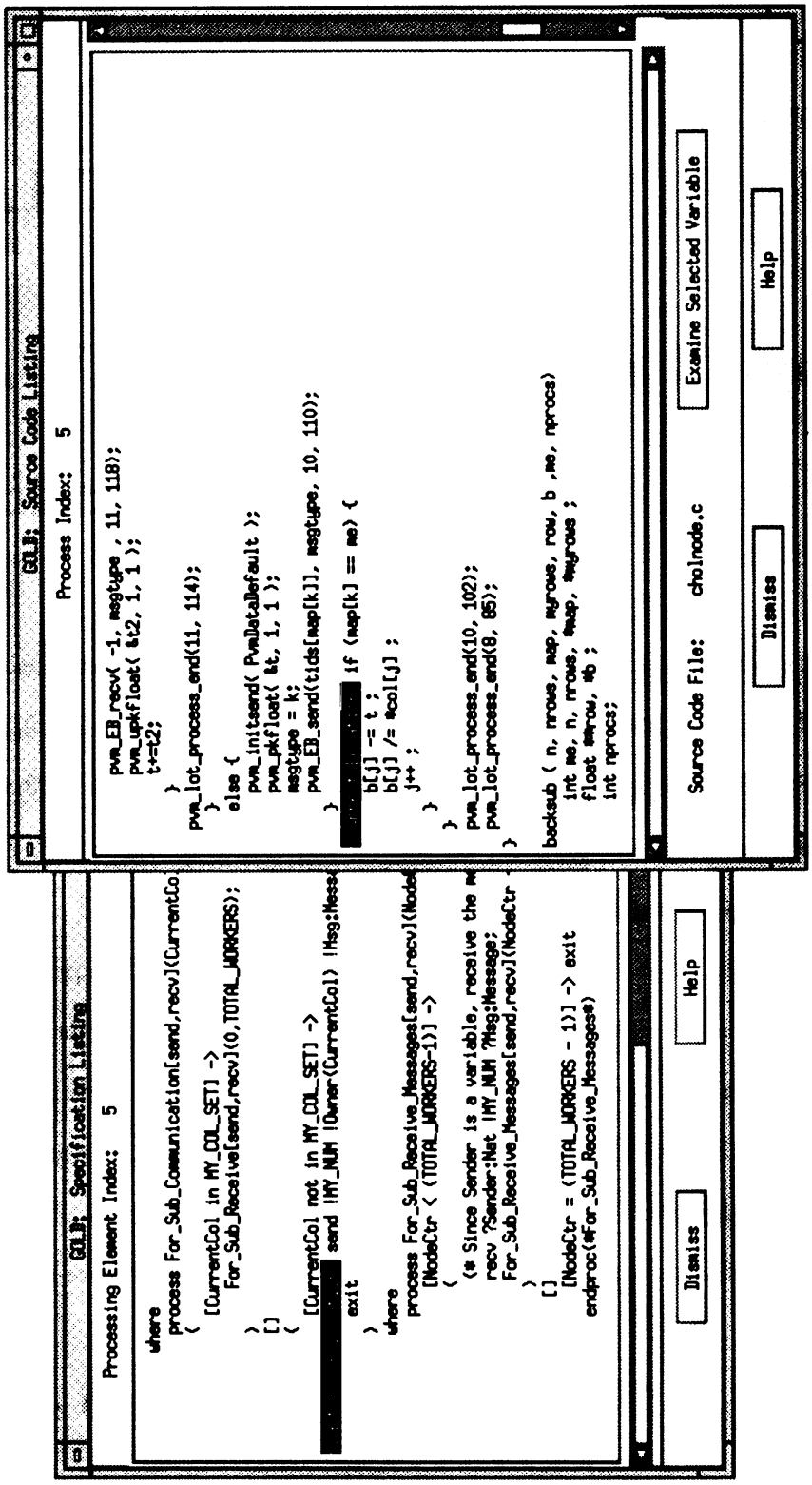

Figure 7.14. Comparison of specification listing to source code listing

CHAPTER 8

Data Visualizations from Formal Specifications

The Lockheed Integrated Visualization Environment (LIVE) project [7, 29, 51] uses formal specifications to facilitate the selection of data visualizations. LIVE maps formal specifications of a program's data structures to a graphical unit termed a *cell model.* The cell model consists of several components used to characterize a data structure. LIVE uses a rule-based system for selecting appropriate debugging visualizations depending on the contents of the cell model. In this chapter, we describe the cell model, show how the formal specification of a data structure is mapped to the cell model, and illustrate how the cell model is used to select visualizations that are useful for debugging. We conclude with the discussion of an improvement to the cell model approach, where the programmer may provide input regarding the selection of the application-specific visualization appropriate for the particular case.

8.1 The Cell Model

The cell model is used to formally capture the characteristics of a data structure so that the mapping from data structure specification to visualization is feasible

[7, 29, 51]. The model contains several components that cumulatively provide enough information to distinguish among data structures with different visual representations. Four components are used to accomplish this task: *dimension*, *size*, *complexity*, and *access ports*. Each of these four components are discussed in detail below.

8.1.1 Dimension

The *dimension* component of the cell model is used to categorize the structural layout of the overall data structure. The value of the dimension component is used to determine the layout of the visualizations used for a particular data structure. The value for this component is a natural number if the data structure layout contains an array-like ordering, as is the case with arrays, stacks, queues, and deques. If the data structure does not contain an array-like ordering, then it is considered *dimensionless* and therefore classified according to its *cell-connectivity*, i.e., the relationship between the cells in the data structure. There are many different types of cell-connectivity, with the following high-level classification:

- *None*: There is no relation between the cells in the data structure. The *set* data structure is an example of such a case since the elements in a set are not necessarily related.

- *Hierarchical*: The relation between cells is such that when a new cell is added to the data structure, it must be at the lowest level in the hierarchy and it cannot have as a *child* a cell that was added earlier to the data structure. A tree is one example of a data structure with this type of cell-connectivity. This type can be further decomposed by differentiating according to the maximum number of children a cell can have.

- *Random, symmetric*: A cell can have a relation to any other cell in the data structure. The *symmetric* property indicates that if there is a relationship

between two cells, then that relationship holds in both directions (i.e., it is directed in both directions). For example, an undirected graph fits in this category.

- *Random, antisymmetric*: A cell can have a relation to any other cell in the data structure. The *antisymmetric* property indicates that if there is a relationship between two cells, it must be in only one direction.

- *Random, not necessarily symmetric*: A cell can have a relation to any other cell in the data structure. The *not necessarily symmetric* property indicates that if there is a relationship between two cells, then it may be in either one or both directions. A directed graph is an example of a data structure with this type of cell-connectivity.

8.1.2 Size

The *size* component is used to denote the capacity of the data structure. The determination of this particular component is optional since the data structure capacity is not always available from the specification. If the capacity of a data structure is available from the specification, then it may be used to fix the boundaries of the visualizations of that data structure. The availability of the capacity is especially advantageous in the case of data structures usually considered to have a static size, as is the case, for example, with arrays.

8.1.3 Complexity

The *complexity* component is used to denote the number of values that are stored within each cell of the data structure. An appropriate glyph (graphical object) may be chosen to represent the cell values given the complexity of the cell as well as the dimensions of the values within the cell. It should be noted that the values within

each cell need not be simple data types with dimension zero (e.g. integers, floats, etc.). For example, given a one-dimensional array of queues (i.e., each element in the array is an entire queue), the complexity of each cell is one and yet the value within a cell is an entire queue (dimension equal to one). In this particular case, the value within a cell can be visualized using a representation for a queue of cells.

For cases in which the dimensions of the values within a cell are all zero, relatively simple glyphs (compared to the above example) may be used to model the values. Depending on their design, such glyphs may have several distinguishable features, where each feature may be used to model a different property. For example, a height tile from a height field visualization has at least two distinguishable features: its height and its color. A height tile may thus be used as the glyph for data structures with a complexity of one or two, where the data values within the cells each have dimension zero. As another example, an arrow has at least four distinguishable features: its height, its color, and its x and y directions. An arrow may thus be used as the glyph for data structures with a complexity of one, two, three, or four, where the data values within the cells once again each have dimension zero.

8.1.4 Access Ports

Access ports are the constraints placed on entering or removing data elements from a data structure. This component was added to the cell model to differentiate between data structures such as stacks, queues, and one-dimensional arrays, where the dimension, size, and complexity may all be identical yet different visual representations are desirable. Currently, the classification of access ports are limited to the following types, although additional types may be added in the future as the visualization of more complex data structures are investigated:

- *Input/Output (I/O) access from a single cell*: A single cell in the data structure

is used for input and output to the overall data structure. The particular cell may change, but we will still have only a single cell handling I/O at any time. This scenario is applicable in the case of a stack.

- *I/O access from two cells*: This type of access is the same as the previous, except that two cells are handling I/O at any time. This scenario would apply in the case of a deque.

- *I/O access from greater than two but less than all cells*: This case is the generalization of the above two.

- *Random I/O access*: Each cell has input/output access. This type of access would apply in the case of an array, where indices are used to map to a particular cell.

- *Input access from one cell in the data structure and output access from another cell*: This type of access permits the input and output functions to be separated. This scenario is applicable to a queue.

The limitation of the cell model approach is that it considers only structural characteristics of data types. In order to obtain a complete understanding of the manipulation of data in a parallel program, operations on the data structures should also be considered.

8.2 Mapping Specifications to the Cell Model

This section defines a mapping from the formal specification of a data structure to the dimension component of the cell model, considering only data structures that have a dimension that can be represented by a natural number (i.e., that are not dimensionless). Extensions to the procedure for mapping to the size, complexity,

and access ports components of the cell model have not yet been considered. For this particular work, the formal specification is written using the Larch specification language [52]. (Although we used Larch for this portion of the investigations, we favor LOTOS for the other investigations. The features of LOTOS that influenced this decision are discussed in Section 2.1.)

Larch is a two-tiered specification language, where each Larch specification has components written in two languages: one designed for a specific programming language (interface) and the other common to all programming languages (shared). The former is referred to as a *Larch Interface Language (LIL)*, and the latter is the *Larch Shared Language (LSL)*. The LSL is an algebraic specification language that allows a programmer to formally specify data abstractions, known as *traits*, that are independent of program state and programming language. A significant advantage to using traits is that they are reusable when writing new specifications. The LIL is used to specify what can be observed about the behavior of components written in a particular programming language. In the following discussion, we limit our use to the LSL. A further discussion of the LIL portion of Larch specifications is presented by Wing [53].

Each Larch trait corresponds to an abstract data type (ADT). An ADT is a class of values and an associated collection of operators that act on those values, where the properties of the operators are specified using only axioms [54]. The operators of an ADT may be grouped into three sets: constructors, modifiers, and observers. The set of constructors are used to produce all possible values of the particular ADT, where the range of each constructor is of a type known as the *distinguished sort*[1]. The set of modifiers are the non-constructor operators whose range is the distinguished sort. Lastly, the observers are those operators that contain the distinguished sort in their

[1]The term "sort" is used in order to avoid confusion with the similar concept "type" from programming languages.

domain, while their range consists of a sort other than the distinguished sort.

An example of a Larch trait is shown in Figure 8.1 for the case of a two-dimensional

---

Array(A, DX, DY, E) : **trait**
introduces
{}: → *A*
bind : *A, DX, DY, E* → *A*
apply : *A, DX, DY* → *E*
defined : *A, DX, DY* → *Bool*
asserts
A **generated by** {}, *bind*
A **partitioned by** *apply, defined*
∀ *a* : *A, di, d1i, d2i* : *DX, dj, d1j, d2j*: *DY, e*: *E*
apply(bind(a, d2i, d2j, e), d1i, d1j) == **if** *d1i* = *d2i* ∧ *d1j* = *d2j*
then *e* **else** *apply(a, d1i, d1j)*
¬ *defined*({}, *di, dj)*
defined(bind(a, d2i, d2j, e), d1i, d1j) ==
(d1i = *d2i* ∧ *d1j* = *d2j)* ∨ *defined(a, d1i, d1j)*
implies
converts *apply, defined*
exempting ∀ *di*: *DX, dj* : *DY apply*({},*di,dj)*

Figure 8.1. Two-dimensional array trait

---

array. The portion of the trait labeled by the **introduces** clause contains a list of the operators for the particular trait. The operators for the two-dimensional array trait are {}, *bind*, *apply*, and *defined*. Each operator is immediately followed by its *signature* that specifies the sorts of the operator's domain and range. The range is always a single sort (S_i), but the domain may consist of zero or more sorts ($S_{i,1}, \ldots, S_{i,k_i}$). Algebraically, k_i represents the number of sorts in the domain of the signature for the operator op_i, where the signature is represented by:

$$S_{i,1}, \ldots, S_{i,k_i} \rightarrow S_i, \qquad k_i \geq 0$$

The "*A* **generated by**" clause indicates that the set of operators immediately following it are the constructors for the sort (in the case of the two-dimensional array trait, {} and *bind*).

The above description of the Larch specification language is a high-level overview, where many details have been omitted for the sake of brevity. For further details regarding the language, the relevant technical reports [52, 55] may be consulted.

The procedure that is used to map from a Larch trait to the dimension component of the cell model is shown in Figure 8.2. Step 2 of the procedure tests for the case in which there exists a single constructor of the form $op :\rightarrow M$ for the sort M, where the domain of the signature is empty. This situation occurs when the data structure can only be assigned a single value, in which case its dimension is zero. An example of this type of data structure is the Boolean value *true*.

Step 4 of the procedure tests for the existence of a constructor for which a "new" value of the data structure is generated by joining two or more "old" values; for instance, the insertion of a tree at the leaf of another tree. The computation of the dimension for this case is beyond the current scope of our technique, and so, consequently, the procedure returns a value of `DIMENSIONLESS`.

Step 6 of the procedure tests for the existence of indices. Two is subtracted from the total count of domain sorts in a signature in order to account for one occurrence of M, plus one occurrence of a sort representing an element to be added to M. Only one of the sorts is an element to be added since the operators have been constructed in an abstract fashion. (If it were possible to add two different types of elements to the data structure, then two different constructor operators would have been used.) The other sorts in the domain of this particular constructor are thus indices for placement of the element, and so the number of these other sorts is equal to the dimension of

INPUT: The trait that defines the sort *M* corresponding to the ADT.

OUTPUT: The dimension of the ADT, where the value `DIMENSIONLESS` indicates that the data structure lacks a dimensional ordering computable by this technique.

PROCEDURE:

1. Determine the set of sorts found in the domains of the constructors' signatures.
2. If the set of sorts found in step 1 is empty, then return dimension equal to zero and stop.
3. Count the maximum number of times that the sort M appears as a domain sort within a constructor's signature.
4. If the result from step 3 is greater than one, then return dimension equal to DIMENSIONLESS and stop.
5. Count the maximum number of domain sorts within any of the constructors' signatures.
6. If the result from step 5 is greater than or equal to three, then return dimension equal to the count minus two and stop.
7. If the set found in step 1 contains a sort not equal to *M* and that sort appears as the range for one of the signatures, then return dimension equal to one and stop.
8. Return dimension equal to `DIMENSIONLESS` and stop.

Figure 8.2. Technique for finding the dimension of an ADT from its trait

the data structure. For example, for the trait of a two-dimensional array, shown in Figure 8.1, step 5 of the procedure determines the maximum number of domain sorts within a constructor to be four for the case of the constructor *bind*. The procedure will thus return the expected value of two.

Step 7 of the procedure tests if there is an ordering imposed on the data structure, such that the cells of the data structure may be depicted in a one-dimensional layout. In order to determine if there is such an ordering, we use the rule that if one of the

non-constructor operators permits access to a *particular* element within the container, then that operator imposes an ordering. Examples of data structures for which this rule applies are stacks and queues.

Finally, step 8 of the procedure handles the case in which no ordering was found for the ADT. Specifically, the procedure returns dimension equal to `DIMENSIONLESS`.

8.3 An Example Application

This section describes how LIVE-generated visualizations based on Larch specifications are used to detect an error in the implementation of a column-sorting routine for the assignment problem. The assignment problem is identical to the (weighted) bipartite graph matching problem, which is defined as: $G = (V, E)$ is an undirected bipartite graph such that V can be partitioned into two disjoint sets S and T and all edges have one end point in S and the other in T. A set M of edges is a matching for G if each vertex in G is the end point of at most one edge in M. In the weighted bipartite matching, a weight $w(i, j)$, is associated with each edge (i, j), and the cost of a matching is the sum of the weights of its edges [56]. For this problem, the maximum matching is the one with the minimum cost. For example, the arcs, represented by ones in the match matrix, shown in Figure 8.3, produce the maximum match with the minimum cost for the corresponding weight matrix.

11	22	4	76
32	95	2	8
100	55	6	7
333	25	88	0

Weight Matrix

1	0	0	0
0	0	1	0
0	0	0	1
0	1	0	0

Match Matrix

Figure 8.3. Maximum match with the minimum cost

A C/Paris [57] implementation of the algorithm was adapted for the case of Single Instruction Multiple Data (SIMD) machines[2] A critical component of the algorithm is the sorting routine. The strategy of this routine is to sort columns by the row position of matches, where the first row equals 0, the second row equals 1, and so on. The row positions of the matched arcs are used to label the columns of a *sort matrix* S, where zeros are placed in the positions that do not contain a match. The preprocessing of the sort matrix S involves spreading the row numbers of the matching arcs along the columns. Next, infinity values, termed *infinites*, are substituted for zeros. An example is shown in Figure 8.4. The columns of this sort matrix are then sorted in ascending order. The weights matrix and the match matrix are sorted based on how S is sorted.

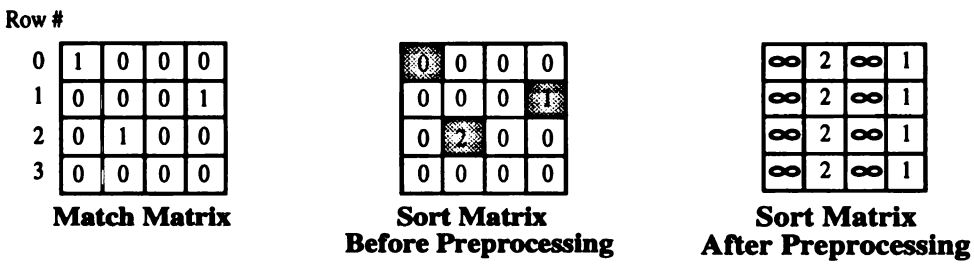

Figure 8.4. Construction of the sort matrix from the match matrix

The underlying data structure for a matrix is a two-dimensional array, and so the trait in Figure 8.1 applies. Using LIVE's rule-based system [51], the programmer is able to generate visualizations for this problem such as those in Figures 8.5 and 8.6.

An error is detected in the visualizations of the match matrix[3] in Figures 8.5 and 8.6, where a white dot represents a match. More specifically, as the sorting progresses, the white dot in the first row moves in the opposite direction of the other

[2]See [56] for details of this algorithm and the results obtained from its implementation.
[3]These visualizations are for a 128 x 128 matrix.

white dots (Figure 8.5) until it eventually ends up at the far right end of the matched columns upon completion of the sorting (Figure 8.6). Since the columns are sorted according to match row positions, this particular white dot should instead migrate to the first column.

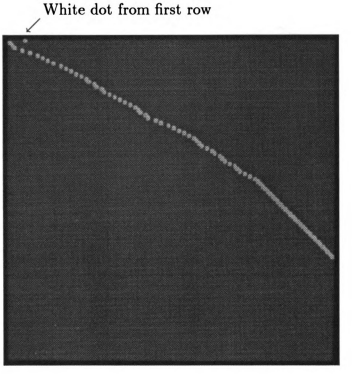

Figure 8.5. A visualization of column sorting for the implementation of the assignment algorithm

From our earlier discussion, we recall that a column without a match will contain all zeros and, thus, be used to set the corresponding column of the sort matrix to infinites. The sorting routine will position these columns to the right of the columns containing a match, as is shown in Figure 8.6. Noting that the misbehaving column is sorted similar to the columns that do not contain a match, we hypothesize that the corresponding column in the sort matrix has been incorrectly filled with infinites.

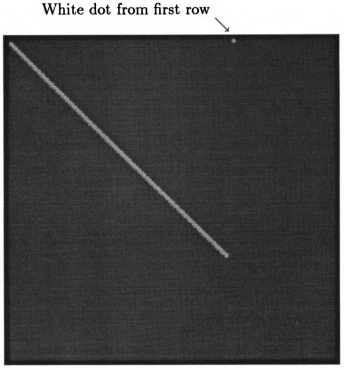

Figure 8.6. The visualization of the match matrix after the misbehaving column sort

(Visualizations of the sort matrix were used to confirm this hypothesis, as discussed in [58].) Backtracking in the procedure to find an explanation for the error, we also recall that row indices start at zero. Since any column of the match matrix containing all zeros will trigger its corresponding column in the sort matrix to be set to infinites, it is concluded that the bug was caused by infinites being incorrectly substituted for the zeroth row position.

8.4 Improvement to the Mapping Strategy

The procedure for determining an appropriate data visualization can be improved by letting the programmer guide the final selection of the visualization. The procedure discussed earlier may be used to initially determine the correct category of data visualizations, based on the static characteristics of the data structure. At that

point, a graphical user interface may then be used to let the programmer select the application-specific visualization appropriate for the particular case.

As an example, we consider the case in which the algorithm discussed earlier determined that a particular data structure was of type 'integer', based on the information in the trait. There are several possibilities for how an 'integer' may be used, where, in each case, the programmer may have a different conceptual idea of how the data visualization should illustrate the respective operations. Some of the possible uses for an integer are the following, where the characteristics of an appropriate data visualization are discussed in each case:

- Counter: The basic operation is to increment the integer, where the initial value of the integer is zero. A data visualization for illustrating this use of an integer could be similar to that shown in Figure 8.7, where the increment operation is illustrated by "filling" the gauge by one unit.

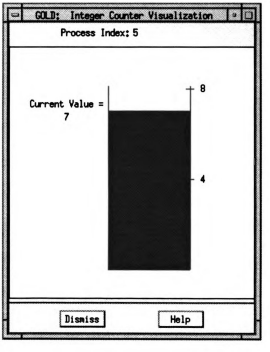

Figure 8.7. A data visualization for a counter

- Array index: An integer used as an array index may be either incremented or decremented, although the value of the index should always lie within the bounds of the array. If the array index extends outside the bounds, then the application-specific visualization should highlight the error. (For example, this case may occur if a program references position $n + 1$ of an array that contains only n elements.) In order to illustrate this type of data visualization, the number of elements in the array must be determined, in addition to the value of the integer.

- Other uses for an integer: A default case should be provided, where the textual value of the integer is simply printed.

CHAPTER 9

Related Work

In this chapter, we discuss work related to the thesis research. The discussion is organized into two sections corresponding to the two main contributions of the research. Specifically, we present investigations into the integration of visualization-based functionality with on-line parallel debuggers, and we then present alternative modeling techniques for visualizations of program execution.

9.1 On-line Parallel Debuggers Supporting Visualization

Replay techniques offer a promising approach for providing deterministic execution in on-line parallel debugging tools [12, 13, 14, 15]. However, many common debugger operations, such as breakpointing and the testing of predicates, present problems of their own for the case of a parallel programming environment, and thus must be resolved before a completely functional on-line parallel debugger is feasible. Methods for performing such operations have only recently been offered for the case of a parallel environment [22, 41, 42, 59]. Consequently, the builders of visualization-based parallel debuggers have typically designed their tools based on a post-mortem approach, where development methods are better established than for the case of

on-line debugging [6]. Thus, the use of visualization in on-line parallel debuggers is still relatively modest at this time. We discuss below the small number of cases where visualization has been integrated into the design of an on-line debugger, illustrating in each case how visualization has supplemented the usual operations of an on-line debugger. In all cases, the use of visualization is mainly limited to the depiction of message-passing behavior. Where applicable, we also discuss scalable techniques that have been applied for handling executions with large numbers of processes.

9.1.1 Bugnet

Bugnet [16, 60] provides a variety of on-line debugging capabilities for the case of distributed programs that communicate via message-passing. The design of Bugnet is based on a replay strategy, where the messages along with their contents are saved during an initial program execution. In addition, checkpoints are periodically recorded during the initial execution so that replay can begin at intermediate program states rather than from the beginning of the program. During replay, the user can re-execute either a single processing element or a selected subset of processing elements, where the messages from any processing elements not being replayed are supplied by the message log. Although the replay mechanism of Bugnet is useful for debugging, many other common on-line debugging operations are not offered, such as breakpointing.

Bugnet supports visualization of program execution by allowing the user to arrange groups of processing elements (depicted as boxes) in an arrangement desirable for viewing the interaction between the processing elements. As the program executes, the beginning of a message-passing event is shown by having the sending processing element darken, followed by a line appearing between the sending processing element and the receiving processing element, and then having the receiving processing element darken. The ending of the message-passing event is shown by having

the sending processing element lighten, the line between the processing elements disappear, and finally by having the receiving processing element lighten. The limitations of this visualization strategy are that it does not scale well, it does not show message-passing activity across time effectively, and it is difficult for the programmer to relate the visualizations to the expected behavior for the program.

9.1.2 Idd

Idd [17] is another example of an on-line debugging tool for distributed, message-passing programs. One of the key features of Idd is that it allows breakpoints to be set in the source code, but with the limitation that processing elements are not guaranteed to stop in a consistent state. Another feature of Idd is that it allows the programmer to define assertions for the global message-passing activity, where a *supervisor* program automatically stops program execution when an assertion is violated. At that time, the programmer is able to inspect the various processing elements in order to search for the error.

The monitoring component of Idd supports the visualization of message-passing events as they occur. These events are displayed across time in the form of a space-time diagram, where the y-axis of the diagram represents the processing element number, the x-axis represents the time, and lines between points in the diagram represent message-passing activity. Scroll bars are provided on each of the axes to allow the programmer to go either forward or backward through the processing elements or time. This visualization approach supports scalability and depiction of behavior across time better than that provided by Bugnet, but the programmer is still left with the task of finding a match between the expected behavior for the program and the behavior that is actually depicted.

9.1.3 VISTOP/TOPSYS

The visualization tool *VISTOP* is part of the integrated tool environment *TOPSYS* for programming message-passing multiprocessors, designed specifically for the parallel programming library MMK (Multiprocessor Multitasking Kernel) [18]. An on-line mode for the TOPSYS monitoring system allows breakpoints to be set in an application, where state inspection is possible when the breakpoint is reached during program execution. An additional option provided by TOPSYS is that variables may be modified when breakpoints are reached in order to test fixes. TOPSYS also offers an alternative post-mortem mode, where traces are recorded during the program execution to later be visualized off-line.

The visualizations provided by VISTOP depict program execution in terms of the basic object types of MMK: the task, the mailbox, and the semaphore. Icons are used to distinguish between each of these object types. The communication of a task with a mailbox or a semaphore is indicated by an arrow. A thin arrow represents the actual communication, while a thick arrow represents the case where a task is waiting for a communication request to be satisfied. The limitations of VISTOP are the same as for Bugnet: lack of scalability, poor depiction of behavior across time, and the difficulty of relating the visualizations to the expected behavior of the program. In addition, since VISTOP is specifically oriented towards depicting the basic objects of MMK, its visualization approach is not easily portable to other message-passing programming environments.

9.1.4 ParaRex

In the cases of Bugnet, Idd, and TOPSYS, program visualization was provided as an additional feature of an on-line parallel debugging tool. In the case of *ParaRex* [19], however, the design strategy was to integrate the post-mortem visualization tool

ParaGraph [24] with the on-line debugging capabilities of DECON, a debugger for the distributed-memory machine iPSC/2. The result of this integration was a debugging tool that used the features of ParaGraph to provide on-line visualization, while simultaneously allowing examination of variables through the features of DECON. An execution replay technique was used as the underlying mechanism for providing deterministic execution, and, thus, meaningful results.

The visualization options offered by ParaRex are, of course, identical to those of ParaGraph. Specifically, ParaGraph provides a large variety of visualizations, covering many different aspects of a program's execution. Examples of the information visualized include message-passing activity, the degree of parallelism at any instant, and statistical data, such as the total volume of messages. One disadvantage of the visualization approach of ParaRex is that ParaGraph's visualizations are actually oriented towards locating performance problems rather than debugging functional errors. In particular, as was the case for the three tools discussed earlier in this section, the visualizations of ParaRex are not easily related to the expected behavior of the program. Another disadvantage of ParaRex is that DECON has been designed specifically for the iPSC/2, and thus the overall approach used by ParaRex is not easily portable.

9.1.5 Panorama

The design strategy of the tool *Panorama* [20] was similar to that of ParaRex, in that visualization capabilities were integrated with the functionality of an on-line debugger[1]. However, in the case of Panorama, a key design goal was portability across a variety of parallel platforms. Thus, the developers of Panorama have designed generic

[1]Despite the resemblance in names, the tool Panorama should not be confused with the tool PANORAMA, discussed earlier in Section 7. Both tools were developed in approximately the same time frame, and unfortunately were assigned similar names.

debugger functions to interface between the base debugger of a parallel machine and the visualization component of Panorama. The base debuggers that Panorama supports are the ipd debugger from the iPSC/860 and the ndb debugger from the Ncube.

Two visualizations are offered by Panorama: a time-line visualization and a processor map visualization. The time-line visualization is similar to a space-time diagram, except that send and receive events are distinguished by using a circle to represent a send and a square to represent a receive. The processor map visualization is an alternative portrayal of message-passing activity, where the programmer may place the processors (shown as boxes) in an arrangement desirable for viewing their interaction, with the interconnections between processors shown as lines between the boxes. In this case, message-passing activity is depicted by labeling each incoming line with the number of messages waiting to be read from the neighboring processor. Although this latter visualization provides a different perspective for viewing message-passing behavior, neither visualization offered by Panorama relates such behavior to the expected behavior of the program.

9.1.6 PPT/xipd

A *program phase tree (PPT)* [46] is a graphical representation of program structure, where the programmer may modify this representation in order to portray program activities in a particular manner. A PPT is, by itself, not a tool, but rather a type of visualization. The research discussed in [46] illustrates how PPT visualization capabilities were added to the xipd on-line debugger for the iPSC/860.

In contrast to the visualizations provided by the tools discussed above, a PPT focuses on illustrating the entries and exits from the various phases of a program rather than message-passing activity. The initial form of a PPT is a standard call graph, but the programmer can delete nodes or edges and collapse subtrees until

a desired arrangement is achieved. During program execution, as each processing element invokes a procedure corresponding to an entry point of a node, a new line is drawn between the caller and callee nodes, where multiple colors are used to distinguish between the invocations by different processing elements.

A key advantage of a PPT is that it provides the programmer with the flexibility to visualize program execution in a format that resembles the expected behavior of the program, although it is limited to structures based on the call graph of the program. Some disadvantages of a PPT are that it does not illustrate behavior across time well, nor does it show the interaction between processing elements.

9.1.7 Prism

Prism [61] provides on-line debugging capabilities for message-passing programs, specifically developed for the CM-5. A main visualization supported by Prism is a "where tree", a call graph structure that illustrates where each process is located in its execution. As such, the "where tree" shows the global status of the system in a single visualization. However, a "where tree" does not illustrate communication between the processes.

One of the features of Prism is an option for filtering which processes should be targeted for debugging. A language is provided for specifying lists and ranges of processes to which debugging commands should be issued. Although the language is extensive, the disadvantage of this approach is that a programmer must first learn the language before process filtering can be applied.

9.2 Models for Visualization of Program Execution

In Section 7, we discussed a strategy for visualizing a parallel program's execution using a model of expected behavior, where the expected behavior model was constructed based on the LOTOS specification of the program. We then showed that debugging of incorrect message-passing communication is facilitated by modeling visualizations in this manner. We also described, in Section 8, a method for generating debugging visualizations of data structures, where, in that case, the visualizations were modeled based on the Larch specifications of a program's abstract data types. We will now discuss additional models that have been used to construct visualizations oriented towards debugging incorrect behavior in parallel programs.

9.2.1 Event-oriented Models

A common strategy for the construction of visualizations is to use a model based directly on the event data generated during the program execution. The main advantage of this approach is that little or no post-processing of the event data is required, but the main disadvantage is that the programmer must manually find a correlation between the graphically-represented events and the conceptual understanding of the program. An example of a visualization constructed according to an event-oriented model is a space-time diagram, as was discussed in Section 7.3. In this case, the event data recorded during the program execution is used to directly render a visualization of communication between processing elements across time. Examples of visualization tools that offer space-time diagrams or other visualizations constructed according to an event-oriented model are ParaGraph [24], Idd [17], the Moviola system [23], and Radar [62].

9.2.2 Clustering Techniques

An approach that has been used to reduce the congestion in a visualization of event data is to cluster groups of low-level events into user-defined high-level events. The visualizations of program execution are then modeled in terms of these high-level events. For example, the tool Belvedere [63, 64] visually abstracts low-level behavior of a program by using the *Event Definition Language (EDL)* [65] to define the pattern of low-level events that constitute a particular high-level event. An event recognizer is used to identify occurrences of the high-level events among a stream of low-level events. The visualizations produced by Belvedere depict program execution in the context of the high-level events. There is an overhead cost in this approach, however, since the programmer must learn and apply EDL to construct the high-level events. A similar tool, Ariadne [66], also supports the definition of high-level events in terms of low-level events, but suffers the same disadvantage in that the programmer must learn and apply its event modeling language. The approach that we used in the PANORAMA project avoids this additional overhead by modeling the high-level events of a program in terms of the program's formal specification, where this specification has already been created during the design phase of the program.

9.2.3 Source Code Models

One approach that has been used to simplify the task of finding a correlation between the program visualizations and one's understanding of the program is to model the visualizations in terms of the source code structure. An example of a visualization constructed according to this approach is a dynamic call graph, where the nodes in the graph represent high-level program objects such as subroutines and the lines between nodes represent call relations. Tools which provide dynamic call graphs include Faust [67], Schedule [68], and the visualization system of Zimmermann, Perrenoud, and

Schiper [69].

An enhanced version of a dynamic call graph is a program phase tree (PPT) [46], as was discussed in Section 9.1.6. This type of visualization provides the programmer with the flexibility to modify a standard call graph to more closely reflect a particular understanding of the call relations in the program.

Another type of visualization modeled in terms of the source code structure is a flow diagram. For example, PF-View [45] represents program execution as a series of icons linked linearly to show the flow of execution, where each icon corresponds either to a parallel construct or to serial code. The programmer may click on an icon to expand it, revealing the individual processors involved in the corresponding computation.

The disadvantage of using dynamic call graphs or flow diagrams is that the level of insight that they can provide is limited to aspects of call or flow relations within a program. If the incorrect behavior is not based entirely on these relations, then these visualizations, when used alone, have limited utility. When used in an environment that supports multiple visualization windows, however, the information they provide may be correlated with the information from other visualizations to provide a beneficial overall view of program execution. The advantages of using multiple visualizations to depict a program's execution are discussed by LeBlanc, Mellor-Crummey, and Fowler [23].

9.2.4 Application-specific Techniques

Application-specific visualizations are those that have been designed to visually model one's natural understanding of a program. Examples of tools that support application-specific visualizations are Voyeur [70], POLKA [71], and ParaGraph [24]. An example of a typical application-specific visualization is one used to depict the execution of a sorting program, consisting of the graphical image of an aligned set

of bars (rectangles), where the height of each bar reflects the magnitude of the corresponding number. The layout for such a visualization would be similar to that shown in Figure 9.1, where ten bars (corresponding to numbers) are depicted in a random order. As the sorting algorithm proceeds, the bars exchange positions accordingly, until, at the end of the execution, they are all ordered in either increasing or decreasing order.

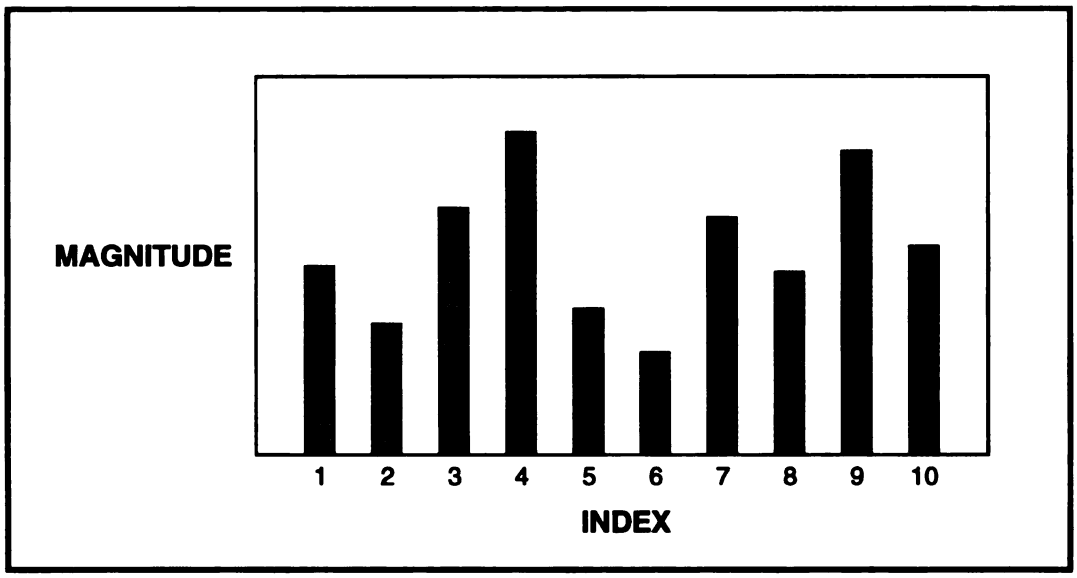

Figure 9.1. Typical layout for an application-specific visualization of the sorting problem

The main advantage of application-specific visualizations is that they are designed to reflect one's conceptual understanding of a problem, but the main disadvantage is that a new visualization must be created for each new problem investigated. Since window programming is tedious, the time requirements for creating application-specific visualizations may outweigh their benefit. In order to make the application-specific visualization approach feasible, tools are often developed to simplify the task of creating new visualizations. For example, Voyeur [70] is an application-

specific visualization tool that provides a hierarchical library of visualizations. The creation of new visualizations is simplified by starting with the inherited code from a visualization higher up in the hierarchy. A sample of the class hierarchy of visualizations for Voyeur is shown in Figure 9.2. As an example, the `x,y` class of visualizations have the characteristic that they represent state by placing graphical elements on an `x,y` coordinate space that fills the drawing area. The icon visualization is constrained to using a grid with integer-valued coordinates, while, in contrast, the vector visualization is constrained to using real-valued coordinates. For both the icon visualization and the vector visualization, the behavior may be derived from the base `x,y` class. A time-savings is achieved by starting with the code from the base `x,y` class when deriving the icon and vector classes.

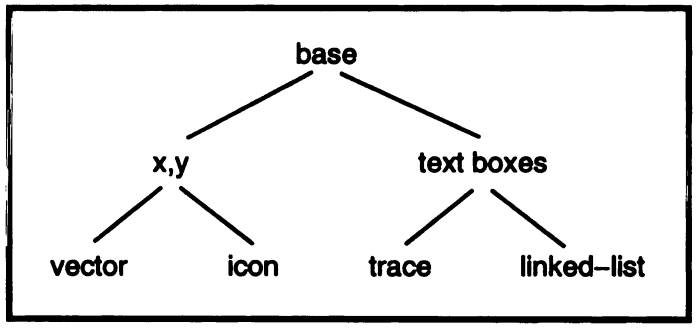

Figure 9.2. Sample class hierarchy of Voyeur visualizations

9.2.5 Property-oriented Models

In Sections 7 and 8, we discussed methods that we have used to model visualizations in terms of the formal specification of a program. Another similar approach is to model visualizations in terms of formal properties used in correctness proofs of a program. For example, Roman and Cox have developed a method for mapping from

the shared dataspace paradigm to appropriate visualizations [72, 73]. The shared dataspace paradigm is one in which processing elements have access to a common content-addressable data structure (typically a set of tuples) whose components may be examined, inserted, and deleted. The mapping used by Roman and Cox is from shared dataspace program states to graphical objects that may be rendered as images. In particular, formal properties expressed in a logical calculus are modeled within the images, such that interpretation of the images may be done in the context of the properties used to reason about the computation. A disadvantage of the approach presented by Roman and Cox is that the mapping method is limited to the shared dataspace paradigm. Popular parallel programming paradigms such as shared variables and message-passing are not handled.

CHAPTER 10

Conclusions and Future Investigations

This dissertation has discussed research that provided two main contributions to the area of visualization-based parallel debugging. First, we developed new techniques for using visualization to support the operation of an on-line parallel debugger [9, 11]. The results included a visualization-based approach for the insertion of distributed breakpoints, where the difficult task of setting distributed breakpoints is reduced to a simple, graphical operation. For example, in order to insert a causal distributed breakpoint, the programmer selects an appropriate event from a space-time diagram, where the graphical interface is then updated to reflect the location of the last event along each process axis that had a causal effect on the execution of the triggering process. The programmer may use this interface to evaluate the location of potential breakpoints, and to insert those breakpoints that are found to be relevant to the current problem. In order to assist the programmer in setting a distributed breakpoint, we also provided support for marker events. The programmer may use marker events to instrument the program at appropriate points, where the generated trace data from the marker events is then used to mark the relevant execution points in the space-time diagram. This feature facilitates the task of finding the appropriate

location for inserting a distributed breakpoint.

Another technique that was developed to support the operation of an on-line parallel debugger was scalable techniques for visualizing communication events [10, 11]. The basis for these techniques was to use filtering to reduce the amount of information presented in the visualizations. The filtering options are provided via a simple graphical user interface, such that the programmer may easily direct which events should be filtered. A key outcome of this approach was a new visualization referred to as a Single Point-of-View (SPV) communication graph, in which only the events received at or sent from a selected process are depicted.

An additional outcome of our research is the integration of several types of visualizations into a common environment for supporting the top-down examination of program states [9, 11, 25]. These visualizations included communication graphs (i.e., space-time diagrams and SPV graphs), call graphs, source code listings, and data visualizations. In the case of the data visualizations, additional filtering options were provided to highlight particular instances of data, such as all values that exceed some maximum value, or all values that are within some range. These visualizations are applied in a framework where program execution is initially examined from a broad scope, using visualizations such as communication graphs. In order to determine the potential source of an error, the user may narrow the scope of the visualizations that are used, applying call graphs, source code listings, or data visualizations, in that order. Graphical support is provided for coordinating the use of the individual visualizations. This visualization-based framework is beneficial since it supports a methodical approach for using visualizations to debug a parallel program.

The second main contribution of this dissertation was the development of a methodology for visualizing a program's execution using the formal specification of the program as a model of expected behavior. One outcome of this research was a strategy for modeling expected behavior in space-time diagrams [8, 26]. A visualization

referred to as a BC-graph was developed, where shaded rectangles representing LOTOS processes were overlapped onto a space-time diagram of the message-passing events. This visualization may be used to compare the expected behavior, as represented by the LOTOS processes, to the actual behavior, as represented by the message-passing events. We also developed techniques for visualizing the current location in a program execution in terms of the formal specification. When the program stops at a breakpoint, visualizations may be activated for depicting the current location of a process either within the call graph of the specification or within the specification listing. These visualizations provide a simple approach for visualizing the current program activity within a model of the program's expected behavior.

An additional outcome of this research was a technique for using the formal specifications of a program's data structures to guide the generation of data visualizations [7, 29]. In particular, the formal specifications are mapped to a graphical unit termed a cell model. The cell model consists of several components used to characterize a data structure. A rule-based system is then used to select appropriate debugging visualizations depending on the contents of the cell model.

The research contributions discussed above were implemented in prototype debugging environments, where the main purpose of these environments was to give a proof of concept demonstration of the research results. The debugging environment in which most of the contributions were implemented was the Graphical On-Line Debugger (GOLD), a parallel debugger for PVM message-passing programs. GOLD may be used to debug a program as it is running, where a replay strategy is used to guarantee that re-executions of the program will be deterministic. By developing GOLD using a replay approach, visualization could be applied liberally, since events were guaranteed to occur in the same relative order as in the initial execution.

Two other debuggers, PANORAMA and LIVE, were developed prior to the development of GOLD, where the purpose of these debuggers was to demonstrate

various aspects of the methodology for using a program's formal specification as a model for visualizing the program's execution. These debuggers were useful for these demonstrations, but could not be used for the entire span of our research due to certain limitations. In particular, PANORAMA was developed according to a post-mortem approach, where monitor intrusion and data storage constraints limited the visualization capabilities. In the other case, LIVE was developed for SIMD programs, where a SIMD program has a single instruction stream. Thus, LIVE could not be used to debug programs that use multiple instruction streams. These limitations motivated the development of GOLD.

Future investigations should continue to address scalability issues for program visualization, particularly considering the cases where programs use a large number of processes or large data structures. Although this dissertation has presented several filtering options for handling these cases, the scalability of parallel debuggers must continue to improve in order to efficiently support the needs of parallel programmers. For example, visualization-based techniques may be explored for clustering processes, such that the number of separate items plotted along the vertical axis of a space-time diagram is reduced. In addition, visualization-based techniques for examining large data structures may be explored, such as improving the filtering techniques for one- and two-dimensional arrays.

An additional area for future investigation is to strengthen the tool support for parallel program specification. The specification of parallel programs is a growing science, yet tool support must still be improved. In the particular case of LOTOS, tools that would be beneficial for supporting the expected behavior modeling techniques discussed in this dissertation include editors, syntax checkers, semantic checkers, and simulators. (Although there exist tools that perform these functions for LOTOS, they are mostly at a prototype level.)

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Keijiro Araki, Zengo Furukawa, and Jingde Cheng. A general framework for debugging. *IEEE Software*, pages 14–20, May 1991.

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

[3] Jeannette M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, pages 8–24, September 1990.

[4] Betty H. C. Cheng. Synthesis of procedural abstractions from formal specifications. In *Proc. of COMPSAC'91*, pages 149–154, Tokyo, Japan, Sept. 1991.

[5] Betty H. C. Cheng. Applying formal methods in automated software development. *Journal of Computer and Software Engineering*, 2(2):137–164, 1994.

[6] Eileen Kraemer and John T. Stasko. The visualization of parallel systems: An overview. *Journal of Parallel and Distributed Computing*, 18:105–117, 1993.

[7] Mark V. LaPolla, Joseph L. Sharnowski, Betty H. C. Cheng, and Kevin Anderson. Data parallel program visualizations from formal specifications. *Journal of Parallel and Distributed Computing*, 18:252–257, 1993.

[8] Joseph L. Sharnowski and Betty H. C. Cheng. A formal approach to modeling expected behavior in parallel program visualizations. In *PARLE '94: Parallel Architectures and Languages Europe, Lecture Notes in Computer Science, vol. 817*, pages 202–213, Athens, Greece, July 1994. Springer-Verlag.

[9] Joseph L. Sharnowski and Betty H. C. Cheng. A visualization-based environment for top-down debugging of parallel programs. Accepted to appear in the Proceedings of the *9th International Parallel Processing Symposium (IPPS '95)*.

[10] Joseph L. Sharnowski and Betty H. C. Cheng. A scalable approach to visualization-based parallel debugging. Technical Report CPS-94-67, Michigan State University, December 1994. Submitted for publication.

[11] Joseph L. Sharnowski and Betty H. C. Cheng. GOLD: A Graphical, On-Line Debugger for parallel programs. Technical Report CPS-95-16, Michigan State University, April 1995.

[12] Thomas J. LeBlanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–481, April 1987.

[13] Richard H. Carver and Kuo-Chung Tai. Replay and testing for concurrent programs. *IEEE Software*, pages 86–74, March 1991.

[14] Robert H. B. Netzer and Barton P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Supercomputing '92*, pages 502–511, 1992.

[15] Robert H. B. Netzer and Jian Xu. Adaptive message logging for incremental program replay. *IEEE Parallel & Distributed Technology*, pages 32–39, November 1993.

[16] Larry D. Wittie. Debugging distributed C programs by real time replay. *SIGPLAN Notices*, 24(1):57–67, January 1989.

[17] Paul K. Harter, Jr., Dennis M. Heimbigner, and Roger King. Idd: An interactive distributed debugger. In *Proc. of IEEE 1985 Distributed Computing Systems*, pages 498–506, 1985.

[18] Thomas Bemmerl and Peter Braun. Visualization of message passing parallel programs with the TOPSYS parallel programming environment. *Journal of Parallel and Distributed Computing*, 18:118–128, 1993.

[19] Eric Leu and André Schiper. Execution replay: A mechanism for integrating a visualization tool with a symbolic debugger. In *Proceedings of 2nd Joint International Conference on Vector and Parallel Processing (CONPAR 92 - VAPPV), Lecture Notes in Computer Science, vol. 634*, pages 55–66, Lyon, France, September 1992. Springer-Verlag.

[20] John May and Francine Berman. Panorama: A portable, extensible parallel debugger. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 96–106, San Diego, California, May 1993.

[21] Cherri M. Pancake and Sue Utter. Models for visualization in parallel debuggers. In *Proceedings of 1989 Supercomputing Conference*, pages 627–636, 1989.

[22] Jerry Fowler and Willy Zwaenepoel. Causal distributed breakpoints. In *Proceedings of 10th International Conference on Distributed Computing Systems*, pages 134–141, Paris, France, May 1990.

[23] Thomas J. LeBlanc, John M. Mellor-Crummey, and Robert J. Fowler. Analyzing parallel program executions using multiple views. *Journal of Parallel and Distributed Computing*, 9:203–217, 1990.

[24] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, pages 29–39, September 1991.

[25] Joseph L. Sharnowski and Betty H. C. Cheng. A top-down approach to visualization-based debugging of parallel programs, November 1994. Presented at the poster session of *Supercomputing '94*.

[26] Joseph L. Sharnowski and Betty H. C. Cheng. A formally-based expected behavior model for parallel program visualizations, April 1994. Submitted for publication.

[27] International Organization for Standardization, IS 8807. LOTOS: A formal description technique based on the temporal ordering of observational behavior, 1989.

[28] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987.

[29] Joseph L. Sharnowski, Betty H. C. Cheng, and Mark V. LaPolla. Mapping formal specifications to parallel program visualizations. In *Proceedings of Minnowbrook Workshop on Software Engineering for Parallel Computing*, pages 29–34, Minnowbrook Conference Center, New York, August 10-13, 1992.

[30] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors*, volume 1. Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[31] Luigi Logrippo, Mohammed Faci, and Mazen Haj-Hussein. An introduction to LOTOS: learning by examples. *Computer Networks and ISDN Systems*, 23:325–342, 1992.

[32] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification.* Springer-Verlag, Berlin, 1985.

[33] Robin Milner. A calculus of communicating systems. In *Lecture Notes in Computer Science 92.* Springer-Verlag, 1980.

[34] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, Englewood Cliffs, New Jersey, 1985.

[35] Mazen Haj-Hussein and Luigi Logrippo. Specifying distributed algorithms in LOTOS. To appear in Revue reseaux et informatique repartie.

[36] Adam Beguelin, Jack Dongarra, Al Geist, Robert Manchek, and Vaidy Sunderam. A users' guide to PVM: Parallel Virtual Machine. Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, July 1991.

[37] Alan George, Michael T. Heath, and Joseph Liu. Parallel Cholesky factorization on a shared-memory multiprocessor. *Linear Algebra and Its Applications*, 77:165–187, 1986.

[38] A. Gaber Mohamed, Geoffrey C. Fox, Gregor von Laszewski, Manish Parashar, Tomasz Haupt, Kim Mills, Ying-Hua Lu, Neng-Tan Lin, and Nang-kang Yeh. Applications benchmark set for Fortran-D and High Performance Fortran. Technical Report SCCS 327, Northeast Parallel Architectures Center, Syracuse University.

[39] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1994.

[40] Jason Gait. A probe effect in concurrent programs. *Software – Practice and Experience*, 16(3):225–233, March 1986.

[41] Barton P. Miller and Jong-Deok Choi. Breakpoints and halting in distributed programs. In *8th International Conference on Distributed Computing Systems*, pages 316–323, San Jose, California, 1988.

[42] Yoshifumi Manabe and Makoto Imase. Global conditions in debugging distributed programs. *Journal of Parallel and Distributed Computing*, 15:62–69, 1992.

[43] Dieter Haban and Wolfgang Weigel. Global events and global breakpoints in distributed systems. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences, Volume II, Software Track*, pages 166–175, January 1988.

[44] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[45] Sue Utter-Honig and Cherri M. Pancake. Graphical animation of parallel Fortran programs. In *Proceedings of Supercomputing '91*, pages 491–500, Albuquerque, New Mexico, November 1991.

[46] Cherri M. Pancake. Customizable portrayals of program structure. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 64–74, San Diego, California, May 1993.

[47] Alva L. Couch. Seecube user's manual. Technical report, Tufts Univ. Dept. of Computer Science, 1987.

[48] Jack Dongarra, Orlie Brewer, James Arthur Kohl, and Samuel Fineberg. A tool to aid in the design, implementation, and understanding of matrix algorithms for parallel processors. *Journal of Parallel and Distributed Computing*, 9:185–202, 1990.

[49] Daniel A. Reed, Ruth A. Aydt, Tara M. Madhyastha, Roger J. Noe, Keith A. Shields, and Bradley W. Schwartz. The Pablo performance analysis environment. Technical report, Department of Computer Science, University of Illinois, 1992.

[50] Allen D. Malony, David H. Hammerslag, and David Jablonowski. Traceview: A trace visualization tool. *IEEE Software*, pages 19–28, September 1991.

[51] Mark V. LaPolla. Towards a theory of abstractions and visualizations for debugging massively parallel programs. In *Hawaii International Conference on System Sciences-25*, 1992.

[52] John V. Guttag, James J. Horning, and Jeannette M. Wing. Larch in five easy pieces. Technical Report 5, Digital Equipment Corporation Systems Research Center, July 1985.

[53] Jeannette M. Wing. Writing Larch interface language specifications. *ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[54] John V. Guttag, Ellis Horowitz, and David R. Musser. Abstract data types and software validation. *Communications of the ACM*, 21(12):1048–1064, December 1978.

[55] John V. Guttag, James J. Horning, and Andrés Modet. Report on the Larch shared language: Version 2.3. Technical Report 58, Digital Equipment Corporation Systems Research Center, April 1990.

[56] M.L. Brady, K.K. Jung, M.V. LaPolla, H.T. Nguyen, R. aghavan, and R. Subramonian. The assignment problem on parallel architectures. In *The 1st DIMACS Intl. Algorithm Implementation Challenge: Problm Definitions and Specifications*, 1992.

[57] Thinking Machines Corp., Cambridge, Massachusetts. *Paris Reference Manual*, version 6.0 edition, Feb. 1991.

[58] Mark Vincent LaPolla, Joseph L. Sharnowski, Betty H. C. Cheng, and Kevin Anderson. Using formal specifications to generate visualizations of data parallelism. Technical Report CPS-92-05, Michigan State University, July 1992.

[59] Robert Cooper and Keith Marzullo. Consistent detection of global predicates. In *Proc. of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 167–174, 1991.

[60] R. Curtis and L. Wittie. BugNet: A debugging system for parallel programming environments. In *Proc. of 1982 Distributed Computing Systems*, pages 394–399, 1982.

[61] Steve Sistare, Don Allen, Rich Bowker, Karen Jourdenais, Josh Simons, and Rich Title. A scalable debugger for massively parallel message-passing programs. *IEEE Parallel & Distributed Technology*, pages 50–56, Summer 1994.

[62] Richard J. LeBlanc and Arnold D. Robbins. Event-driven monitoring of distributed programs. In *Proc. 5th International Conference on Distributed Computing Systems*, pages 515–522, May 1985.

[63] Alfred A. Hough and Janice E. Cuny. Belvedere: Prototype of a pattern-oriented debugger for highly parallel computation. In *1987 International Conference on Parallel Processing*, pages 735–738, 1987.

[64] Alfred A. Hough and Janice E. Cuny. Initial experiences with a pattern-oriented parallel debugger. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distr. Debugging*, pages 195–205, January 1989.

[65] Peter Bates. Debugging heterogeneous distributed systems using event-based models of behavior. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 11–22, January 1989.

[66] Janice Cuny, George Forman, Alfred Hough, Joydip Kundu, Calvin Lin, Lawrence Snyder, and David Stemple. The Ariadne debugger: Scalable application of event-based abstraction. In *Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–95, San Diego, California, May 1993.

[67] Vincent A. Guarna, Jr., Dennis Gannon, David Jablonowski, Allen D. Malony, and Yogesh Gaur. Faust: An integrated environment for parallel programming. *IEEE Software*, pages 20–27, July 1989.

[68] J.J. Dongarra and D.C. Sorensen. SCHEDULE: Tools for developing and analyzing parallel fortran programs. In L. Jamieson, D. Gannon, and R. Douglass, editors, *The Characteristics of Parallel Fortran Programs*, pages 363–394. The MIT Press, Cambridge, MA, 1987.

[69] M. Zimmermann, F. Perrenoud, and A. Schiper. Understanding concurrent programming through program animation. In *Proc. of 19th SIGCSE Technical Symposium on Computer Science Education*, pages 27–31, 1988.

[70] David Socha, Mary L. Bailey, and David Notkin. Voyeur: Graphical views of parallel programs. *SIGPLAN Notices*, 24(1):206–215, January 1989.

[71] John T. Stasko and Eileen Kraemer. A methodology for building application-specific visualizations of parallel programs. *Journal of Parallel and Distributed Computing*, 18:258–264, 1993.

[72] Gruia-Catalin Roman and Kenneth C. Cox. A declarative approach to visualizing concurrent computations. *IEEE Computer*, pages 25–36, October 1989.

[73] Gruia-Catalin Roman and Kenneth C. Cox. Declarative visualization in the shared dataspace paradigm. In *1989 IEEE 11th Intl. Conf. on Software Engineering*, pages 34–43, 1989.

APPENDIX

APPENDIX: LOTOS Specification of the Parallel Computing Environment

In this appendix, we provide the full LOTOS specification for the parallel computing environment that was discussed in Section 2.1.2. The process definitions for *Host_Node* and *Worker_Node* are intentionally left blank, since their behavior depends on the particular program that is being specified.

```
specification Message-Passing_Program(TOTAL_WORKERS:Nat): noexit
(* List of Program-Specific Variables should be added to the parameter list *)

library NaturalNumber endlib

behavior
   (
      Host_Node[send,recv](TOTAL_WORKERS)
         |||
      All_Worker_Nodes[send,recv](TOTAL_WORKERS,TOTAL_WORKERS)
   )
      ||
   (
      All_Host_to_Worker_Channels[send,recv](TOTAL_WORKERS)
         |||
      All_Worker_to_Worker_Channels[send,recv](TOTAL_WORKERS,
                                 TOTAL_WORKERS)
   )
where
   process Host_Node[send,recv](NUM_WRKRS:Nat): exit :=
   (
      (* Fill in definition for Host Node *)
   )
   endproc (* Host_Node *)

   process All_Worker_Nodes[send,recv](NUM_WRKRS,NodeCtr:Nat): noexit :=
      [NodeCtr > 0] →
      (
         Worker_Node[send,recv](NUM_WRKRS,NodeCtr)
            |||
         All_Worker_Nodes[send,recv](NUM_WRKRS,NodeCtr - 1)
```

```
   )
where
   process Worker_Node[send,recv](NUM_WRKRS,MY_NUM:Nat): exit :=
   (
      (* Fill in definition for Worker Node *)
   )
   endproc (* Worker_Node *)
endproc (* All_Worker_Nodes *)

process All_Host_to_Worker_Channels[send,recv](NodeCtr:Nat): noexit :=
   [NodeCtr > 0] →
   (
      Host_to_Worker_Channel[send,recv](NodeCtr,0)
         |||
      Host_to_Worker_Channel[send,recv](0,NodeCtr)
         |||
      All_Host_to_Worker_Channels[send,recv](NodeCtr - 1)
   )
where
   process Host_to_Worker_Channel[send,recv](Sender,Rcvr:Nat): noexit :=
      (* Replace "Message" with the name of an appropriate sort *)
      send !Sender !Rcvr ?Msg:Message;
      recv !Sender !Rcvr !Msg;
      Host_to_Worker_Channel[send,recv](Sender,Rcvr)
   endproc (* Host_to_Worker_Channel *)
endproc (* All_Host_to_Worker_Channels *)

process All_Worker_to_Worker_Channels[send,recv](NodeCtr,
                              NUM_WRKRS:Nat): noexit :=
   [NodeCtr > 0] →
   (
      Single_Worker_Channels[send,recv](NodeCtr,NUM_WRKRS)
         |||
      All_Worker_to_Worker_Channels[send,recv](NodeCtr - 1,NUM_WRKRS)
   )
where
   process Single_Worker_Channels[send,recv](S_Node,R_Ctr:Nat): noexit :=
      (* Do not create a self-loop channel *)
      [R_Ctr > 0 ∧ S_Node ≠ R_Ctr] →
      (
         Worker_to_Worker_Channel[send,recv](S_Node,R_Ctr)
            |||
         Single_Worker_Channels[send,recv](S_Node,R_Ctr - 1)
      )
      [ ]
```

```
        [R_Ctr > 0 ∧ S_Node = R_Ctr] →
        (
            Single_Worker_Channels[send,recv](S_Node,R_Ctr - 1)
        )
    where
        process Worker_to_Worker_Channel[send,recv](Sender,
                                    Rcvr:Nat): noexit :=
            (* Replace "Message" with the name of an appropriate sort *)
            send !Sender !Rcvr ?Msg:Message;
            recv !Sender !Rcvr !Msg;
            Worker_to_Worker_Channel[send,recv](Sender,Rcvr)
        endproc (* Worker_to_Worker_Channel *)
    endproc (* Single_Worker_Channels *)
endproc (* All_Worker_to_Worker_Channels *)

endspec (* Message-Passing_Program *)
```