Debugging Parallel Programs Using Abstract Visualizations*

Alfred A. Hough

COINS Technical Report 91–53
September 1991

Department of Computer and Information Science
University of Massachusetts
Amherst, Massachusetts 01003

*The Parallel Programming Environments Project at the University of Massachusetts is supported by the Office of Naval Research under contract N000014-84-K-0647 and by the National Science Foundation under grants DCR-8500332 and CCR-8712410.

DEBUGGING PARALLEL PROGRAMS USING ABSTRACT VISUALIZATIONS

A Dissertation Presented

by

ALFRED ANDREW HOUGH

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1991

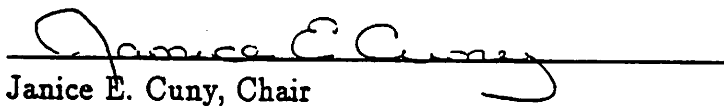
Department of Computer and Information Science

© Copyright by Alfred Andrew Hough 1991
All Rights Reserved

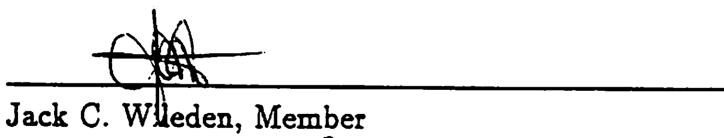
DEBUGGING PARALLEL PROGRAMS USING ABSTRACT VISUALIZATIONS

A Dissertation Presented
by
ALFRED ANDREW HOUGH

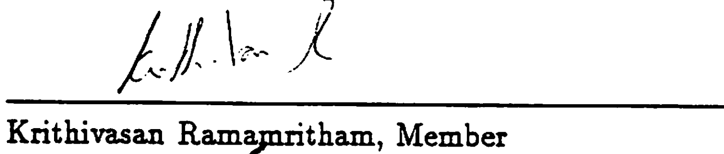
Approved as to style and content by:



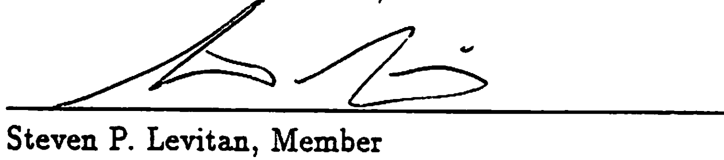
Janice E. Cuny, Chair



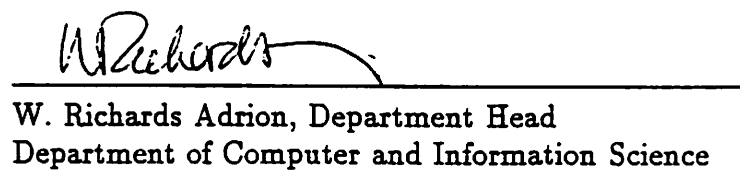
Jack C. Wileden, Member



Krithivasan Ramamritham, Member



Steven P. Levitan, Member



W. Richards Adrion, Department Head
Department of Computer and Information Science

ACKNOWLEDGMENTS

There are a number of people who helped bring this dissertation about. First and foremost, I am indebted to Jan Cuny for her knowledge and guidance. Writing a dissertation is an act of apprenticeship and I've been privileged to work with a great teacher. Her standards of research and scholarship have given me a goal to strive for in my professional life.

I would also like to thank the rest of my committee for their efforts: Jack Wilden and Krithi Ramamritham for their patient reading of the manuscript, Steve Levitan for his belief in the ideas even when I had trouble explaining them.

The company of friends and colleagues has made my time at UMASS pass quickly. Thanks especially to Duane Bailey and Larry Lefkowitz for their wit and encouragement, and to Bill Verts for being there year in and year out. Thanks also to Bruce Leban for his insights into TeX.

Finally, thanks to those whose unwavering support kept me going: my parents, Alfred and Yanina Hough, and my wife, Cynthia Thomson.

ABSTRACT

DEBUGGING PARALLEL PROGRAMS USING ABSTRACT VISUALIZATIONS

SEPTEMBER 1991

ALFRED ANDREW HOUGH, B.S., UNIVERSITY OF LOWELL
M.S., UNIVERSITY OF MASSACHUSETTS
PH.D., UNIVERSITY OF MASSACHUSETTS
Directed by: Professor Janice E. Cuny

This dissertation addresses problems in the visualization and debugging of asynchronous, highly parallel programs. Its primary contributions are: (1) a technique that uses abstraction to improve the visualization of parallel program behavior, and (2) an improved method of describing behavior at an abstract level.

Highly parallel, asynchronous, MIMD programs are difficult to debug. Traditional state-based, break/examine debugging techniques are successful in the sequential domain but parallel systems are not easily understood in terms of state transitions: multiple processes result in potentially overwhelming state spaces and asynchrony makes it difficult to predict transitions. Instead, highly parallel programs are best understood in terms of the interprocess flow of data and control. For code running on nonshared memory architectures, this flow is often very structured: nearly homogeneous, low grain processes communicating frequently across regular networks result in logically regular patterns of communication. In order to debug highly parallel programs, we believe that the programmer must be able to determine the extent to which these intended *logical patterns* of communication actually occur during execution.

Simple visualizations of parallel programs are not readily understandable because the independent asynchronous execution of processes can cause logically concurrent actions to be displayed at widely distant times and logically ordered actions to be displayed concurrently. Our approach is to construct *abstract* animations, in which user-defined, logically related behaviors are seen as distinct visual units. In these abstract animations the behavior of the program is easier to understand and bugs are easier to find.

Abstract animations rely on the ability of the user to describe abstract behavior. Current languages for describing abstract behavior are not well suited for describing errorful behavior in highly parallel programs. We introduce a new language for describing abstract behavior that more precisely specifies the behavior of parallel programs.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS iv

ABSTRACT v

LIST OF FIGURES x

Chapter

1. INTRODUCTION 1

1.1 Issues and Techniques in Parallel Debugging 2
1.2 Assumptions and Approach 6
1.3 Contributions 9
1.4 Outline of the Dissertation 11

2. APPROACHES TO PARALLEL PROGRAM DEBUGGING 12

2.1 Sequential Debuggers 12
2.2 Program Animation Environments 14
2.3 Analyzing, Monitoring, and Debugging Parallel Programs 16

2.3.1 Analysis of Parallel Programs 17
2.3.2 Performance Monitoring 19
2.3.3 Interactive Distributed Debuggers 21
2.3.4 Reduction 24
2.3.5 Database Approaches 26
2.3.6 Modeling Approaches 27
2.3.7 Hybrid Approaches 32

2.4 Conclusions 32

3. BELVEDERE 34

3.1 Belvedere 34
3.2 Using Primitive Event Animations 45
3.3 Abstract Animations 48
3.4 Problems with Abstract Animations 54
3.5 Summary and Conclusions 56

4. ABSTRACT VISUALIZATIONS . . . 59

4.1 Introduction . . . 59
4.2 Orderings between Abstract Events . . . 64
4.3 Reordering Transformations and Perspective Views . . . 68
4.4 Algorithm for Computing Perspective Views . . . 72

4.4.1 Example of a Full Perspective . . . 76
4.4.2 Example of a Partial Perspective . . . 78
4.4.3 Example of a Recursive Perspective . . . 79

4.5 Examples of the Utility and Generality of Perspective Views . . . 81

4.5.1 Process-Time Graphs . . . 82
4.5.2 User-Directed Animations . . . 86
4.5.3 Automatic Animations . . . 88

4.6 Conclusions . . . 92

5. MODELING . . . 96

5.1 Background . . . 97

5.1.1 Comparing Modeling Languages . . . 99

5.2 PEDL: Parallel Event Definition Language . . . 110

5.2.1 Language Operators . . . 111
5.2.2 Modeling with Perspectives . . . 112

5.3 Implementing PEDL . . . 114

5.3.1 Parallel Automata . . . 115
5.3.2 Translation . . . 119
5.3.3 Interpretation of Nondeterministic Parallel Automata . . . 120
5.3.4 Complexity of Nondeterministic Parallel Automata . . . 128
5.3.5 Compilation to Deterministic Automata . . . 130

5.4 Examples . . . 132

5.4.1 Traveling Salesman Problem . . . 132
5.4.2 Median Filter . . . 140

5.5 Summary and Conclusions . . . 141

6. DEBUGGING . . . 143

6.1 Case Studies . . . 143

6.1.1 Traveling Salesman . . . 144

6.1.2 Median Filter 146
6.1.3 Dictionary Search 147
6.1.4 Dynamic Programming 150
6.1.5 Fast Fourier Transform 155
6.1.6 Connected Components 157

6.2 Conclusions 160

7. CONCLUSION 162

7.1 Summary and Contributions 162
7.2 Future Research 164

APPENDICIES

A. COMPLEXITY OF THE PERSPECTIVE ALGORITHM 166
B. DETAILS OF THE DICTIONARY SEARCH EXAMPLE 170

BIBLIOGRAPHY 180

LIST OF FIGURES

Figure Page

1.1 Expected phases of communication in a hypercube program 7

1.2 Possible snapshots from animation of cube program 7

1.3 Error in the second logical communication phase 9

1.4 Debugging system overview 10

2.1 Process-Time graph 21

3.1 Events used in Belvedere 35

3.2 Information flow in Belvedere 36

3.3 Supported object references by event type 39

3.4 Events from an execution 42

3.5 Closed event set after selection of message-send event 43

3.6 Successive communication phases in the bitonic sort 45

3.7 Deadlocked version of an iterative relaxation program 46

3.8 Complex communication pattern in the median filter program 47

3.9 Median Filter: Traced animation of messages sent from Process e22 . . 47

3.10 Wavefront Matrix Multiply 48

3.11 Primitive event animation of the Matrix Multiply 49

3.12 Belvedere with an abstract event recognizer 50

3.13 EDL model of a message transmission 51

3.14 EDL model of an inner product step in the Matrix Multiply program . 53

3.15 Unexpected communication patterns in the Matrix Multiply 53

3.16 EDL model of a phase communication 57

4.1 Belvedere with reordering algorithm 73
4.2 SOR mesh 82
4.3 Primitive communication on a process-time graph for the SOR program 83
4.4 Logical communication on process-time graph from the SOR algorithm 84
4.5 Primitive communication on process-time graph from the Median Filter algorithm 85
4.6 Logical communication on process-time graph from the Median Filter algorithm 86
4.7 Logically consecutive time steps in the lives of sharks and fishes 87
4.8 Actual execution of the Sharks and Fishes program 88
4.9 Primitive event animation of the SOR program 89
4.10 Successive overrelaxation with abstract events 90
4.11 Low level communication behavior of the FFT 91
4.12 Abstract communication behavior of the FFT 91
4.13 Two snapshots from a standard visualization of an even row swap in the Gridsort program 93
4.14 Two snapshots from a perspective visualization of an even row swap in the Gridsort program 94
5.1 Hierarchical modeling 98
5.2 Modeling sequential program control constructs 100
5.3 Parsing vs. pattern Extraction 101
5.4 Model whose recognition is sensitive to process speed 104
5.5 A matching of repetitive behavior that is sensitive to process speed . . 105
5.6 A behavior represented as a graph and as edge expressions 109
5.7 A model and an associated predecessor automata that implements the model 110
5.8 Communication phases on an application running on a small grid . . . 112
5.9 Sequential behavior (a) and parallel behavior (b) 117

5.10 Converting PEDL expressions to Nondeterministic Parallel Automata . 121

5.11 EDL model of the first phase behavior of the Traveling Salesman program 133

5.12 Better EDL model of the first phase behavior of the Traveling Salesman program 135

5.13 PEDL model of the first phase behavior of the Traveling Salesman program 136

5.14 Alternate PEDL model of the first phase behavior 137

6.1 Traveling Salesman 145

6.2 Median Filter algorithm 148

6.3 Snapshots of Belvedere's animation of the Dictionary Search 151

6.4 Dynamic Programming 153

6.5 Communication between columns one and three in phase two of the Fast Fourier Transform 156

6.6 Extraneous communication during phase two of the Fast Fourier Transform. 156

6.7 Connected Components 159

6.8 Connected Components 159

B.1 Primitive event animation of the Dictionary Search 171

B.2 EDL model of an abstract Query behavior in the Dictionary Search . . 172

B.3 Abstract event animation of a single query in the Dictionary Search . . 173

B.4 Process code for the Dictionary Search, Part 1 175

B.5 Process code for the Dictionary Search, Part 2 176

B.6 Process code for the Dictionary Search, Part 3 177

B.7 Process code for the Dictionary Search, Part 4 178

B.8 Revised process code for the Dictionary Search 179

CHAPTER 1

INTRODUCTION

Parallel programs are hard to debug. Unlike sequential programs which consist of a single process, parallel programs may consist of hundreds or even thousands of asynchronous, interacting processes. Understanding the behavior of such a system is a potentially overwhelming task; yet that is precisely what we must do during debugging.

Existing debugging tools work for single processes but provide little help in understanding the interrelated behavior of multiple processes. This dissertation presents a visualization-based approach to debugging that focuses on understanding the collective behavior of processes. It is based on the observation that programmers often think of parallel program behavior not in terms of individual actions but in terms of patterns of related actions. We provide an *abstraction* technique for identifying groups of related actions and a *reordering* technique for coherently interpreting program behavior in terms of these abstractions. The reordered, abstract visualizations produced with these techniques make it easier to understand behavior and easier to detect bugs.

Section 1.1 describes issues in parallel debugging. Section 1.2 presents our approach and our assumptions. Section 1.3 details our contributions and Section 1.4 outlines the remainder of the dissertation.

1.1 Issues and Techniques in Parallel Debugging

Parallel debuggers must confront all of the issues faced by sequential program debuggers as well as several difficult problems introduced by parallelism. Problems specific to parallelism include:

- *Lack of a Common Clock.* Asynchronous machines do not usually have a global clock and thus it is not always possible to order events that occurred on different processes. Understanding the ordering of events, however, is often necessary during debugging.

  One approach to this problem is to use logical time instead of physical time to order events. Using the ordering of events implied by intraprocess sequencing and interprocess communication, Lamport[48] defined a system of software clocks that provides a logical timestamp consistent with the partial ordering of events. Logical timestamps based on this ordering can be computed and supplied by the debugger.

- *Nondeterminism.* Nondeterminism in parallel programs results from the existence of race conditions. Race conditions arise when parallel activities access a common resource without synchronization and thus can occur in any order. In shared memory systems, nondeterminism occurs in the order of accesses to common memory; in message passing systems, nondeterminism occurs in the order that messages are read at a process. Nondeterminism implies that the path taken through a program is a function not only of the input data but also of external factors such as the load on the machine. Nondeterminism makes it harder for a programmer examining the states of the processes in parallel program to decide whether or not those states belong to a correct computation. The programmer must decide which of the many valid, dif-

ferent computations for the given input occurred and whether the examined states are valid for that computation.

Nondeterminism is addressed with both analysis and debugging techniques. Static analysis techniques[85, 24] can identify the points of nondeterminism, allowing the user to ensure the nondeterminism was intended. Debugging techniques[13, 82] provide user control over the resolution of races, allowing the user to explore different possible executions for a given input.

- *Irreproducibility.* Nondeterminism means that two executions of the same program on the same input may not yield the same results. Programmers of sequential machines frequently reexecute programs on the same input to examine an earlier state in the computation. Irreproducibility prohibits the use of this technique on nondeterministic parallel programs because the re-execution may take a different path.

  Irreproducibility is addressed through two techniques: logging and check-pointing. Logging saves the outcome of races so that any program state can be regenerated deterministically. Regeneration is done under the control of a debugger which reexecutes the program and forces the outcome of races to match those recorded in the log. Checkpointing, which saves the state of the program at periodic intervals, can be used in conjunction with logging so that program executions can be regenerated from points other than the beginning of the program. Checkpointing can also be used without logging; in this use the program can not be regenerated, but earlier states, stored in the checkpoints, can be accessed without reexecuting the program.

  The extra computation required to implement checkpointing and logging techniques may change the path taken through a program. This issue is discussed next.

- *Nontransparency.* Transparency is the ability of a debugger to examine a program execution without changing it. Sequential program debuggers are transparent in two respects. First, they can monitor the state of a running program without changing the computation. Second, a debugger can *start and stop* the execution without changing it. Parallel debuggers for asynchronous MIMD (multiple instruction, multiple data) programs usually possess neither of these properties. Parallel debuggers can not transparently access the state of a running program because any inserted code or additional computation on the machine can modify the outcome of a race in the target program. This has been called the *probe effect*[31]. Second, processors on asynchronous machines can not be stopped without affecting the computation. An individual processor can only be stopped at an instruction boundary but processors are not synchronized; while one processor is at an instruction boundary another may be in the middle of an instruction. Halting each processor at its next instruction boundary affects the relative progress of the processors and may modify the outcome of a race.

The probe effect admits no software solution but can be addressed using a massive amount of hardware. Plattner[69], considering a single process on a sequential machine, demonstrates that *any* monitoring activity executing on the same processor or using the same resources can introduce *unbounded* delay into the target process. On a parallel machine the delay introduced by monitoring can change the outcome of a race. Plattner constructs a minimal system for transparent monitoring that relies on a transparent hardware tap to collect information, a second faster machine to process the information, and a queue to decouple the machines. To transparently monitor a parallel computer, this monitoring configuration would need to be replicated for all processors in the parallel computer. A transparent parallel debugging system,

employing essentially a duplicate parallel machine, is described by Rubin, Rudolph, and Zernik [74].

Software solutions to transparency seek to minimize intrusion upon the target program rather than trying to eliminate it. Techniques such as logical time, low-overhead logging, and user control of race resolution will be discussed in Chapter 2.

- *Very large state spaces.* To understand the state of parallel program it may be necessary to examine the state of hundreds or thousands of processors. Sequential debugging techniques that only allow access to a single variable in a single process at a time are clearly inadequate.

  A number of techniques have been applied to the huge amount of state generated by parallel programs. Databases are used to store information which is retrieved using query languages[79] or temporal logic[38]. Visualization is used to transfer large amounts of data to the user; filtering[63, 45, 10], clustering[63, 20, 10], and abstraction[6, 10, 60, 43], are used to reduce the data to be examined.

- *Asynchrony.* Understanding the actions of a parallel program is complicated by asynchronous execution. Independent, asynchronous processes performing related actions do not execute in lockstep. A user observing the actions of a process can make no assumptions about the actions of other processes. To understand the behavior of the whole program each process must be examined individually.

  Asynchrony can be removed from the visualization of program execution by displaying events in logical time instead of physical time[80, 42]. In logical time, related events on different processes can be observed simultaneously, allowing the user to easily understand the actions of multiple processes.

1.2 Assumptions and Approach

We are primarily interested in the issues of *large state spaces* and *asynchrony*. We ignore the issues of nondeterminism and nontransparency and we use existing techniques to address the problems of a lack of a common clock and irreproducibility. We have adopted a postmortem approach to debugging that uses trace files to collect information about program behavior. The traces allow us to reconstruct a software clock to order events. We use the same trace files to circumvent the problem of irreproducibility; to reproduce a program behavior we simply reexamine the trace file.

We consider only nonshared memory, MIMD, asynchronous parallel programs. Programs in this computational model consist of independently programmed processes, each with its own private memory, that communicate only through message passing. We expect that this model will persist because it gives the programmer the greatest control over data distribution and communication costs; there is evidence that use of this model results in more efficient programs on both distributed memory and shared memory machines[56, 51].

We limit our attention to debugging the "parallel" parts of programs; that is we attempt to find bugs that occur because of (or affect) the interactions between processes. Sequential debugging technology, extended with techniques such as checkpointing and logging, is adequate for finding those bugs that occur entirely within a process.

We help the user understand complex, asynchronous behavior by combining two techniques, *visualization* and *abstraction*. Visualization allows the user to quickly absorb a large amount of information. Abstraction reduces information by combining many low level actions into a single high level action.

To illustrate our approach, consider for example an iterative application running on a hypercube in which nodes communicate in a sequence of phases, each phase crossing a different dimension of the cube as shown in Figure 1.1. Arrows represent a message that has been sent but not yet received. If the processes exe-

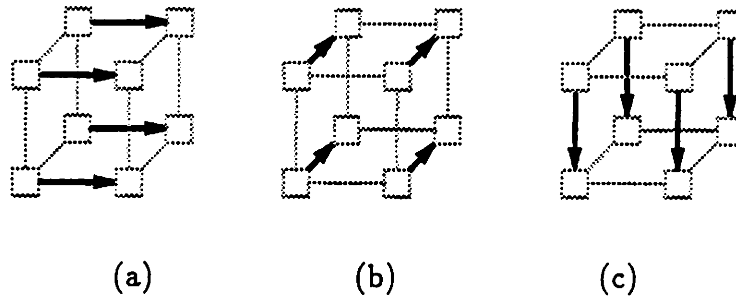


Figure 1.1 Expected phases of communication in a hypercube program.

cute independently without synchronization, an animation system might produce the sequence of snapshots shown in Figure 1.2. Initially, the processes commu-

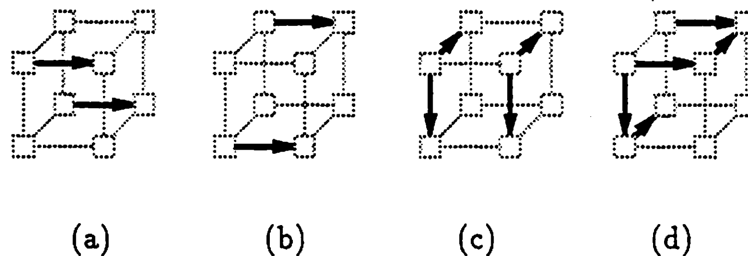


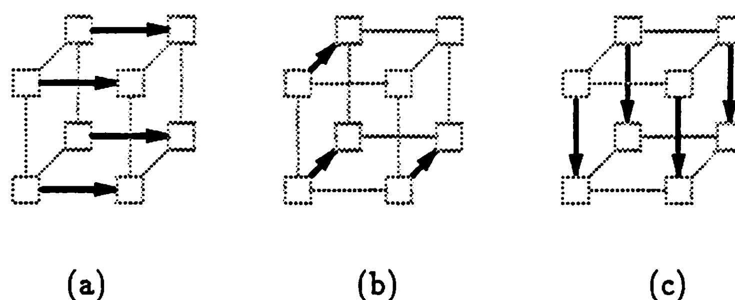
Figure 1.2 Possible snapshots from animation of cube program.

nicate at slightly different times within the same phase (Figure 1.2a-b). After a number of iterations, however, their execution becomes so skewed (Figure 1.2c-d) that some of the processes are still completing communications from the second or third phase of an iteration while others have progressed to the first phase of the next iteration. It is doubtful that this series of snapshots would be meaningful to the programmer.

The problem is not with the program but rather that the visualization is at too low a level of abstraction. The programmer thinks of the communication across a dimension as a single abstract communication event but this abstract behavior can not be seen in the low-level visualization of the program. If we could display the communication behavior in terms of the user's own abstractions (in this case, three distinct communication phases), the program would become easier to understand and program errors become easier to find.

To display abstract behavior the user must first define abstract events. Modeling languages, in which an abstract event is defined as a group of primitive events, are used for this purpose. Existing modeling languages, however, are inadequate for modeling the behavior of highly parallel, asynchronous parallel programs. They are imprecise, in that it is difficult to describe a group of events in a way that is independent of the relative speed of processes, and they are not robust, in that it is difficult to describe a group of events in a way that is unaffected by program bugs. In this dissertation we address this problem by introducing a new modeling language that permits descriptions of behavior that are not affected by changes in process speed and that are more robust than existing languages.

How should we animate such abstract events? We would like to display abstract events so that all of the messages belonging to an abstract event are visually associated and distinct from messages belonging to other abstract events. This is difficult because abstract events may be concurrent, overlapping in both time and space. We have developed techniques that produce a coherent display by reordering the events prior to display to emphasize logical relationships. Figure 1.3 shows a high level animation of the cube algorithm, showing each of the communication phases at a unique time. In this animation we can see that one of the low-level communications in phase 2 did not occur. This missing communication error,



(a) (b) (c)

Figure 1.3 Error in the second logical communication phase.

easily visible when the program is viewed at a high level of abstraction, was very difficult to see when the program was viewed at a low level of abstraction.

Our goal then is to automatically produce comprehensible, high-level animations. In our approach, the user provides a specification of the abstract behavior of the program which logically groups low-level events. We then reorder the events to insure that abstract actions are presented as distinct visual units. We attempt to make this reordering consistent, that is, all processes execute the same set of actions in the same order as before. If this is not possible, we provide a means of viewing behavior using user-selected, partially consistent visualizations.

A debugging architecture using these techniques is illustrated in Figure 1.4. The user provides a model that is compiled and matched against events produced by parallel program. Matching groups related primitive events into single abstract events. The abstract behaviors are then reordered and animated. After examining the visualization, the user may change the visualization, refine the models, or view another part of the behavior.

1.3 Contributions

This dissertation addresses the problem of debugging highly parallel, asynchronous parallel programs by supporting the use of abstract visualizations. The contribu-

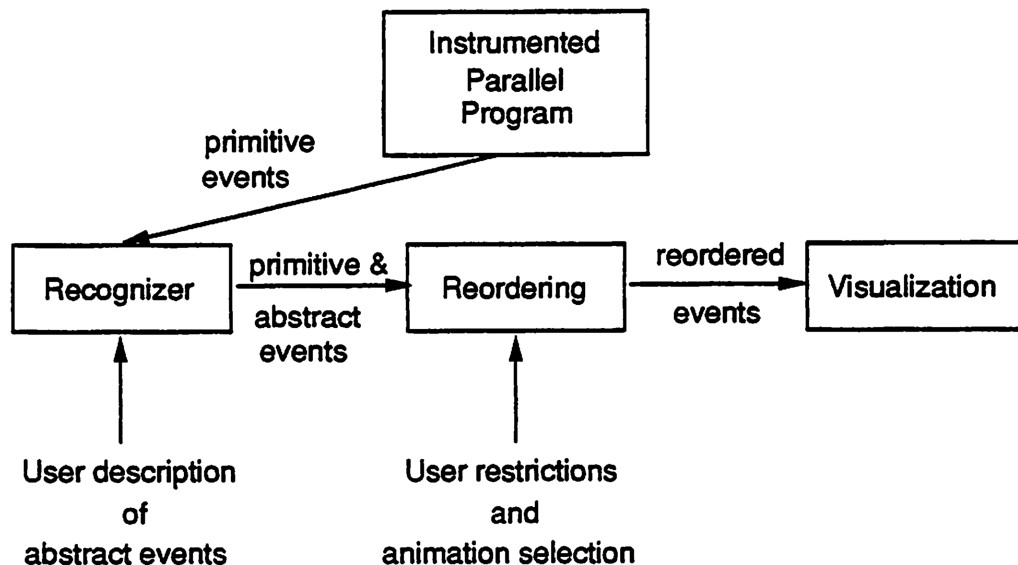


Figure 1.4 Debugging system overview.

tions of this work are (1) a technique that uses abstraction and reordering to improve the visualization of parallel program behavior, and (2) an improved method of describing behavior at an abstract level.

Visualization Techniques. Parallel programs have complex behaviors that are hard to understand. Abstract visualizations aid comprehension by displaying program behavior in familiar terms. We present a technique for automatically producing comprehensible, abstract visualizations of asynchronous parallel programs.

Modeling Techniques. To use abstraction, users must be able to describe the behavior of their program with a *modeling language.* Existing modeling languages are inadequate because they do not allow precise, unambiguous descriptions of the behavior of highly parallel programs. We introduce *PEDL*, a language that uses a notion of logical ordering to support more precise descriptions of behavior.

1.4 Outline of the Dissertation

Chapter 2 is an overview of the current status of parallel debugging with particular attention to those debuggers that address the complexity of parallel program behavior. Also reviewed are related techniques that help users understand program behavior.

Chapter 3 introduces Belvedere, a prototype debugger for testing visualization techniques for parallel programs.

Chapter 4 presents *perspective views*, a technique for generating abstract visualizations of parallel program behavior by reordering events prior to display. An efficient algorithm to compute perspective views is provided and the utility of the technique for different kinds of visualizations is demonstrated.

Chapter 5 presents an improved language for modeling the behavior of parallel programs.

Chapter 6 demonstrates the use of perspective views and modeling to debug programs. Several case studies of debugging and modeling are given.

Chapter 7 summarizes the results of this dissertation, and includes directions for future research and our conclusions.

CHAPTER 2

Approaches to Parallel Program Debugging

In this chapter we review techniques for debugging parallel programs. Our particular interest is the issue of understanding large programs with complex behaviors. Along with debuggers that address this issue we also review related techniques in static analysis, performance monitoring, visualization and modeling.

Other surveys of parallel debuggers have been published. McDowell and Helmbold have done a general survey[59]. Utter and Pancake have compiled an extensive bibliography[88] which they periodically update[89]. Pancake and Utter have also surveyed debugger visualization systems[67].

The next section introduces debugging in the context of sequential program debuggers. Section 2.2 reviews program animation. Section 2.3 describes parallel debuggers and work related to parallel program debugging, including static and dynamic analysis techniques and performance analysis.

2.1 Sequential Debuggers

The fundamental techniques of debugging were developed in the context of sequential programs. Nearly all debuggers for sequential programs use the *controlled-execution* paradigm. A controlled-execution debugger provides several basic capabilities: state examination, state modification, state monitoring, controlled execution and exception handling. With this paradigm the target program is executed under the control of the debugger. The target program can be stopped and restarted and the user can examine and modify values its address space. Several

features allow control over the execution of the program: *single-stepping* runs the target program for one step; *breakpointing* runs it until a particular statement is executed; and *tracing* runs it until the value of a variable meets a specified condition. A typical example of this class of debuggers is the Unix[1] debugger dbx[57].

Although a controlled-execution debugger can only examine the current state of a program, the technique of *cyclical debugging* provides a means to examine both past and future states. Future states are examined by continuing the execution and halting again; prior states are examined by reexecuting the program from the beginning until the desired state is reached. The key requirement for cyclical debugging is *reproducibility* — given the same input the program produces the same result. Nearly all sequential programs have reproducible behavior; the primary exception is real-time systems.

A significant advance in sequential debugging occurred with the development of *source-level* debugging. Early debuggers only allowed access to the bare machine state: users had to cope with machine addresses and the untyped, uninterpreted contents of memory cells. Source level debuggers allowed users to work with typed, symbolic variables and line numbers referring to the program source. At this higher level of abstraction, programs were easier to understand and debug.

Balzer's EXDAMS[9] is one the few sequential debuggers that uses neither the controlled execution nor the cyclic debugging paradigm. Instead EXDAMS records a complete record of all state-space transitions on a history file. With EXDAMS, a program can be reexecuted at selected speeds, forwards or backwards, under control of the recorded history. EXDAMS additionally provides sophisticated views to aid the examination of state. One such view is a *flowback analysis*, a tree-oriented display that shows how the current value of a variable was computed from the

[1] Unix is a trademark of AT&T.

values of other variables. Other displays include a scrolling variable map, dynamic data flow maps, and a source file display.

Integrated programming environments such as Interlisp[86] and PECAN[72] provide high level views that support debugging, coupled with views supporting other phases of program development. The program is executed under the control of the environment which supplies a variety of source level views into the stack, symbol tables, source files, data history, and session history. The integration and availability of information from other phases of development improves the user's ability to debug.

2.2 Program Animation Environments

Although program animation systems are not primarily debuggers, they are reviewed here because they aid in the understanding of program behavior. The acknowledged inspiration for these systems is Baeker's film *Sorting out Sorting*[5]. The film's animation associates data values with proportionally scaled sticks and associates visual transformations on the sticks with the fundamental operations of sorting. The film imparts an intuitive understanding of the standard sorting algorithms that is difficult to obtain by other means. *Sorting out Sorting* took three years to make; the systems described here provide tools that make program animations easier to create.

The most widely known animation kit is the BALSA[16, 15, 14] system developed by Brown and Sedgewick. In BALSA, the programmer annotates the code to identify interesting events. During animation, the program is run under the control of BALSA. Events are sent through *adapters* to *modelers* and *renderers*. Modelers use their own data structures to maintain an abstract representation of the program. Renderers produce a view from an abstract model. Several renderers can be used to produce different views of a model, and several models can be main-

tained for a program. Each modeler implicitly defines an input protocol; adapters, used between the program and its modelers, enhance reusability. BALSA has been successfully used for teaching algorithms at Brown University.

London and Duisberg[58] built an animation kit in Smalltalk using its model-view-controller (MVC) system. The integrated, interactive nature of the Smalltalk environment supports the quick creation of small animations in contrast to the large, polished animations created in the stand-alone BALSA environment. As in BALSA, screen updates are generated in response to interesting events in the algorithm. Interesting events are selected by the designer of the animation. Usually they are chosen to illustrate program invariants and how they are maintained. Roman and Cox[73] share this view of interesting events and advocate guidelines for displaying them: events related to safety properties such as invariants should be presented as stable patterns and events related to progress properties should be presented as evolving patterns.

Duisberg's Animus[23] investigated the use of a constraint based animation kit that modeled temporal constraints explicitly. Constraint satisfaction was used to drive the animation. Constraints were triggered from intercepted messages between objects. Duisberg was able to provide animations from these messages without annotation of the program. Animus also improved on systems using the Smalltalk MVC model by introducing a time-ordered queue of messages for communication between programs and views.

Stasko's Tango system[81] improves upon the BALSA system by separating the generation of interesting events from the running of the animation and by providing a language for describing animations. The target program and the animation system run in separate processes that communicate through message passing. Tango's language is based on four data types: *images*, *locations*, *paths* and *transitions*. Images and locations are used to hold the shapes and locations of graphical

objects; paths and transitions control changes in the objects. Tango supports a flexible mechanism based on regular expressions for binding sequences of interesting events to sequences of changes in objects. Stasko notes that local algorithm designers with access to Tango implemented animations prior to the completion of the algorithm so that the animations could be used to debug the program. He intends to further simplify the creation of animations to support this use of Tango.

Voyeur is a parallel program animation environment[80] that implements a BALSA-like architecture for building animations to debug parallel programs. It supports a hierarchy of animations to create visualizations at different levels of abstraction. Voyeur can describe animations at a variety of levels including the application level, the data structure level, and the architecture level. Information from each of these levels is useful in debugging.

2.3 Analyzing, Monitoring, and Debugging Parallel Programs

Flynn's taxonomy[28] broadly classifies parallel machines into two groups: Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD). In SIMD machines each processor executes the same instruction but does so on its own data. In MIMD machines each processor is independent and has its own program, state and local memory. Depending on the architecture, MIMD processors communicate with each other through an interconnection network or a shared memory. Debugging technology for parallel machines has largely focused on MIMD machines. All of the techniques reviewed here were developed for MIMD machines.

Analysis and monitoring techniques are reviewed, in Sections 2.3.1 and 2.3.2 respectively, because of their close relationship to debugging. Analysis techniques can be used to decide whether or not certain kinds of errors exist in a program.

Monitoring techniques must perform a data collection and analysis similar to that required by parallel debuggers.

Sections 2.3.3 through 2.3.7 describe parallel debugging systems. Many systems have been built to debug parallel programs but most address one or two issues in parallel debugging. We group debugging systems into five categories — interactive, reduction, database, modeling, and hybrid — on the basis of the dominant approach used by the debugger. Interactive approaches extend traditional sequential debuggers into parallel debuggers. Reduction approaches address nondeterminism by introducing determinism to get repeatable parallel program executions. Database approaches capture an event trace of the program and use database techniques to analyze the trace. Modeling approaches use abstraction to reduce the amount of detail needed to understand program behavior. Hybrid systems focus on the integration of two or more techniques.

2.3.1 Analysis of Parallel Programs

Analysis techniques are used to prove that certain kinds of errors, such as deadlocks or races, do not exist in a parallel program. Analysis techniques can be static, dynamic, or both. Static analysis techniques use only the text of the program. Dynamic analysis techniques use a trace file of events collected during execution. With one exception the following approaches are all static analysis techniques.

Data flow techniques are static analysis techniques that use fast, polynomial time algorithms to determine when and where values of variables might be used. Taylor and Osterweil[85] extended algorithms originally developed for sequential programs to a restricted class of parallel programs. Their algorithms can detect race conditions as well as standard data usage errors such as a reference to an undefined variable. Their algorithms cannot analyze a program in which the sets of processes that synchronize with each other is determined at run time.

A more extensive but more expensive alternative to data flow techniques are state based techniques. State based techniques attempt to compute all possible states of a parallel program as a graph and then analyze that graph for properties of the corresponding program. The first such algorithm was developed by Taylor[84] for Ada programs. It generates a graph of all possible *concurrency states* and transitions. Nodes represent the state of every task in the system; edges represent the execution of a single statement in a single task. Interesting properties of a program can be determined using this graph. For example, a race condition can be detected by searching for statements that can execute in parallel (as represented by the lack of a path containing edges that represent the executions of those statements) and that access a common variable. Potential deadlocks can be detected by searching for tasks that do not terminate properly, as represented by nodes that refer to active tasks but have no successors.

The primary problem with state based approaches is size: the number of states in a concurrency state graph is exponential in the number of tasks. Techniques have been developed that use more compact representations[3], or analyze the graph in pieces[84], but analysis of large programs is still quite difficult.

Avrunin, Dillon, Wileden and Riddle[4] developed a constrained expression formalism for reasoning about the behavior of distributed systems using events rather than states. The constrained expression formalism allows a programmer to ask questions about the behavior of a program without executing the program. From the source code, automatic translation rules are used to develop regular expressions that describe each task activity in terms of events. These expressions are then interleaved to yield an expression of system activity. A final phase filters out interleavings that do not obey the semantics of the language, resulting in an expression describing possible system behaviors.

This expression can be analyzed at design time to discover behavioral aspects of the system. The designer may wish to know, for example, if the system can ever deadlock or if the designed program maintains certain invariants. One implementation[91] uses a linear programming package to determine whether the behavior of interest is consistent with the behavior of the system.

Static analysis techniques in general suffer from an inability to detect infeasible paths in a program (that is, paths that will never be executed). As a result, static analysis techniques can report program errors that do not exist because they occur on infeasible paths. Detection of infeasible paths has been shown to be equivalent to the halting problem[19].

An approach which supplements static analysis with dynamic testing is presented by Allen and Padua[2] and Emrath and Padua[24]. Their approach uses dependency analysis techniques developed for parallelizing sequential FORTRAN programs to identify the existence of data races. Their approach combines static and dynamic analysis. Static analysis directly detects some data races and reports on the possible existence of others. Dynamic analysis is used on the areas indicated as potentially unsafe by static analysis. The dynamic analysis reads an event trace to determine when potentially unsafe memory accesses are occurring.

2.3.2 Performance Monitoring

Performance monitoring systems help users tune the performance of their programs. Their goal is to help find bottlenecks and suggest how the program might be modified so that it executes in less time. Performance monitoring systems for parallel programs must collect, filter and analyze large amounts of data. In doing so they address many of the same issues confronted by parallel debuggers. Here we present five such systems.

In the system by Miller, Macrander and Sechrest[63] *meter events* are associated with various system calls on each monitored process in the distributed system. Calls can be metered without changing the monitored program. When the program makes a metered call, a duplicate message is sent to a central filter process. Messages that match user-specified templates are stored. Post-mortem analysis routines interpret the traces created by the filter and generate performance statistics.

Several systems use this general architecture. In some systems a separate communication network is used for the messages generated by metered calls. A separate network increases the transparency of the performance monitor.

The Jade system from Joyce *et al.*[45] is an extensible architecture for monitoring distributed systems. Messages are routed to a local monitor before being sent to their destination. The monitor feeds several selectable output displays called *consoles*. Jade consoles can be constructed and plugged in at will; standard consoles include a textual trace display with simple pattern matching, a basic animation display, a statistics window and a deadlock detector. The authors propose a dynamic protocol checker console that would match patterns of expected message traffic against specified models.

Couch's Seecube[21] provides a variety of animations and communication views of computations on hypercubes. Given a log of message sends and receives, Seecube first constructs an approximate global timestamp for each event and orders the events accordingly. Seecube's animations currently support seven different predefined representations of hypercubes. Other features include displays of message values, process state and queue statistics. Seecube provides a simple facility for grouping processors for displaying large networks.

Fowler *et al.*[29] present a tool called Moviola in the context of a debugging toolkit. Moviola displays the behavior of a program on a process-time graph. A

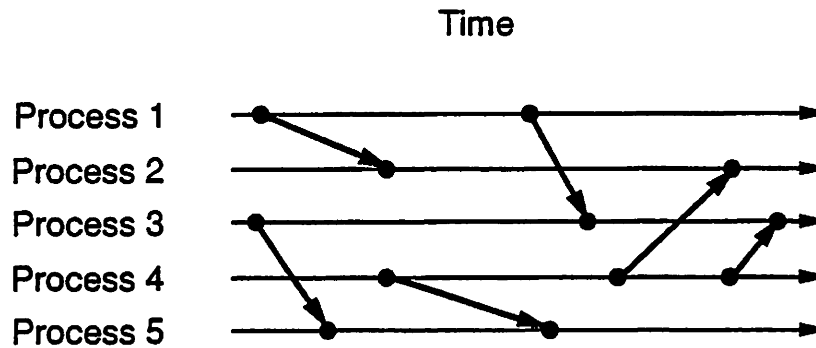


Figure 2.1 Process-Time graph.

process-time graph (not from Moviola) is shown in Figure 2.1. Events that occur on a given process are shown as nodes on a time line associated with the process and synchronizing relationships between processes are shown as edges between the nodes. Synchronizing relationships can be either accesses to shared variables or messages. Moviola features zooming, panning and scaling to view interesting portions of the graph.

The PIE system (Segall *et al.*[76, 55]) is a complete programming environment for parallel programs in a shared-memory environment. A substantial portion of this environment is devoted to performance monitoring. PIE inserts *taps* into the user's program at compile time to collect timing data. The user can also insert taps to monitor the values of variables and to generate arbitrary events for later analysis. The trace files generated by the taps are analyzed and several views can be generated. These views currently include process structure graphs, bar charts of execution time, dynamic invocation views and shared memory module usage.

2.3.3 Interactive Distributed Debuggers

The most straightforward approaches to parallel debugging are those that extend sequential debugging techniques. The basic controlled-execution, break/examine

sequential debugger is augmented with techniques to address the parallel aspects of the program.

Pdbx[77] is a parallel debugger produced by the Sequent Corporation for use on their multiprocessor. It is a parallel extension of the Unix dbx[57] debugger. Dbx provides the usual sequential debugging facilities for a single Unix process. Pdbx provides these capabilities for a multi-process program on a per process basis.

Chandy and Lamport present an algorithm[18] for halting distributed programs. A distributed program can not be halted transparently, but the authors show how it can be halted in a consistent state. An attempt to halt a distributed program may produce an inconsistent state because the messages in transit may not be recorded properly. Chandy and Lamport present a marker passing algorithm that guarantees that all messages are recorded at a consistent logical time. Miller and Choi[61] use this algorithm to define breakpoints in a distributed program.

Interactive distributed debuggers extend the standard sequential controlled-execution paradigm by adding commands to manipulate interprocess communication. The direct control of interprocess communication disrupts the relative message timing and severely affects debugger transparency. Some systems do not attempt transparency, giving the user control over message queues and scheduling and allowing the user to force the outcome of nondeterministic constructs. Helmbold and Luckham[39] present such a system for debugging Ada tasking programs. Their debugger builds a task state graph during execution. Each time a task changes state, the graph is analyzed for various kinds of deadness errors. The user debugs his program by single stepping at the granularity of task interaction. At a breakpoint the user can examine the task state graph.

There is general problem with run-time monitoring: information is not necessarily processed at the monitor in the order that it was sent, thus allowing the monitor to have an incorrect model of the monitored program. This greatly com-

plicates the user's interpretation of the task state graph and diminishes his ability to detect errors. The problem admits no simple solution.

The SPIDER system by Smith[78] implements control of messages through programmable demons that monitor message traffic. These demons have the ability to examine, interpret, and change interprocess communication. Associated with each demon is a boolean trigger expression constructed from predefined message fields such as *ProcessName* or *PortName*. When the expression becomes true the demon is activated. The use of demons provides a flexible and powerful method of intervention in program execution.

TAP by Gordon and Finkel[35] uses an active tracing facility to construct a timing graph at run time. A timing graph is a history of message traffic partially ordered using Lamport logical clocks. The user can examine the timing graph to determine the order of messages. The authors show that the overhead for the active tracing facility is less than ten percent.

The interactive debugger by Schiffenbaur[75] addresses the transparency problem directly by having processes operate in logical time defined by the debugger. His debugger is composed of a central process through which all communication is routed and *nub* processes that exist on each processor. The user interacts with the central process which can forward, destroy, delay or change each message that comes through. The nub processes handle forwarding of messages, maintenance of local time, and local process suspension. Because processes operate in logical time, the effect of delaying a message for examination can be limited by stopping all other processes at that logical time. Although an interactive debugger for a distributed system must necessarily interfere with the program being debugged, Schiffenbaur's scheme controls for the effects of intervention. It does so, however, at the cost of introducing a central bottleneck.

2.3.4 Reduction

The following approaches use sequential debugging technology but also address the issues of nondeterminism and nonrepeatability by adding determinism to the debugging activity.

The simplest such approach is by Brindle, Taylor and Martin[13], who advocate debugging Ada tasking programs using an interpreter on a uniprocessor system. The single processor requirement enforces a deterministic execution that removes the problems of nondeterminism and nonrepeatability. Their debugger allows control over the initialization of the scheduling algorithm, advancement of time and the choice of alternatives so the user can simulate nondeterministic choices. Using these features, alternative paths through the program can be examined.

Leblanc and Robbins' Radar debugger[50] is a post-mortem animating debugger that animates a simulated execution during replay. During execution events related to interprocess communication are collected and automatically stamped with a logical clock value. During replay events are animated at the appropriate logical time. Newly created processes are drawn on a blank part of the screen selected by the debugger and messages are depicted as boxes moving from the sender's output port to the receiver's input port. Radar allows the user to control the speed of the animation and to interact with message icons to obtain values. Because the program is not executing during replay, only those values stored during execution can be recovered.

A number of systems support the concept of *reverse execution* as a solution to the problem of nonrepeatability. These systems do not actually execute a program in reverse but instead provide access to earlier states of the computation through checkpointing or deterministic regeneration. Wittie's Bugnet[92] system logs all messages between processes and periodically checkpoints the system. During *replay*, execution is restarted from a selected checkpoint and messages are

delivered as they occurred in the original log. Instant Replay[52] by Leblanc and Mellor-Crummey reduces the logging overhead by saving only the relative order of process interactions on shared objects — no data values are saved. Since checkpoints are not used the program must always be replayed from the start state. Pan and Linton's Recap[66] operates in both shared-memory and message passing environments. Recap uses a novel checkpointing scheme that periodically forks and suspends a duplicate process. A full logging system saves each value read from shared memory or message. Igor[26] optimizes checkpoints in virtual memory systems by only writing out pages that have been written on since the last checkpoint. Igor does not use event logging and so only supports nondeterministic replay — executions restarted from a checkpoint may not necessarily follow the original path.

A nondeterministic replay is also used by Stone[82]. By logging events on a per process basis rather than a shared object basis logging overhead is reduced but some information is not captured. *Speculative replay* is used when the replayed execution differs from the original. Additional dependency information can be entered during speculative replay to control the path of execution. Stone uses diagrams of sequential code blocks connected by dependency arcs called *concurrency maps* to help users visualize possible interleavings of events.

Miller and Choi's PPD debugger[62] implements Balzer's flowback analysis of variables for shared memory parallel programs. In this technique, the debugger uses a minimal logging technique based on a semantic analysis of the program. The compiler divides the program into blocks. At the beginning of a block, code is inserted to save all the variables which might be read before the next logging point. At the end of a block, all variables which have been written since the last logging point are saved. The debugger is able to give a flowback analysis of any

variable by scanning the log and if necessary re-executing portions of the program that lie between logging points.

Re-execution techniques differ in how costly they are, how accurate the replay is, and where the cost is incurred. The logging techniques used by Stone and by Feldman are very inexpensive but do not accurately replay the program. Minimal logging techniques such as Instant Replay accurately replay the program but only support global replay — the entire program must be replayed to get the proper state of any process. More complex techniques such as PPD move the cost of logging into the compilation and replay phases.

An important aspect of re-execution techniques is that an equivalent virtual machine environment must exist for these techniques to be successful. As with sequential programs, the value of the real–time clock must be simulated if the program made use of it, as well as the results of the resource allocation calls. In parallel programs, re-execution techniques must take care that nondeterministic constructs such as guarded commands execute deterministically during replay. Finally, these techniques are not applicable to real–time programs or those that depend on the physical characteristics of resources allocated to them by the system.

2.3.5 Database Approaches

Several parallel debuggers use trace files to collect information for post run interrogation. A fully detailed trace can be collected during program execution, or more practically, a sufficient amount of information to support a replay technique is collected. Using replay, the program may be rerun as needed to collect additional information. A natural technology for the examination of trace files is available in relational database systems. Database systems are adept at storing the huge amount of data in execution traces. Database query languages are useful for examining the behavior of the program over time.

Garcia-Molina, Germano and Kohler[33] used a database approach in a debugger built for a transaction system. In their system, a coarse trace is automatically produced by a debugging nub at each node in the distributed system. The authors advocate a two-phase debugging paradigm in which a coarse trace is used to locate the general area of the bug. A second run with finer tracing in the area of the bug is then used to locate the bug.

Snodgrass[79] divides the tuples in event trace databases into *event relations* and *interval relations*. A tuple in an event relation represents an instantaneous state change. A tuple in an interval relation specifies a relationship valid over an interval of time, for example, the interval of time between a process blocking to wait for a message and that message arriving. Snodgrass has formalized the semantics associated with event and interval relations and incorporated them directly in a temporal query language. The support of time as a language primitive greatly simplifies the task of examining the state of the system.

LeDoux and Parker's YODA debugger[54] stores a full event trace of Ada programs as a Prolog database. Statements in Prolog become queries for the database. Queries use a temporal logic that supports the notion of before, during, and after. Temporal logic allows states to be described as constraints on events, for example, a task is in a callable state after its activation and before its termination. The authors argue that Prolog's flexibility makes it a more appropriate formalism than the relational database model for the examination of event traces. A similar approach is used by Goldszmidt[34] for Occam programs.

2.3.6 Modeling Approaches

Modeling approaches attempt to deal with the complexity of parallel programs by providing a means to describe portions of a program behavior at a higher level of abstraction. Instead of thinking of program behavior in terms of individual actions

the user thinks of it in terms of groups of actions. For example, a user may think of a sequence of actions within a single process as a single logical action. Alternatively, a user may think of a group of actions performed on different processes as a single logical action (as described in the example in Chapter 1). More complex groups of actions are possible.

A common use of models in debugging systems is as a mechanism for setting breakpoints. In a traditional debugger a breakpoint is described as an isolated state transition within a single process. Using models, a breakpoint can be described as a set of transitions. A model specifying a sequence of events within a process could trigger a breakpoint on the last transition of the sequence. A model specifying a set of events across processes could trigger a breakpoint when the last process completes its action. This approach is advocated by Haban and Weigel[37] and Miller and Choi[61].

The IDD system of Harter, Heimbinger and King[38] uses models of program behavior to insure that the program is maintaining specified invariants. An *assertion* in IDD is a model used to describe a behavior that must always be true. IDD checks the model against the behavior of the program while it is running and halts the program if an assertion is violated.

IDD assertions are statements in an extension of temporal logic called *interval logic*. Interval logic belongs to the class of temporal logics which add to first order predicate calculus the operators *always* and *eventually*. *Always* P asserts that predicate P is true for every state in the sequence. *Eventually* P asserts that P is true for some element of the sequence. Interval logic defines operators over bounded intervals of time: the notation $[I]P$ asserts that P is true over the interval of the sequence defined by I. Since any assertion over an interval that doesn't exist is true, interval logic also allows the specification of $*I$ to mean that the interval I must occur. IDD can present different views of system behavior including process-

time diagrams. The user can interact with process-time diagrams to set the scope of assertion matching to the selected regions of the diagram.

Baiardi *et al.*[6] have an assertion-based editor for a language based on CSP. In their system program behavior is specified as a distributed set of models. These models specify a partial ordering of events from the point of view of each process. Specification is in terms of local events or certain global events visible to the process. A specification serves as a trigger for assertion checking; after a specification is matched the assertions associated with that specification are evaluated.

Each application process has an associated debugger process that intercepts communication events and matches them against the behavioral specifications for this process. Nontransparency is a significant problem in this architecture because of the delays introduced by the debugger processes, but the the language provides a means of partially compensating for them. The language construct affected by nontransparency is the *alternative* statement, which nondeterministically evaluates one of its clauses. The language provides an optional *delay* clause that delays evaluation of an *alternative* statement until the behavioral specifications and assertions associated with that statement have been evaluated, so as to mask the overhead introduced by the debugger processes.

Models in Meldal, Luckham, and Haberler's task sequencing language (TSL)[60] are envisioned as a basis for formal proofs of correctness and as a basis for run-time checking of assertions. The basic primitive event is an action indicating that an Ada statement has been executed. Models use a temporal-logic like notation that has activating events, desired conditions, and terminating events. Models are used to specify both bounded liveness properties (conditions that must eventually occur) and safety properties (conditions that must always occur). TSL also has a hierarchical abstraction mechanism (macros) and allows models to have state variables.

Bates and Wileden[12] have created a modeling language called EDL based on the constrained event formalism. Models are described in a regular expression based syntax extended with the *shuffle* operator used to model parallelism. The shuffle operator is a binary operator that matches its two operands in either order. Expressions can be named and combined to build hierarchical models of system behavior. EDL also allows models to use state variables to control how events are recognized. Candidate events must not only meet the temporal constraints specified by the regular expression, but also value constraints expressed in terms of the state variables. This extra ability to control recognition gives the model more power than simple regular expressions.

Bates[10, 11] has developed the event based behavioral abstraction (EBBA) approach to debugging using EDL. Debugging in EBBA is the process of building models of behavior and detecting differences between the models and the actual behavior of the program. A user builds behavioral models of the part of the program of interest. The executing program generates a stream of primitive events which are passed to a model recognizer. The recognizer finds each group of primitive events that match the constraints specified in the model, binds the group together and inserts the group into the event stream as an *abstract* event. All events are eventually passed to a behavior monitor for further analysis. The behavior monitor provides reports on the recognition of models.

The KRAUT debugger by Bruegge[17] uses a generalized version of *path expressions* for debugging programs. Path expressions in KRAUT can be used either to set breakpoints or to control assertion checking. Several versions of path expressions have been developed. Basic path expressions, based on regular-expressions, were originally developed to specify constraints on the synchronization of shared objects in a parallel version of PASCAL. The following path expression, for example,

```
PATH (Open; (Read | Write)*; Close) End;
```

specifies that the sequence of operations on a file must be an `Open` followed by an arbitrary sequence of `Reads` and `Writes` followed by a `Close`. *Predicate path expressions* add predicates and certain predefined functions. The functions compute how many times an operation has been requested, started, or terminated. *Generalized path expressions* add the ability to model in terms of the activation of a statement or group of statements with a single entry/exit. In a *path rule* (used in KRAUT), a generalized path expression is associated with a debugger action that is performed whenever the expression is matched (or fails).

Garcia and Berman[32] have an animating debugger which uses Petri nets as a visual representation of parallel programs. Processes are written in PASCAL and coordinated with path expressions. The path expressions are used to construct an extended Petri net that represents the synchronization behavior of the program with places representing processes or semaphores, and transitions representing operations on semaphores. During execution the marking of the Petri net is updated to show the current state of the program. Breakpoints can be specified textually using an event definition language or graphically by attaching conditions to the Petri Net.

A further extension of path expressions was developed by Hseush and Kaiser[43] as part of the MD debugger in the Meld project. MD permits data expressions involving arbitrary program variables to be used. For example the data path expression

```
[x>0 & y>0]
```

is true whenever `x>0` and `y>0` are both true during program execution. DPE expressions are also hierarchical. A significant change from path expressions is the representation of program behavior. The behavior is represented as a graph where

the nodes represent events and the edges represent Lamport's *happened before* relation. The operators such as sequence, concurrency, and repetition are defined in terms of paths in the graph. This representation allows more precise models to be built. We will present DPEs in more detail in Chapter 5 in the context of our own modeling language.

2.3.7 Hybrid Approaches

A hybrid approach integrates techniques from other approaches. One such system has been proposed by Leblanc, Mellor-Crummey, and Fowler[53]. It incorporates domain specific animations such as those produced by BALSA[14], logical time communication visualizations such as those produced by Voyeur[80] and Belvedere[41, 42], and performance oriented visualizations such as those produced by Moviola. The system will also include a replay mechanism. The user begins debugging with a high level visualization to obtain an overall understanding of the algorithm, and then uses more focused logical time visualization on selected parts of the behavior. Once the user is assured of correctness, performance debugging proceeds with low-level, timing oriented visualizations.

2.4 Conclusions

Many of the techniques developed for parallel debugging seek to provide the same state examination capability for parallel programs that sequential debuggers provide for sequential programs. However, parallel programs are more complex than sequential programs and debuggers for parallel programs must do more than provide access to state information; they must help the programmer interpret those states. Animation and modeling systems provide a step in this direction by allowing the user to think of the program's behavior at a higher level of abstraction. Animations systems provide a means of creating high level visualizations exactly

tailored to the user's needs, but these animations require significant programming effort. Modeling systems provide a means of describing abstract behavior but do not provide sufficient assistance to help the user understand that behavior. Combining animation and modeling techniques would allow the user to create flexible, abstract animations but currently no tools support this.

CHAPTER 3

BELVEDERE

In this chapter we present *Belvedere*,[1] a debugging tool that produces visualizations of communication behavior for asynchronous, message-passing parallel programs. Belvedere produces both *primitive* animations, in which process actions representing interprocess communication are displayed individually, and *abstract* animations, in which logically related actions are grouped together. We demonstrate the use of these visualizations in helping users understand patterns of communication.

Section 3.1 introduces Belvedere and describes how primitive event visualizations are generated. Section 3.2 shows how primitive event visualizations can be used to understand programs and expose bugs. Section 3.3 introduces abstract animations and Section 3.4 discusses their limitations. Section 3.5 summarizes the chapter and presents our conclusions.

3.1 Belvedere

Belvedere is a post-mortem debugger that produces visualizations from *events* representing the behavior of a parallel program. A *primitive* event represents an action on a process at a single point in time. The parallel program is instrumented so that actions that affect interprocess communication generate primitive events. Typical primitive events are the `send`s and `receive`s of messages and the `create`s

[1] *Belvedere* comes from the Latin *bellus* meaning "beautiful" and *vedere* meaning "view."

and `deletes` of processes and channels. Figure 3.1 lists the type of primitive events used in Belvedere and the actions they represent.

Event Type	Action
`status`	define the version number of the database
`process-create`	create a process
`process-delete`	delete a process
`process-update`	change a field value associated with a process
`process-attribute`	associate a name-value pair with process
`process-attribute-update`	change the value of a process name-value pair
`channel-create`	create a channel between two processes
`channel-delete`	delete a channel
`channel-update`	change a field value associated with a channel
`channel-attribute`	associate a name-value pair with a channel
`channel-attribute-update`	change the value of a channel name-value pair
`port-create`	create a port connecting a process to a channel
`port-delete`	delete a port.
`port-update`	change a field value associated with a port
`port-attribute`	associate a name-value pair with a port
`port-attribute-update`	change the value of a port name-value pair
`message-put`	queue a message to a channel
`message-send`	begin transmission of a message
`message-arrive`	arrival of a message at a process
`message-request`	request of a message on a channel by a process
`message-receive`	final read of a message at a process
`message-attribute`	associate a name-value pair with a message
`message-attribute-update`	change the value of a message name-value pair

Figure 3.1 Events used in Belvedere.

Belvedere operates on these events as illustrated in Figure 3.2. The events generated by the parallel program are stored in a database. Within Belvedere, the user selects interesting events from the database using a query language. Selected events can be animated using one or more of the six built-in animation styles.

Belvedere operates in the context of the Simple Simon Programming Environment[22]. A programmer using Simple Simon specifies his program as an anno-

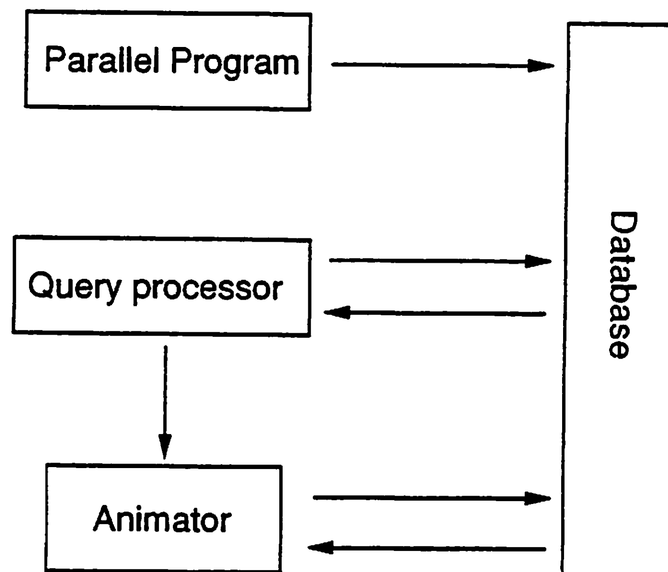


Figure 3.2 Information flow in Belvedere.

tated graph in which nodes represent processes and edges represent communication channels. The graph is stored in a database as a sequence of events denoting the creation and placement of its nodes and edges. Process codes are written as ordinary C programs that communicate using a library of message-passing routines. Programs are compiled and executed under the control of the Simon generic multiprocessor simulator[30] which has been instrumented to produce an event trace. The event interface assigns attributes to each event as it is generated. Each event has attributes that describe its type, the process that executed it, and the local time at which it occurred.[2] Events may have additional attributes that depend on their type. Further documentation of events and their attributes is given by Bailey[8].

Belvedere's query language is used to select events for animation. The query language is a subset of a full relational query language oriented towards simple

[2]Events supplied by the environment are an exception and do not always have a process identifier.

selections. Each event type, such as `message-send` or `process-create`, is treated as a relation containing the events of that type as tuples. The `select` query selects events from an event type by specifying a constraint on an event attribute. A constraint consists of an attribute name, a relational operator ($<, <=, =, >=$, or $>$), and a constant. For example, the following query,

```
select where message-send.time < 61400
```

selects the `message-send` events that occurred before time 61400.

The basic form of the select command does not support arbitrary queries. In particular it provides no means for selecting events based on attributes of events in other event types. For example, suppose the user wishes to select all of the messages sent from a particular named process. The `message-send` event does not have a field that identifies the name of the process that sent the message; the name of a process is only stored in its `process-create` event. This query cannot be expressed using the form of Belvedere's select query given above because there is no way to connect the `message-send` event to the `process-create event`. In a standard language this query could be performed by using a natural join between the `message-send` and `process-create` relations on the process-identifier field and then projecting the result onto the `message-send` relation.

Belvedere's query language does not have a *join* operator but provides a similar function for certain identifier fields using a set of *objects* superimposed on the database. Objects connect events that have the same value of an identifier field. A query containing an object reference provides special interpretation of an identifier field within the event, allowing another event with the same identifier to be treated as a subfield of the current event. For example, the following query uses an object reference to solve the problem posed in the previous example of selecting

all messages sent from a particular named process:

```
select where message-send.process.process-create.name = ''p11''
```

The query selects the events denoting a message sent from process "p11" by selecting only those `message-send` events whose associated `process-create` event contains "p11". It uses the keyword "process" to interpret the process-identifier field of the `message-send` event as an object and reference the `process-create` event with the matching process-identifier. The query language supports four kinds of objects: *processes, ports, channels*, and *messages*. These objects interpret the process-identifier, port-identifier, channel-identifier, and message-identifier fields of events, respectively. Figure 3.3 lists the objects that may be referenced within each event type.

An object reference uniquely identifies a set of events. For example, a process-identifier field uniquely identifies a set of `process-create`, `process-delete`, `process-update`, `process-attribute`, and `process-attribute-update` events. A query using a process identifier field can reference the fields of any event in the set. If the reference field name is unique among these events, a shortened form of the query is supported. For example, in the previous example query, there is only one "name" field among the events pointed to by the process object reference. The following query,

```
select where message-send.process.name = ''p11''
```

also selects the messages sent from process "p11" and does not require the "process-create" event to be specified.

The select command can be combined with other commands to create event selections that can not be specified using the select command alone. All commands in Belvedere operate on *sets* of events, which take their arguments and

Event Name	Can Reference Objects
status	none
process-create	process
process-delete	process
process-update	process
process-attribute	process
process-attribute-update	process
channel-create	channel
channel-delete	channel
channel-update	channel
channel-attribute	channel
channel-attribute-update	channel
port-create	port, process, channel
port-delete	port
port-update	port
port-attribute	port
port-attribute-update	port
message-put	message, process, port, channel
message-send	message, process, port, channel
message-arrive	message, process, port, channel
message-request	message, process, port, channel
message-receive	message, process, port, channel
message-attribute	message
message-attribute-update	message

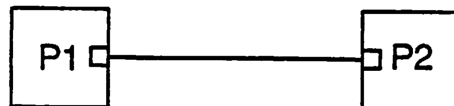
Figure 3.3 Supported object references by event type.

return their results to a stack. Event sets can be positioned on the stack using the `clear`, `dup`, `pop`, and `swap` commands, and combined using the `union`, `difference`, and `intersection` commands. Complex event selections can be made using the stack and commands as a "calculator". For example, to select all messages named "phase-1" that were not sent through "west" ports, the user would first select the "phase-1" messages, duplicate this set on the stack, select the westbound messages from top of the stack, and then subtract the westbound messages from the "phase-1" messages using the `diff` command.

At any time the events on the top of the stack can be animated. The user selects an animation style by means of a command or menu pick. The animation style supplies a set of *drawing* functions and a *closure* procedure. The drawing functions determine how the events will look on the screen. The closure function adds events to the set of selected events to insure that the selected events can be coherently displayed. For example, the standard animation of a message moves an arrow across a channel from one process to another. To produce a coherent display of a message the set of animated events must contain events describing the sending, transmission, and receipt of the message (appropriate `message-put`, `message-send`, `message-arrive`, `message-request`, and `message-receive` events) and events describing the layout and position of the processes, ports and channel involved. If the user selects only a `message-send` event for display, the closure procedure will add the other events required for animation.

The *view* command generates an animation using the selected events and the animation style. First, the selected events are closed using the closure procedure for the animation style. Next, Belvedere binds a drawing function from the animation style to each event based on the type of the event. Belvedere then processes the events in the order they are stored in the database.[3] As each event is processed the associated drawing function is called.

The following example illustrates how a primitive event animation is generated. Consider a parallel program that consists of two connected processes, `P1` and `P2`,



[3]The events in the database must be stored in an order that reflects a possible execution: events from a single process are stored in the order they occurred on that process and the events denoting the sending of a message precede the events denoting the corresponding receipt of the message.

with the following code bodies:[4]

```
Process P1;                           Process P2;
   Port Right Channel C1;                Port Left Channel C1;
   integer myValue = 17;                 integer myValue, marker;
   send(Right, myValue, ''value'');      receive(Left, myValue);
   send(Right, -1, ''marker'');          receive(Left, marker);
end;                                  end;
```

During execution Process P1 sends two messages to Process P2: the first contains its data value and the second contains a marker indicating the end of messages. A possible trace produced by this program is shown in Figure 3.4.

If the user wishes to see only messages that carry actual values, he might use the following query:

```
select where message-send.name = ''value''
```

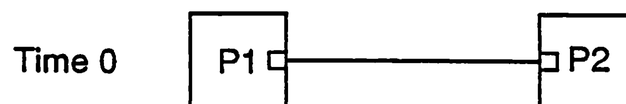
It selects only event 12. The user issues `view` command to begin generating an animation of this event. In the first step of generation, the closure procedure recursively adds the events needed to animate event 12. First, it adds the other events associated with the message transmission: the `message-put` (11), `message-arrive` (14), `message-request` (13), and `message-receive` (15) events. Then it recursively adds events needed to animate the newly added events: the `message-put` event requires the `channel-create` event (5), which in turn requires a `channel-attribute` event (6) to describes its layout and a series of events to describe the connecting processes and ports (1, 2, 3, 4, 6, 7, 8, 9, and 10). The final closed set of events for this selection contains 15 events, shown in Figure 3.5.

[4]These code bodies are Simple Simon programs in which the syntax has been simplified.

Event Number	Time	Process	Event Type	Remark
1	0	P1	process-create	
2	0	P1	process-attribute	layout of process P1
3	0	P2	process-create	
4	0	P2	process-attribute	layout of process P2
5	0		channel-create	
6	0		channel-attribute	layout of channel C1
7	0	P1	port-create	
8	0	P1	port-attribute	layout of port Right
9	0	P2	port-create	
10	0	P2	port-attribute	layout of port Left
11	1	P1	message-put	value message
12	1	P1	message-send	
13	1	P2	message-request	
14	2	P2	message-arrive	
15	3	P2	message-receive	
16	2	P1	message-put	marker message
17	2	P1	message-send	
18	3	P2	message-arrive	
19	4	P2	message-request	
20	4	P2	message-receive	

Figure 3.4 Events from an execution. The events at Time 0 are produced by the environment and represent the creation and placement of processes, ports, and channels. The `channel-create` and `channel-attribute` events do not have process attributes because the events are not associated with any process.

The animation procedure calls the drawing functions associated with each event; computing an image for each timestep. In this example, images will be computed for timesteps 0, 1, 2, and 3. At Time 0, events 1 through 10 are processed:

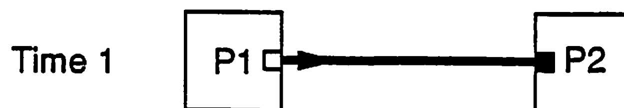


The standard animation in Belvedere draws a process as a large square, a port as a small square, and a communication channel as a connecting line.

Event Number	Time	Process	Event Type	Remark
1	0	P1	`process-create`	
2	0	P1	`process-attribute`	layout of process P1
3	0	P2	`process-create`	
4	0	P2	`process-attribute`	layout of process P2
5	0		`channel-create`	
6	0		`channel-attribute`	layout of channel C1
7	0	P1	`port-create`	
8	0	P1	`port-attribute`	layout of port Right
9	0	P2	`port-create`	
10	0	P2	`port-attribute`	layout of port Left
11	1	P1	`message-put`	value message
12	1	P1	`message-send`	
13	1	P2	`message-request`	
14	2	P2	`message-arrive`	
15	3	P2	`message-receive`	

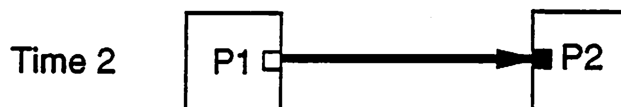
Figure 3.5 Closed event set after selection of `message-send` event.

Events 11, 12, and 13 occur at Time 1: Process `P1` queues a message for sending, the message begins transmission on the channel, and Process `P2` requests the message:

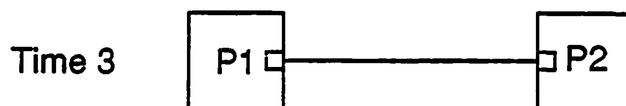


The sending of the message is indicated by an arrow head drawn on the channel near Process `P1` and the highlighting of the channel. The request of the message is indicated by the highlighting of the port on Process `P2`.

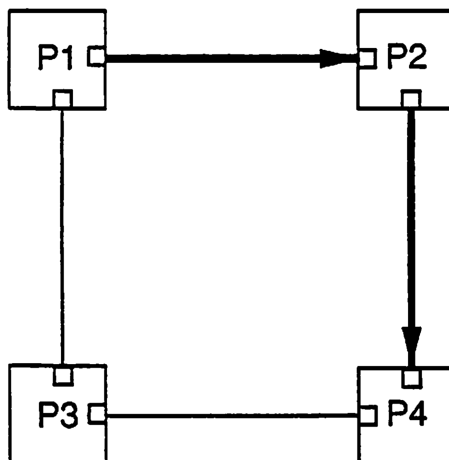
At Time 2 the message arrives at Process P2, as shown by the position of the arrow head:



At Time 3, the message is read. The arrow head is removed, the highlighting on the port and channel is removed, and the animation completes:



An animation style in Belvedere can be easily changed by changing the drawing functions associated with each event type. For example, in the *traced* animation style, highlighting persists: the drawing function associated with the `message-receive` event type undraws the highlighting on the port but leaves the arrowhead and highlighting on the channel. A traced animation of a message traveling from Process `P1` to Process `P4` via Process `P2`, for example, results in the following snapshot,



taken after Process `P4` has read the message.

In the next section we show the use of primitive event animations in understanding and debugging parallel programs.

3.2 Using Primitive Event Animations

Primitive event animations are most useful when the displayed behavior has distinctive patterns of communication. This may occur in programs with tightly coupled behavior that can not be distorted by asynchrony. For example, the processes in the bitonic sort program running on a hypercube[71] are tightly synchronized and have nearly identical behavior. Snapshots from successive phases of the sort are shown in Figure 3.6. Although Figure 3.6(b) shows message exchanges within the second sorting step in various stages of completion, the images are still easily understandable.

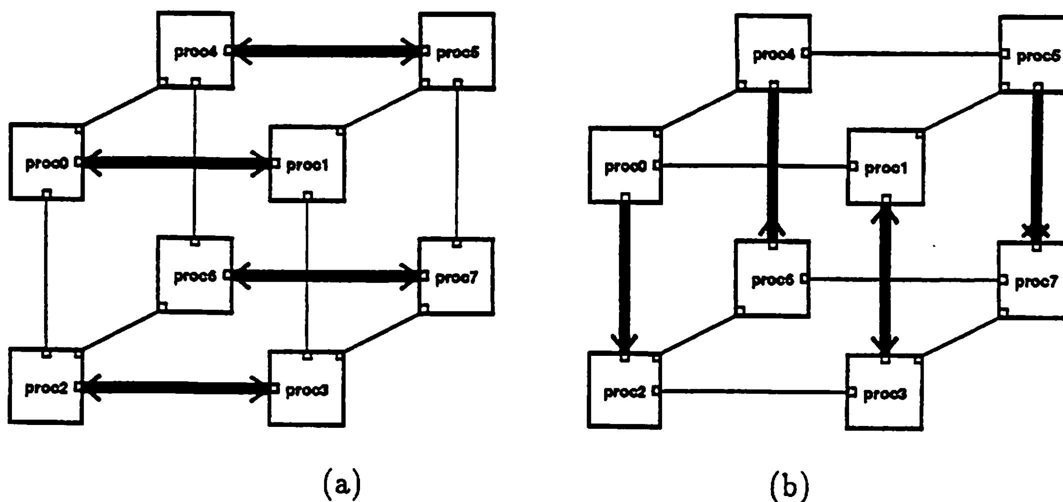


Figure 3.6 Successive communication phases in the bitonic sort.

Primitive event visualizations are also useful in illustrating the deadlocks frequently encountered in the early phases of debugging. Figure 3.7 shows a deadlocked implementation of a program that computes the solution to a partial differential equation using an iterative relaxation algorithm. The snapshot shows the four processes in the upper left corner in the circular wait characteristic of deadlock.

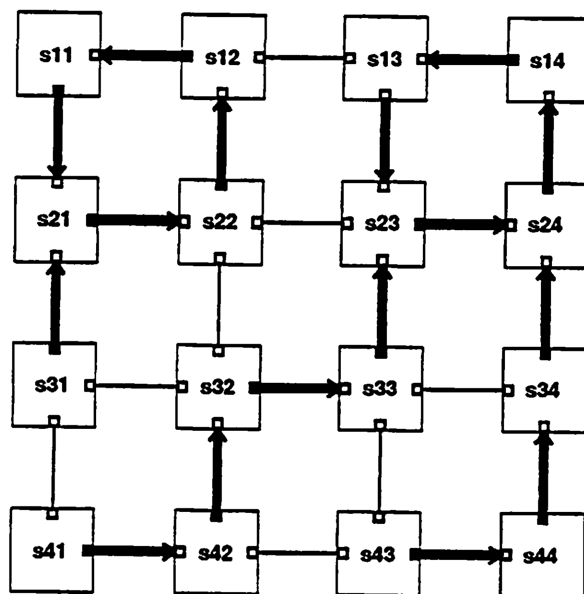


Figure 3.7 Deadlocked version of an iterative relaxation program. Unfilled requests for messages are shown by highlighted ports.

Primitive event visualizations of large or complex behaviors can be hard to understand. With the query facilities of Belvedere, the user can view selected parts of a behavior. For example, a snapshot of a median filter program in which each process sends its value to its immediate neighbors and then reads back its neighbors' values is shown in Figure 3.8. It is difficult to interpret because of the large number of messages. We could look at just the behavior of a single process, using the query

```
select where message-send.process.name = ''e22''
```

which selects only those messages sent by process "e22". The resulting traced animation, shown in Figure 3.9, shows all of the communication from a single process and clearly indicates a bug: processes do not communicate with all of their neighbors.

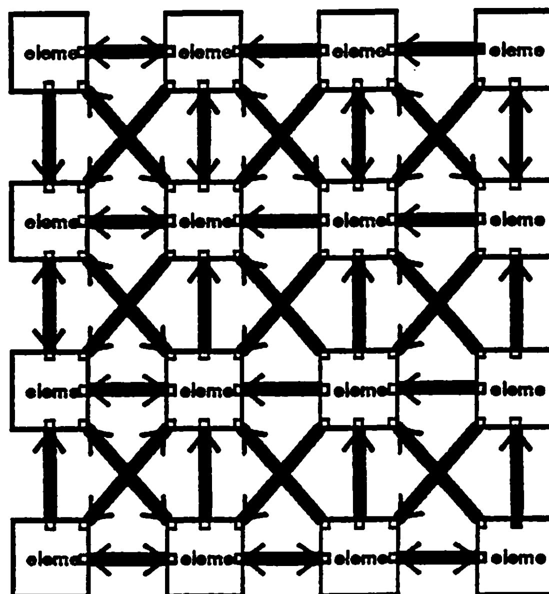


Figure 3.8 Complex communication pattern in the median filter program.

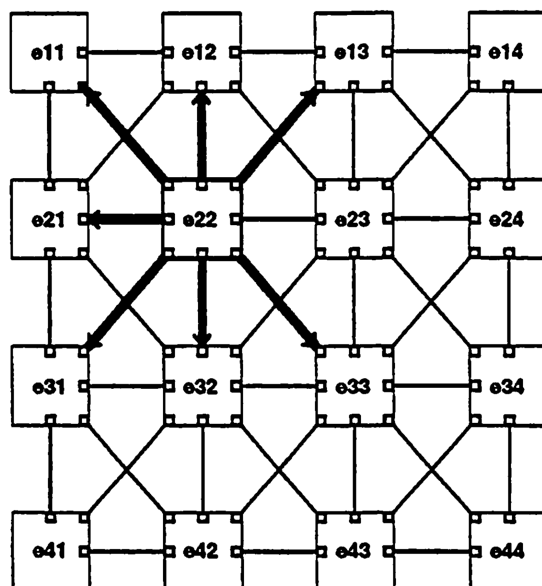


Figure 3.9 Median Filter: Traced animation of messages sent from Process e22.

3.3 Abstract Animations

Communication in parallel programs is often best thought of in terms of groups of logically related messages rather than in terms of individual messages. Primitive event visualizations do not always display behavior so that these groups can be seen. For example, the wavefront matrix multiply of S. Y. Kung[47] has a complex structure (shown in Figure 3.10) in which the input matrices are padded with null

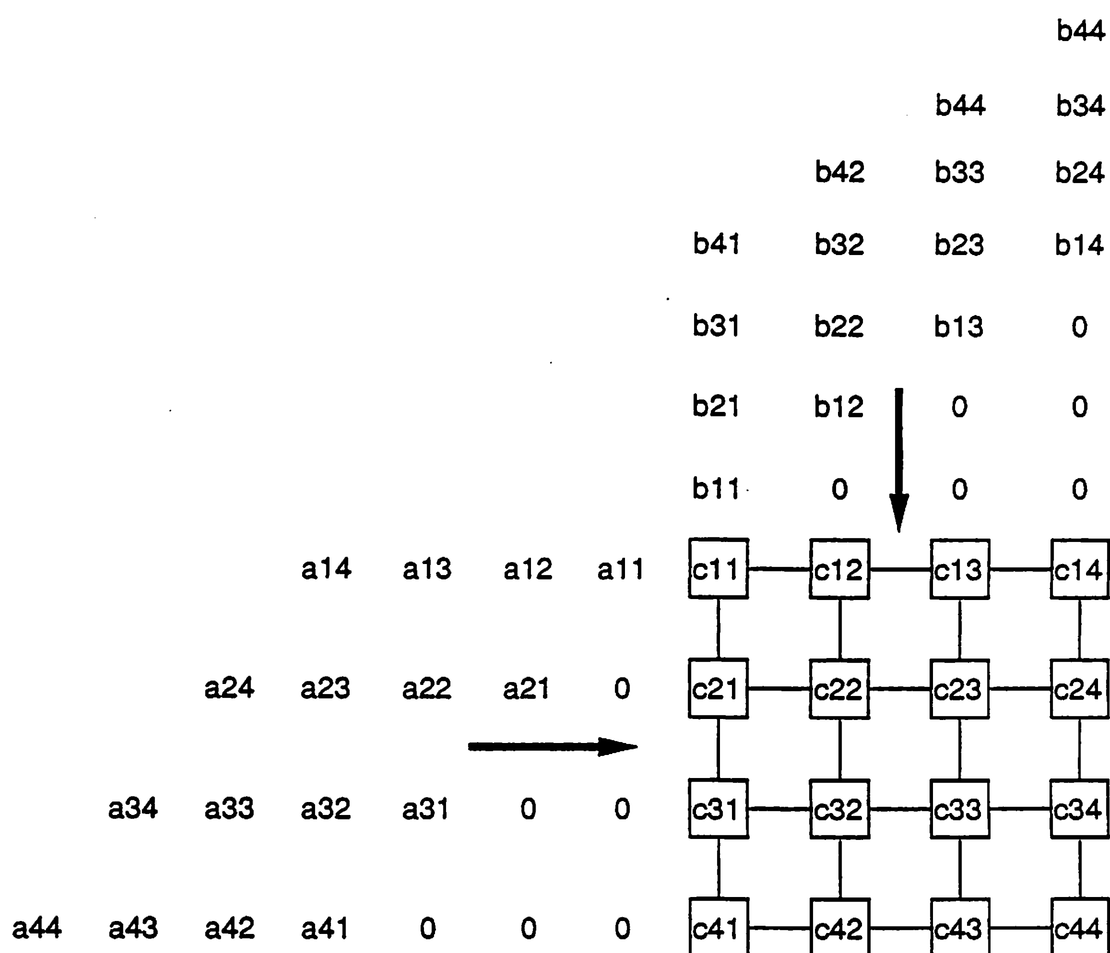


Figure 3.10 Wavefront Matrix Multiply.

values so that the proper elements of the matrices will be multiplied together at the

proper time. The pairing of values is critical to the correctness of the program but this abstract behavior is not visible in a primitive event visualization. A snapshot from a primitive event visualization, shown in Figure 3.11, shows messages moving

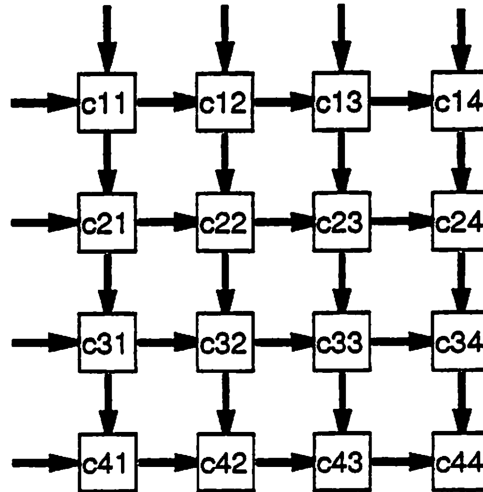


Figure 3.11 Primitive event animation of the Matrix Multiply.

through the process array but does not help us see that the correct pairs of values are being multiplied together.

To help us see the abstract behavior of the program we use an *abstract animation*, in which logically related events are displayed as distinct visual units. The user describes the events that are related to each other and then the system attempts to display these events so that the groups of events are identifiable. For the example in this section we use color to identify groups; in the next section we show that coloring is not sufficiently powerful to produce comprehensible displays of all behaviors.

To create abstract animations Belvedere operates in conjunction with an abstract event *recognizer*, shown in Figure 3.12. The user provides a named model of the abstract behavior. We currently use a version of Bates and Wileden's Event

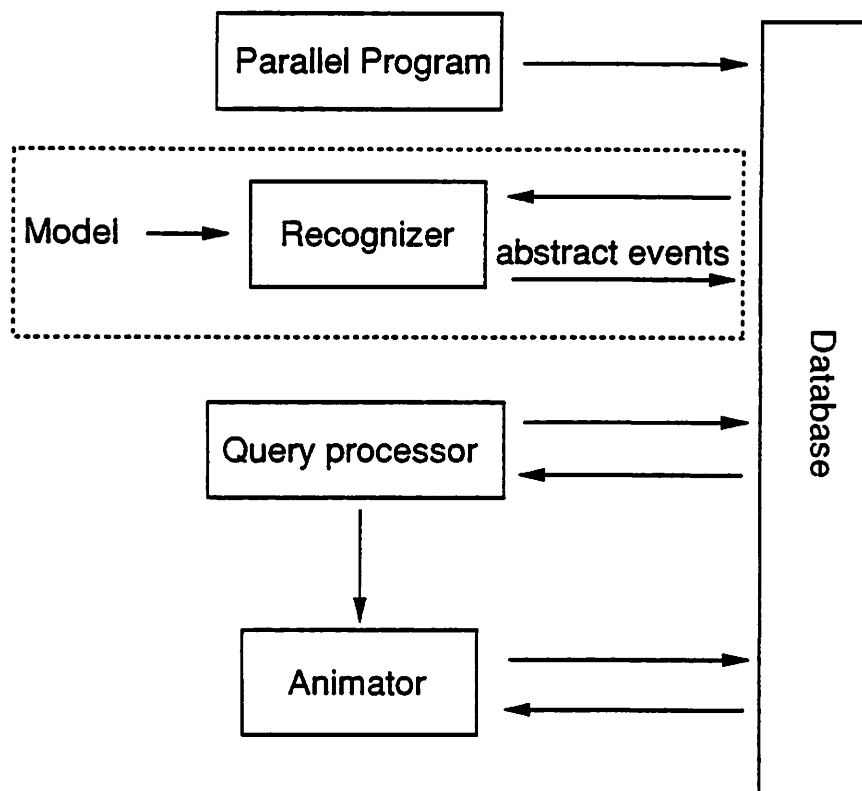


Figure 3.12 Belvedere with an abstract event recognizer. The recognizer reads primitive events from the database, groups them according to a model, and returns both the primitive events and their groupings into abstract events to the database.

Definition Language (EDL) as our modeling language. The recognizer finds a set of events that meet the constraints specified in the model and then binds those events into an abstract event. The abstract events are then added to the database. Abstract events may be selected using the name of the model as an event type; they are displayed using the same facilities that are used for primitive events. The standard animation style displays an abstract event by displaying the primitive events that belong to it in the same color.

We now illustrate the construction and use of an abstract animation using the wavefront matrix multiply example. The user wishes to see if the proper pairs of elements from the input matrices are being multiplied together. We model the

messages carrying the values of each intended inner product as an abstract event. It is easy to observe if each process operates on the proper elements of the inner product by observing whether or not the incoming messages to a given process at a given time are the same color.

The EDL model of this behavior uses two models in a hierarchical fashion. The first model, shown in Figure 3.13, groups together the five events that represent a message transmission between two processes. The model describes an abstract

Message is message-request $\triangle$ *(message-put* $\circ$ *message-send* $\circ$
message-arrive $\circ$ *message-receive)*
with
int sender;
int receiver;
filter
message-put.messageId = message-request.messageId
message-send.messageId = message-request.messageId
message-arrive.messageId = message-request.messageId
message-receive.messageId = message-request.messageId
aftermatching
sender = message-send.processId;
receiver = message-receiver.processId;
end

Figure 3.13 EDL model of a message transmission.

event *Message* that consists of a `message-request` event that occurs with a sequence of events consisting of a `message-put`, a `message-send`, a `message-arrive`,

and a `message-read` event.[5] The *filter* clause insures that the events describe the same message by constraining the *messageId* attribute of each event assigned by the communications library. The *with* clause declares two new attributes of the *Message* abstract event called `sender` and `receiver`. The *aftermatching* clause sets the values of the *sender* and *receiver* attributes to the identifier of the sending and receiving process, respectively.

A second model groups together the *Messages* that should contribute to an inner product step of the matrix multiplication. An inner product step multiplies $a_{i,k}$ by $b_{k,j}$ and accumulates the result in $c_{i,j}$. To model each step of the inner product the model, shown in Figure 3.14, groups together messages carrying the values of $a_{i,k}$ and $b_{k,j}$ when the $a_{i,k}$ message reaches process $c_{i,j}$. The model assumes that the programmer has supplied data in each message that identifies the element of the input matrix that the message is carrying (to identify messages carrying $a_{i,k}$ and $b_{k,j}$) and that the environment supplies a row and column attribute for each process (to identify process $c_{i,j}$). A snapshot from an abstract animation of the *InnerProductStep* events is shown in Figure 3.15. The abstract animation shows the messages belonging to each *InnerProductStep* in the same color, and does not show the messages carrying zero padding values because they do not belong to any *InnerProductStep*. In the snapshot we see that processes are not combining messages of the same color. On closer examination we can see that the B matrix is shifted "up" one row (relative to the A matrix), causing the wrong values to be multiplied together. The abstract animation, which displays the behavior of this program in terms of groups of related events, allows the user to see an erroneous communication pattern that is difficult to see in the primitive event animation.

[5]EDL is presented in more depth in Chapter 5. The reference for EDL is Bates' dissertation[10].

$InnerProductStep$ *is* $Message_1 \triangle Message_2$

filter

$Message_1$*.message-send.value.matrix* = "*a*"

$Message_2$*.message-send.value.matrix* = "*b*"

$Message_2$*.message-send.value.element.row* =

$Message_1$*.message-send.value.element.column*

$Message_2$*.message-send.value.element.column* =

ProcessAttribute[$Message_1$*.message-receiver*]*.column*

end

Figure 3.14 EDL model of an inner product step in the Matrix Multiply program.

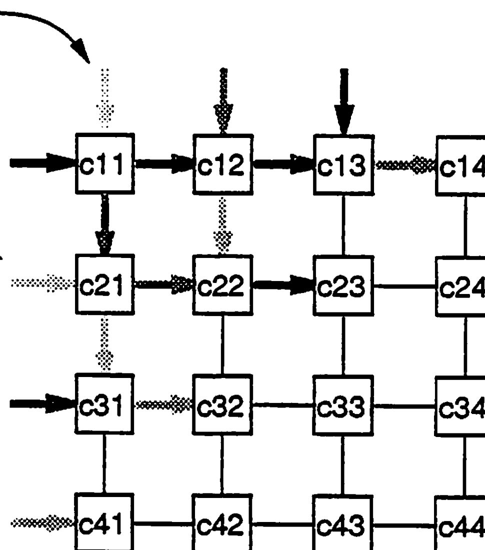
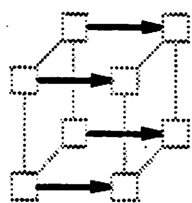


Figure 3.15 Unexpected communication patterns in the Matrix Multiply.

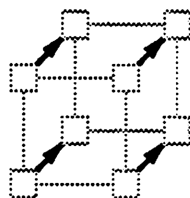
3.4 Problems with Abstract Animations

Abstract animations do not provide a complete solution to the problem of displaying logical patterns of communication. The first problem is that abstract animations do not necessarily display an abstract event as a distinct visual unit when the pattern of primitive events has been distorted by asynchrony. The second problem is that the models of communication patterns can be difficult to specify in the modeling language.

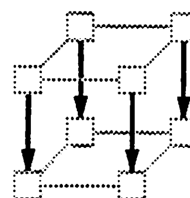
To illustrate the problem of asynchrony let us reconsider the example of an asynchronous application running on a hypercube whose behavior consists of a sequence of phases. We might expect a visualization to be a sequence of images that shows each phase at a distinct time:



Time = 10

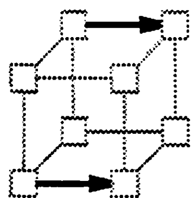


Time = 20

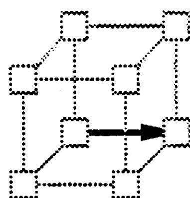


Time = 30

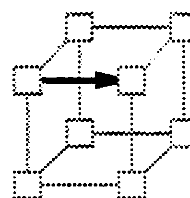
What we would actually see is a visualization where the low-level communication events have skewed. For example, the visualization of the first phase might appear as the following sequence of images:



Time = 10

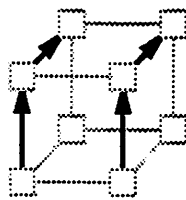


Time = 20

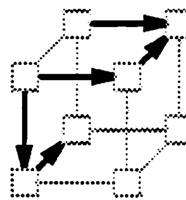


Time = 30

Each process executes independently and asynchronously, proceeding through the computation at a different rate. The skewing is not primarily a result of clock drift between processes; it occurs because processes do individual or data-dependent computations that take different amounts of time. For example, although each process may be running the same program each has a different version of the data; some processes may execute the true part of a conditional branch while others execute the false part. The skew between processes is cumulative and, in the absence of synchronization, will eventually cause behavior from different phases to overlap. Snapshots from later in the execution of the hypercube application show simultaneous communication from the second and third phases (left), and from all three phases (right):



Time = 200



Time = 300

Unique colors for each abstract event would not significantly clarify this animation. Logically related events are displayed at widely separate times and logically unrelated events are displayed at the same time. We address this problem in the following chapter with *perspective views*, a preprocessing step that reorders events prior to display so that abstract events can be displayed as coherent visual units.

The second problem with abstract animations is that they rely on the ability of the modeling language to group together logically related events, but existing modeling languages are not well suited to describing repetitious, asynchronous communication patterns. It is difficult to describe some communication patterns and these descriptions do not work well when the program contains errors. For

example, in EDL we might describe the first phase communication behavior of the hypercube example program with the model shown in Figure 3.16. The model instructs the recognizer to group together four messages that meet the specified criteria — messages that go from the left face to the right face and each message is sent from a different process. The model is complicated, difficult to extend, and prone to mis-specification. Additionally, the model may cause unrelated events to be grouped together if the program contains errors. If a process does not produce a message in an iteration then the recognizer will use a message that meets the constraints from the following iteration. The resulting group of events is probably not what the user intended.

It is very difficult to write models that specify the desired behavior, only the desired behavior, are robust in the presence of program errors and have reasonably compact descriptions. In Chapter 5 we present a modeling language that is better for modeling the repetitive, asynchronous communication patterns common to many parallel programs.

3.5 Summary and Conclusions

Communication patterns are fundamental to the understanding of parallel algorithms. By viewing patterns a user can determine the extent to which expected patterns of communication actually occur. Deviations from expected patterns indicate the presence of bugs.

Belvedere has been designed and implemented to facilitate this process. It has been implemented by the author in C using the X windowing system under Unix and includes a implementation of the EDL language. A basic animation facility animates communication patterns from trace files stored when the program was executed. Animation styles determine how the patterns will be displayed. A

$Phase_1$ *is* $Message_1 \triangle Message_2 \triangle Message_3 \triangle Message_4$

filter

-- *choose only messages belonging to the proper phase.*

-- *make sure that message goes between proper faces by checking*

-- *bits in cube address of process*

$Bit(Message_1.sender, 1) = 0$

$Bit(Message_2.sender, 1) = 0$

$Bit(Message_3.sender, 1) = 0$

$Bit(Message_4.sender, 1) = 0$

$Bit(Message_1.receiver, 1) = 1$

$Bit(Message_2.receiver, 1) = 1$

$Bit(Message_3.receiver, 1) = 1$

$Bit(Message_4.receiver, 1) = 1$

-- *make sure that only one message is sent per process*

$Message_2.sender! = Message_1.sender$

$Message_3.sender! = Message_1.sender$

$Message_3.sender! = Message_2.sender$

$Message_3.sender! = Message_1.sender$

$Message_4.sender! = Message_1.sender$

$Message_4.sender! = Message_2.sender$

$Message_4.sender! = Message_3.sender$

end

Figure 3.16 EDL model of a phase communication.

database front end allows the selective examination of subsets of the program's behavior.

The primitive event animations produced by Belvedere do not always allow the user to see the logical communication patterns in the behavior of a program. We address this problem with abstract animations, in which logically related actions are identified and displayed as distinct visual units. However, problems exist that hamper the utility and generation of abstraction animations. Abstract animations do not display abstract events as distinct visual units when asynchrony has distorted the communication patterns, and existing modeling languages used to describe communication patterns do not work well for repetitious, asynchronous patterns. We address these problems in the following chapters.

CHAPTER 4

ABSTRACT VISUALIZATIONS

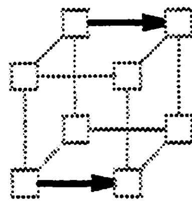
Visualizations help users understand parallel program behavior but primitive event visualizations are often distorted by asynchrony, making them difficult to understand. Abstract animations, in which we attempt to display logically related groups of events together, aid understanding but it is difficult to construct comprehensible abstract animations for general behaviors. In this chapter we introduce *perspective views*, a general technique for generating abstract animations so that logically related events can be displayed as distinct visual units.

4.1 Introduction

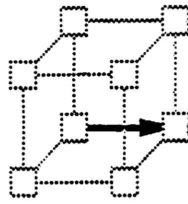
Abstract animations provide a means of simplifying visualizations but they are often difficult to display coherently. The technique used in Chapter 3 — coloring primitive events associated with an abstract event with the same color — works in the simple examples shown but not in the more complicated examples. One problem is that an abstract event may take place over a wide interval of time, making it hard for the user to see that the events that belong to it are related.

To solve this problem, we display a single image for the abstract event over the interval of time defined by its members. As an example, for the hypercube

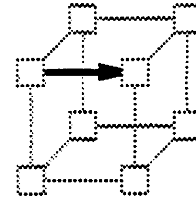
application, instead of showing the first phase of communication as a sequence of images,



Time = 10

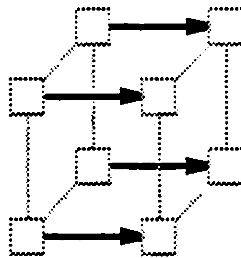


Time = 20



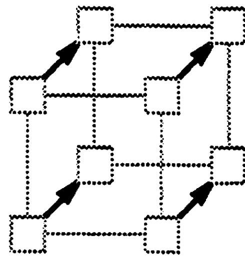
Time = 30

we display it as a single abstract communication that occurs over the interval from time 10 to time 30:

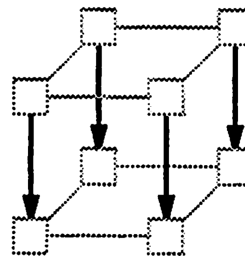


Time 10 - 30

The second and third communication phases of the algorithm can similarly be displayed as units over the appropriate intervals:

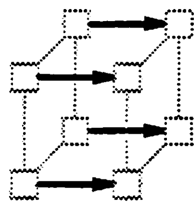


Time 20 - 40

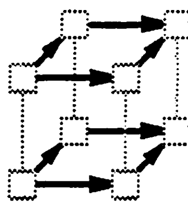


Time 40 - 100

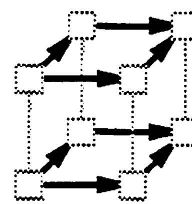
To animate the whole program we could simply display each abstract event over its own interval of time:



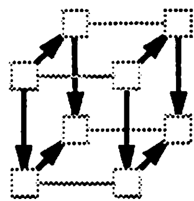
Time=10



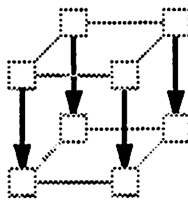
Time=20



Time = 30



Time=40

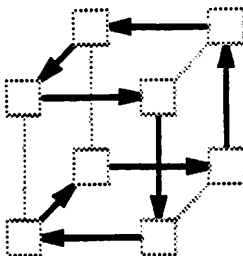


Time=50

At time 10 we start displaying the phase one communication. At time 20, we display the beginning of the phase two communication and the continuation of phase one communication. The display is the same at time 30. At time 40, we delete the phase one communication that has completed and we add phase three communication. At time 50 only phase three communications are shown.

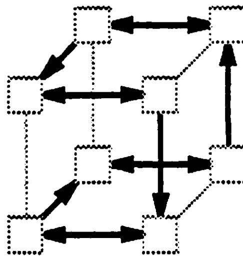
The time intervals associated with each abstract event overlap so the animations of successive phases occur on the screen simultaneously. This animation is comprehensible because the abstract events overlap in *time* but not in *space*. Each communication phase uses a different dimension of the hypercube (and thus an independent set of channels) to communicate. Even though the animation of different phases occurs simultaneously, the animations occur on different parts of the cube and thus different parts of the screen. This is not always the case. If the

hypercube algorithm started with a communication phase with the cube configured as a ring,



Time 0 - 20

an animation of the algorithm in terms of abstract events would show the ring and the first phase communication simultaneously and using the same channels:

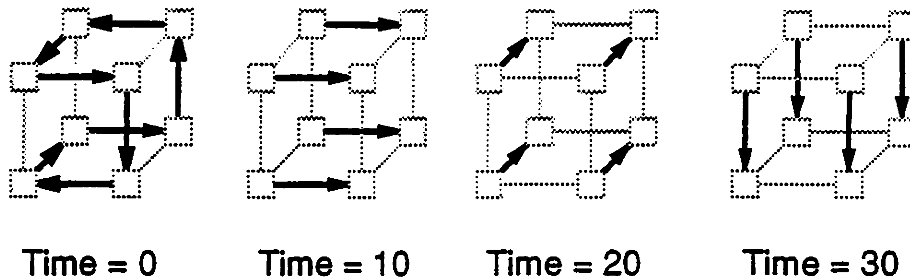


Time 10

From this image it is difficult to understand whether the program is behaving as expected.

In general, abstract events may overlap in time or space or both, making their visualization difficult. Our approach is to provide comprehensible displays of abstract events by reordering events prior to display so that abstract events that overlap in space do not overlap in time. We attempt first to construct *consistent reorderings* that preserve the partial ordering of events imposed by the sequentiality of processes and by interprocess dependencies. In message passing systems the interprocess dependencies mean that the "receive" of a message is dependent on its "send." In the hypercube example, a consistent reordering can be achieved

by putting the ring communication events before the first phase communication events, the first phase events before second phase events, and so on:



This reordered, high-level visualization is easy to understand. Further, the visualization shows an execution sequence that is equivalent to the one that actually occurred; that is, all processors execute the same sequences of operations with the same interprocess dependencies.

For many parallel computations, however, such simple reorderings are not possible because of intertwined dependencies. For these cases, we introduce the concept of *perspective views*. Perspective views enable the user to create comprehensible visualizations by selectively ignoring some events and dependencies to establish partially consistent reorderings. Each such reordering provides a partial view of the system's behavior; several different partial views may be needed to understand its full behavior.

In Section 4.2, we define our notions of ordering between abstract events. In Section 4.3, we discuss reorderings and introduce the concept of perspective views. In Section 4.4, we present an algorithm for computing both full and partial perspective views. In Section 4.5, we demonstrate the use of perspective views with a variety of parallel programs and visualization tools; in Section 4.6, we present our conclusions.

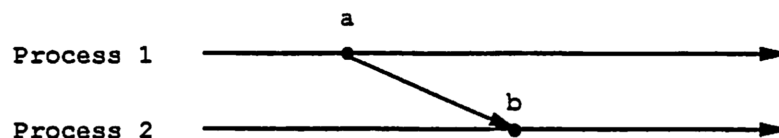
4.2 Orderings between Abstract Events

We present here three ordering relations — *precedes*, *parallel*, and *overlapped* — that characterize the possible relationships between two abstract events.

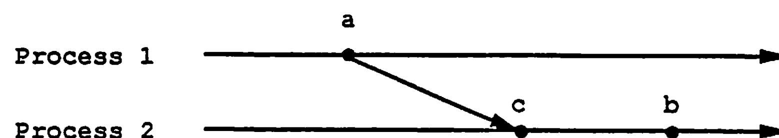
Other ordering relations on abstract events have been proposed[49, 12, 43, 27]; ours is an extension of Lamport's ordering of primitive events[48] that has been tailored to the needs of visualization systems. Lamport's ordering, called *happened before* and denoted $\rightarrow$, is defined on atomic actions of a distributed system; each action a is assumed to have a process-local timestamp, *timestamp*(a). Positioning events on their process time-line according to their local timestamps, a $\rightarrow$ b, iff one of the following conditions holds:

i) events a and b happen on the same process and timestamp(a) $<$ timestamp(b):

ii) a is the act of sending a message and b is the act of receiving it (denoted by an arrow from a to b):

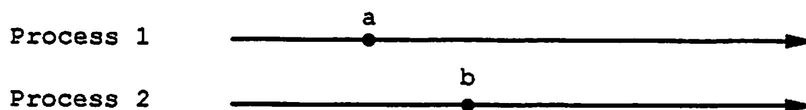


iii) a $\rightarrow$ b is in the transitive closure of i) and ii):

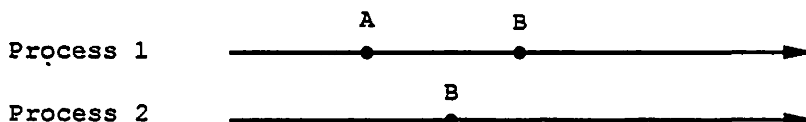


Two events are unordered if they are not related by *happened before* [1]:

$$a \parallel b \text{ iff not } (a \rightarrow b) \text{ and not } (b \rightarrow a)$$



Happened before is defined on primitive events; for our visualization purposes, we must consider relations between abstract, nonatomic events. Here, we define an abstract event to be a set of primitive events (later we extend this to allow hierarchical definitions). For example, in the following behavior,



there are three primitive events and two abstract events. Abstract event A contains one primitive event on Process 1, and abstract event B contains two primitive events, one on Process 1 and one on Process 2. In all of our examples we will use a capital letter over a primitive event to indicate the name of an abstract event that contains the primitive event.

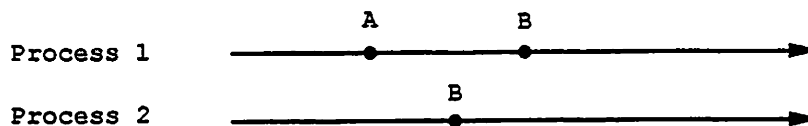
A straightforward extension of *happened before* might require that for abstract events A and B, $A \rightarrow B$ only when each element of A *happened before* each element of B:

$$A \rightarrow B \text{ iff } \forall\, a \in A, \forall\, b \in B: \; a \rightarrow b$$

This extension, however, is overly restrictive. It does not capture all of the behavior that we might reasonably wish to consider "serializable" for the purposes of

[1]This notation is slightly different than Lamport's.

visualization. Consider again the previous example:



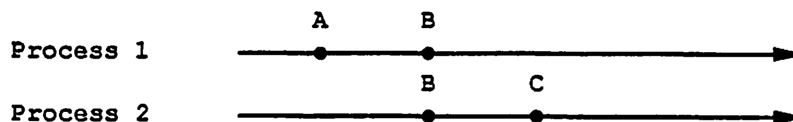
Under this simple extension, event A would not have *happened before* event B because there is no relation between the event belonging to A on Process 1 and the event belonging to B on Process 2. However, for visualization purposes, we might consider A to have happened before B because we would not violate any dependencies by showing the visualization of A before the visualization of B.

To define the relations that we need, we first define a relation *partially precedes*, denoted $\mapsto$. Informally, $A \mapsto B$ if some part of A *happens before* some part of B or A and B share a primitive event; formally

i) $A \mapsto B$ if $\exists a \in A, \exists b \in B : a \rightarrow b$ or $a = b$

ii) $\mapsto$ is closed under transitivity

Partially precedes allows us to order abstract events for visualization purposes that would not have been ordered by the simpler extension to *happened before*. In the following example,



partially precedes provides a total ordering for A, B, and C (because *partially precedes* is transitive, $A \mapsto C$ even though no element of A *happened before* any element of C).

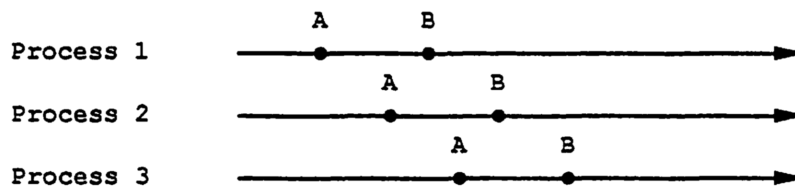
Using *partially precedes* we define three possible relations between two abstract events:

Precedes: $\mathtt{A} \Rightarrow \mathtt{B}$ iff $\mathtt{A} \mapsto \mathtt{B}$ and NOT $(\mathtt{B} \mapsto \mathtt{A})$

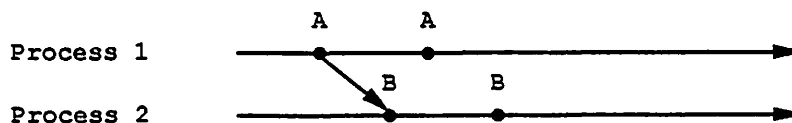
Parallel: $\mathtt{A} \parallel \mathtt{B}$ iff NOT $(\mathtt{A} \mapsto \mathtt{B})$ and NOT $(\mathtt{B} \mapsto \mathtt{A})$,
(alternatively, iff $\forall \mathtt{a} \in \mathtt{A}, \forall \mathtt{b} \in \mathtt{B} : \mathtt{a} \parallel \mathtt{b}$)

Overlaps: $\mathtt{A} \Leftrightarrow \mathtt{B}$ iff $\mathtt{A} \mapsto \mathtt{B}$ and $\mathtt{B} \mapsto \mathtt{A}$

Precedes is intended to capture the notion that one abstract event logically occurs before another. $\mathtt{A} \Rightarrow \mathtt{B}$ if some element of A occurs before some element of B (in the extended sense of *partially precedes*) but no element of B occurs before any element of A. In the following example,

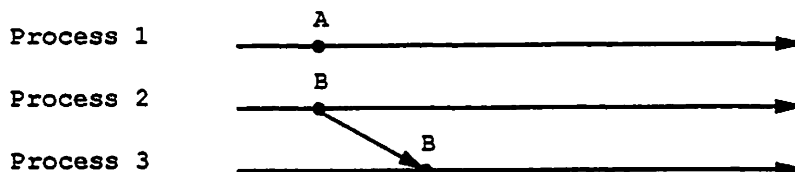


$\mathtt{A} \Rightarrow \mathtt{B}$ because each element of A *happened before* the element of B on the same process. Note that $\mathtt{A} \Rightarrow \mathtt{B}$ even though the element of B on Process 1 has an earlier local timestamp than the element of A on Process 3, because there is no *happened before* ordering from any element of B to any element of A. In the following example,



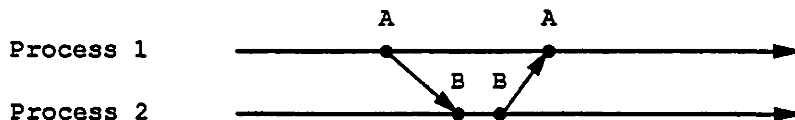
$\mathtt{A} \Rightarrow \mathtt{B}$ because communication imposes the *happened before* relation between the send event of A and the receive event of B.

Parallel events are *unordered.* In the following example,



A || B because the *happened before* relation does not hold between any pair of A and B elements. Note that *parallel* events must occur on disjoint processes because events on the same process would necessarily be ordered.

Overlap is intended to describe events which are logically intertwined: some part of each event happens before some part of the other. In the example,



A ⇔ B because the first element of A on Process 1 *happens before* the first element of B on Process 2 but the second element of B on Process 2 *happens before* the second element of A on Process 1. Overlapped events can also occur on a single process:

In the next section, we consider the use of these relations in reordering abstract events to provide more comprehensible visual displays.

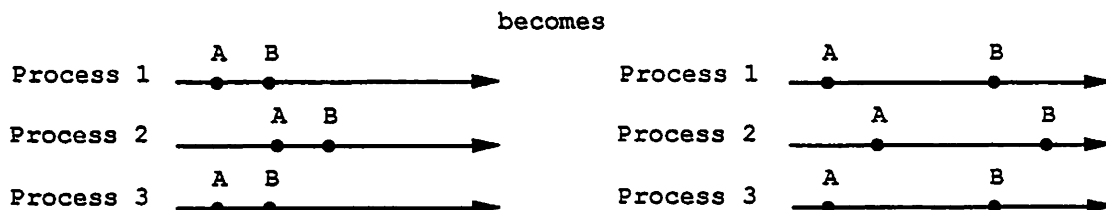
4.3 Reordering Transformations and Perspective Views

Given a set of events and their grouping into abstract events we can compute the *precedes, parallel,* or *overlap* relationship between those abstract events. Associated

with each relation is a transformation that changes event timestamps, reordering the events into a more comprehensible arrangement. For abstract events related by *precedes* or *parallel*, we provide transformations that produce *consistent reorderings.* A reordering is *consistent* if (1) all processes execute the same sequence of events as before and (2) the *happened before* relation of the reordered behavior is unchanged. To an observer inside the system (such as a process), consistent behaviors are indistinguishable. For events related by *overlap*, consistent reorderings are not possible. For these events, we introduce the notion of a *partially consistent reordering* in which (1) each process executes a subsequence of its original event sequence and (2) the *happened before* relation is a subset of the original *happened before* relation.

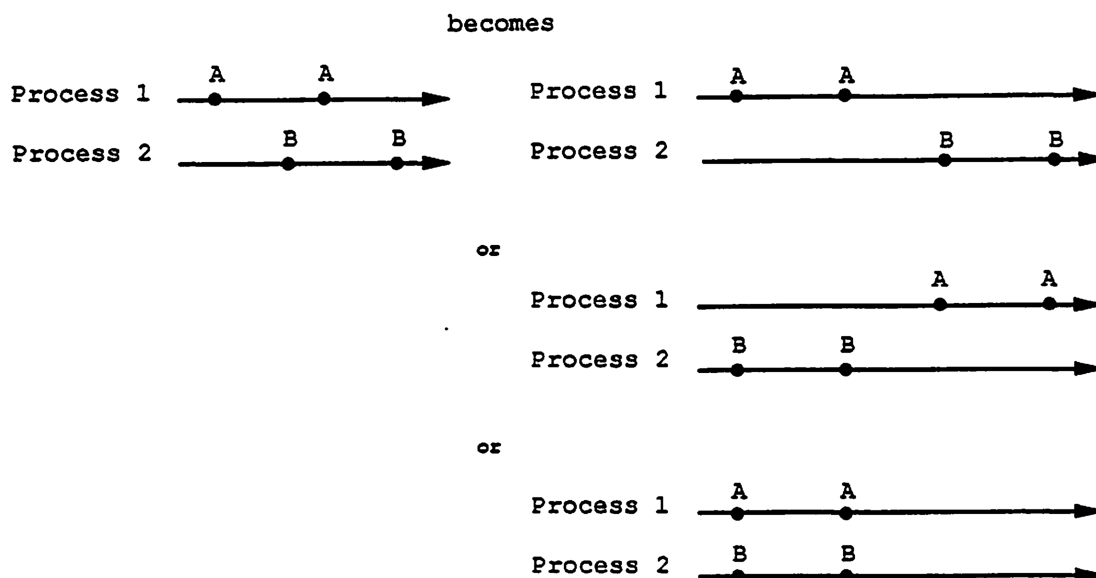
We first describe the consistent reorderings needed for *precedes* and *parallel* and then we describe the partially consistent reorderings needed for *overlap*.

The transformation associated with *precedes* separates two abstract events (by assigning new timestamps) so that all elements of the first event complete before any element of the second event begins. A visualization of the transformed system would show the abstract events as distinct visual units. Thus,



The transformed behavior is consistent with the original behavior. In addition, it is easier to understand because logically related events are grouped together.

For the transformation associated with *parallel*, there are a variety of options:

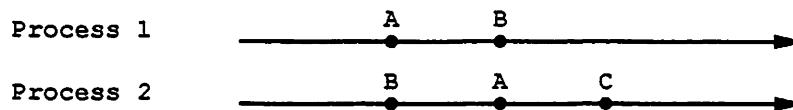


We chose the third — in which *parallel* events are started at the same time — for aesthetic reasons. Parallel events must occur on separate processes so this does not conflict with our goal of making the events visually distinct.

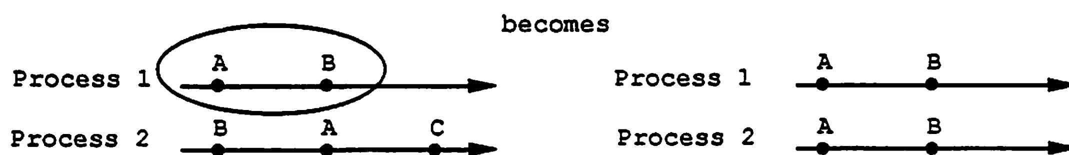
Both transformations were used in producing the comprehensible display of the hypercube computation in Figure 1.3. The transformation for *precedes* separated the abstract events and the transformation for *parallel* synchronized their subevents.

If two events are related by *overlap*, it is not possible to assign new timestamps that will both separate the events and preserve the *happened before* relation. For these events, we use partial reorderings based on a user-selected subset of the events called a *perspective*. If all events are selected, the subset is called a *full perspective*; otherwise it is called a *partial perspective*. Only events named in the perspective are used in computing ordering relations. Events not named in the perspective and not needed for the display of named events are deleted and the remaining events are reordered in a manner consistent with the computed relations. We might, for

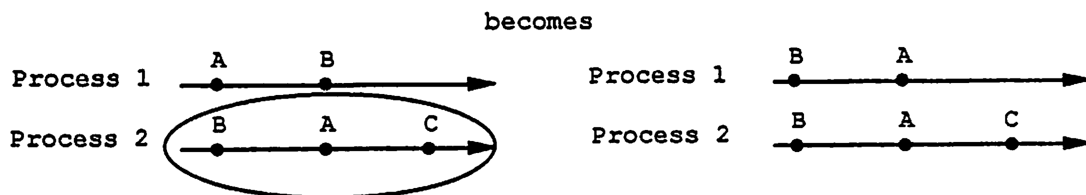
example, consider the following behavior



from the partial perspective of Process 1; in this case, we would use only the Process 1 primitive events for computing ordering relations. Thus,



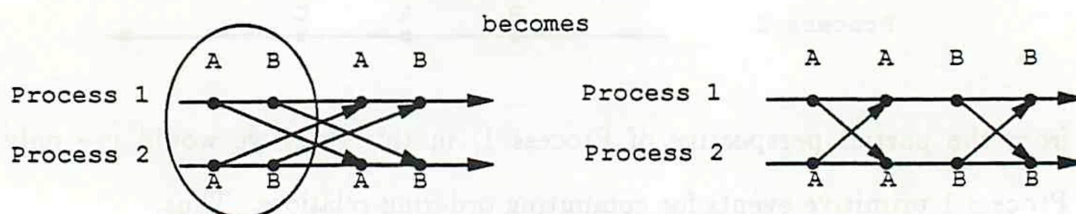
where circles indicate the events used to compute the ordering. From the perspective of the events on Process 1, abstract event A *precedes* abstract event B and abstract event C didn't occur at all. This ordering has been applied to all of the events in the behavior, including those on Process 2. In the reordered behavior we see that all elements of abstract event A are before those of abstract event B. Abstract event C has been deleted since it does not contain elements in the perspective. Alternatively, the same system can be decomposed from the perspective of Process 2, in which case



Each reordering is meaningful; each exposes the order of events on an individual process and provides some insight to the code and environment of that process. The two reorderings together characterize the original behavior of the system.

A perspective need not be confined to the events on a single process. Any subset of events can serve as a perspective. For example, in visualizing the following pair

of "message exchange events," we might use send operations as the perspective, in which case,



because each process sends a message as part of the A exchange before it sends a message as part of the B exchange. From the perspective of the send events, A *precedes* B. (The reordered behavior would have been the same, in this case, if we had used just the receives as our perspective.)

We define a *perspective view* to be a consistent or partially consistent reordering of an execution sequence from a perspective for the purposes of visualization. In the next section, we provide an algorithm for computing perspective views.

4.4 Algorithm for Computing Perspective Views

The perspective view algorithm has been implemented in Belvedere as a phase that processes an animation prior to display, as shown in Figure 4.1. Although the algorithm is implemented as a phase in Belvedere it can also be used as a preprocessing step in other visualization tools. The algorithm requires only suitable sets of events (which can contain both primitive and abstract events) representing the of events to be viewed and the perspective, and (optionally) a set of dependencies imposed by the display system. This last requirement is discussed in more detail below.

The algorithm permits hierarchical definitions of abstract events. Each event is represented as a directed acyclic graph in which the nodes are events (primitive

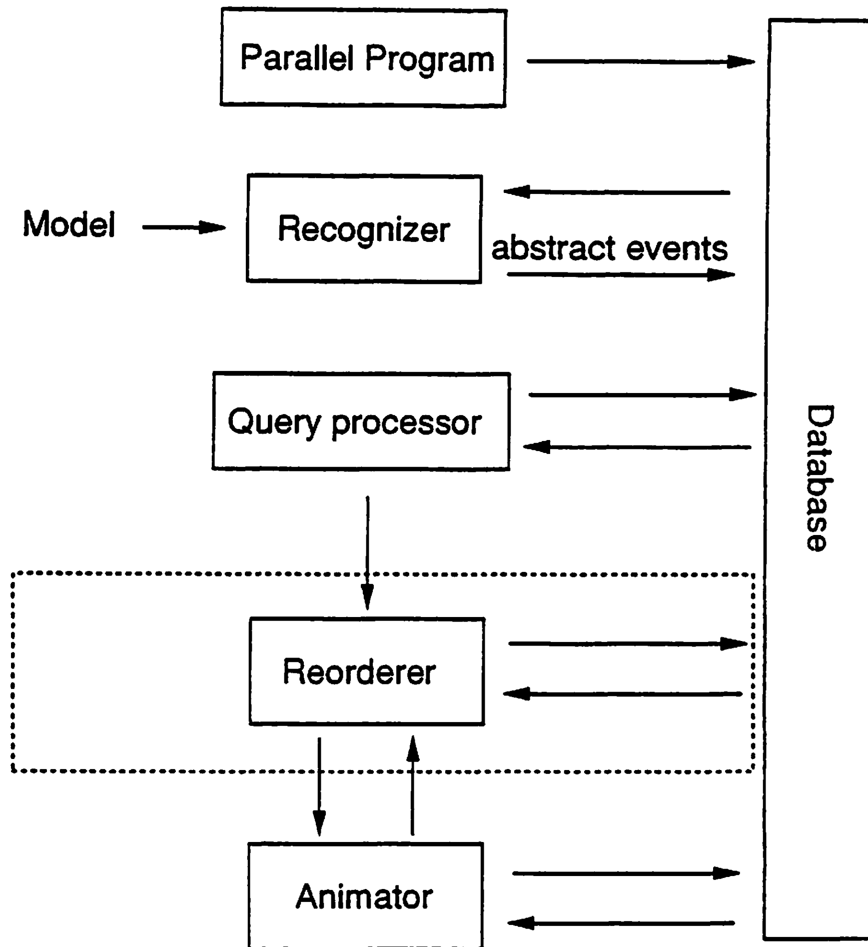
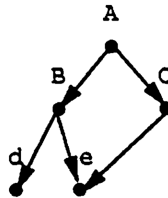


Figure 4.1 Belvedere with reordering algorithm.

or abstract) and the edges signify "contains". Thus, for example,



event A consists of the two abstract events B and C; event B consists of the two primitive events d and e; and abstract event C contains the single primitive event e.

Our algorithm constructs a graph that represents the entire behavior of the system by adding dependency edges between nodes in these graphs.

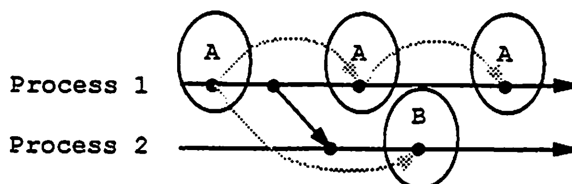
There are three inputs to the algorithm: (1) the set of events to be viewed; (2) the set of events defining the perspective; and (3) a set of dependencies representing constraints imposed by the display system. The dependencies are intended to insure that the reordered behavior can be successfully displayed. Our display tool, for example, requires that a message transmission be displayed prior to message receipt; the explicit inclusion of this as a constraint prevents such a dependency from being deleted in a partial perspective. Constraints are supplied to the perspective algorithm as a list of artificial *happened before* relations. All input events are represented as records that contain event attributes. An abstract event is represented as a variable length record that contains pointers to the records representing the immediate children of the abstract event.

The perspective algorithm has four steps:

1. *Construct a combined graph representation of all inputs.*

   Nodes in the graph represent events (both primitive and abstract). Edges in the graph come from three sources: (1) the binding of events to abstract events; (2) the constraints supplied from the display system; and (3) the *happened before* dependencies between primitive events. In the first case, we add a bidirectional edge between an abstract event and each of its immediate children, copying the information from the input record representing the abstract event. In the second case, edges are added to represent the dependencies introduced by display system constraints. In the third case, we add a minimal set of edges sufficient to generate the restriction of the *happened before* relation to the events of the perspective. Thus, for example, in the following, only edges representing the dependencies shown in gray are added

for the circled perspective:



Initially, the graph contains only nodes representing events in the perspective; other nodes are added as needed to complete dependency edges required from sources (1) and (2).

2. *Find the strongly connected components of the above graph.*

   The strongly connected components of our graph correspond to sets of events that are to be treated as distinct visual units. We use an algorithm developed by Tarjan[83] to find the strongly connected components.

3. *Assign depths to components.*

   A pseudo "origin" component is created and edges are created from the "origin" component to the components that have no incoming edges. The depth of each component is the length of the longest path from the origin to that component, counting along strongly connected components. To compute this length we first compact the graph so that each strongly connected component is represented by a single node; the compacted graph is acyclic. Path lengths are computed using a special case of the Critical Path Method[46, 25] normally used for job scheduling.

4. *Reorder events.*

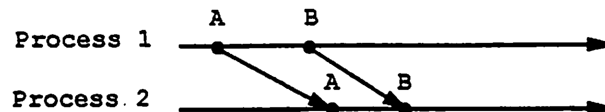
   The events are reordered so that all components at the same depth start at the same time. Each strongly connected component is moved as a unit. Reordering is accomplished with a sort on the depth of each component followed by a pass over the events in sorted order to assign new timestamps.

The algorithm computes a reordering based on a perspective views in $O(N \log N + E)$ time where N is the total number of primitive and abstract events in the system and E is the number of dependencies (ordering relationships) between the events. E can be as large as N^2 in the worst case but in practice it is considerably smaller. A more complete analysis of this algorithm appears in Appendix A.

We illustrate the algorithm with examples showing a full perspective, a partial perspective, and a "recursive" perspective that reorders events at multiple levels of abstraction.

4.4.1 Example of a Full Perspective

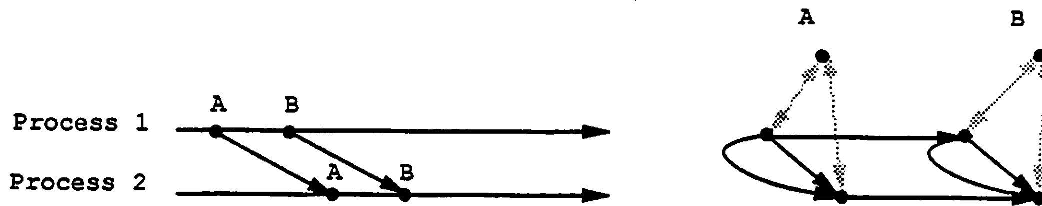
To illustrate the algorithm with a full perspective, we assume that the following behavior



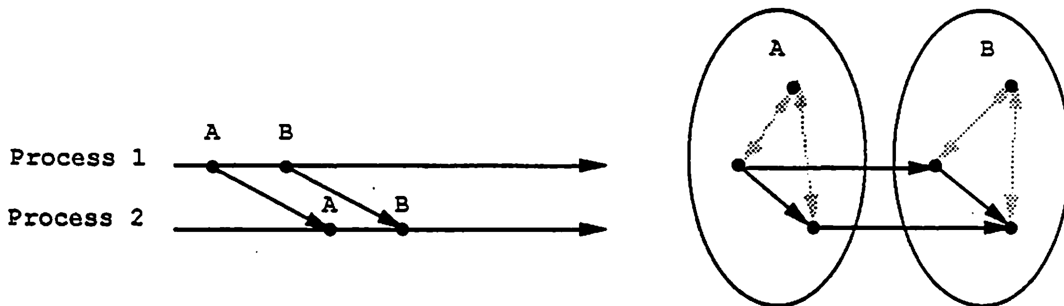
is to be reordered prior to animation.

Step 1: Dependencies are assigned. In the following example dependencies due to the sequential nature of processes and interprocess communication are shown in black and bidirectional dependencies between each primitive event and the abstract event that contains it are shown in gray. Because we intend to animate the behavior, application specific dependencies are added to insure that messages are always sent before they are received; these are indicated by curved arrows. (The curved arrows in the following figure indicate only that a dependency is also supplied by the display system. The graph being built is not a multigraph.) The

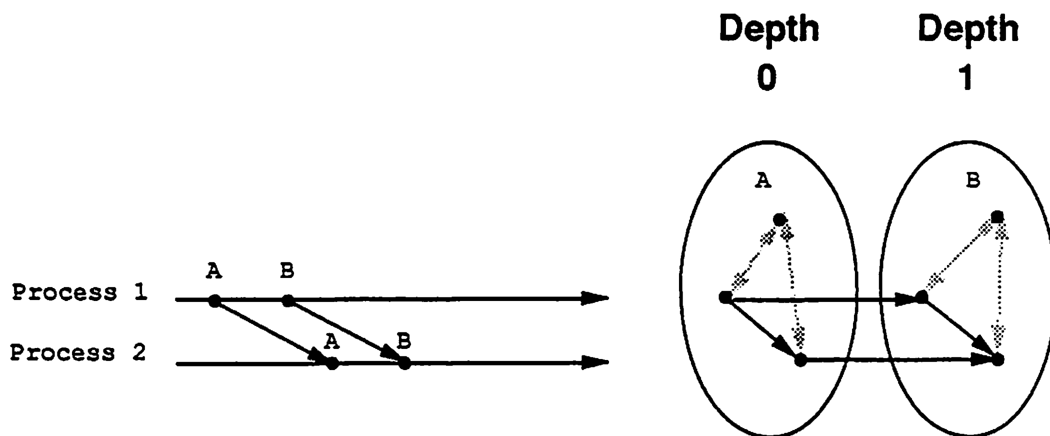
behavior and the resulting graph are:



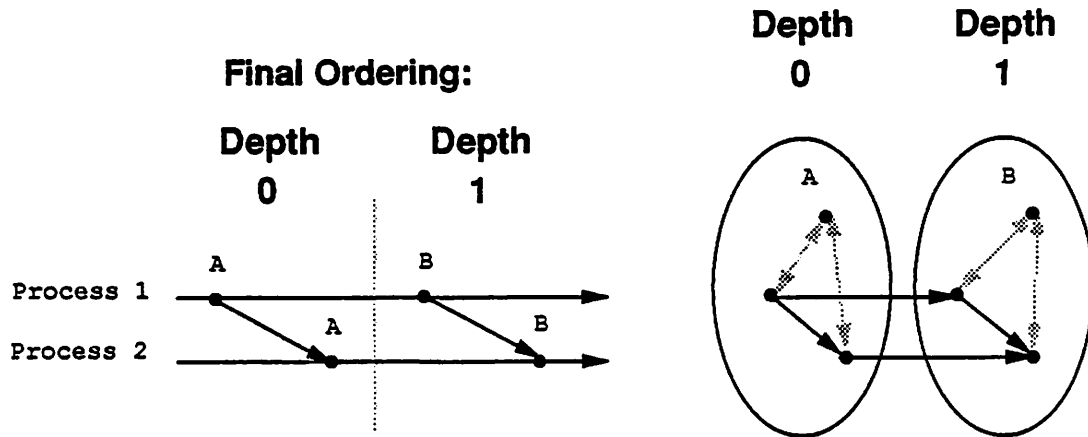
Step 2: The strongly connected components of the graph are calculated (shown circled).



Step 3: Depths are assigned to each component.

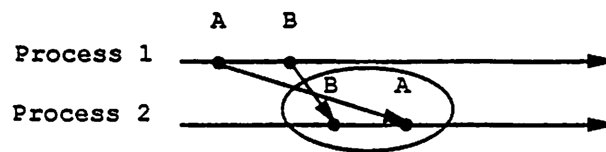


Step 4: The components are separated and moved to new times corresponding to the assigned depths.

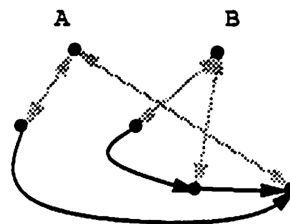


4.4.2 Example of a Partial Perspective

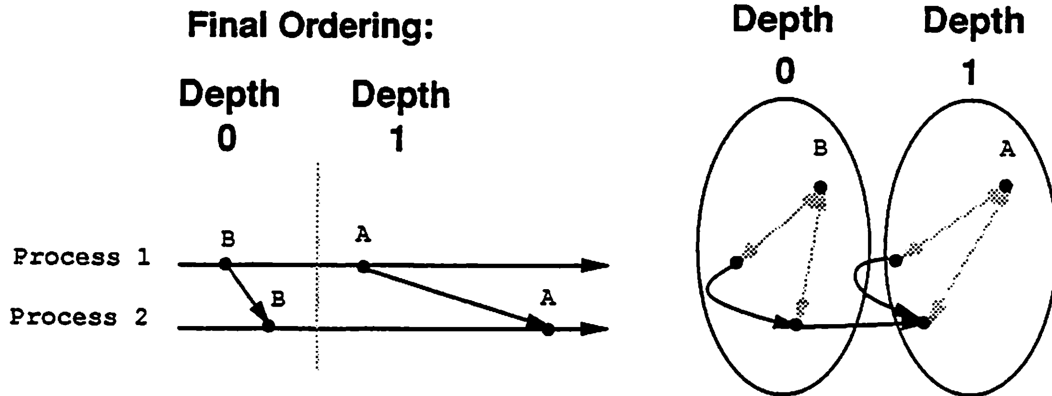
To illustrate a partial perspective, we assume that the following behavior



is to be reordered prior to animation. The same steps are followed except that edges of the first two types are only recorded for events in the perspective. When viewed from the perspective of Process 2, the behavior results in the graph

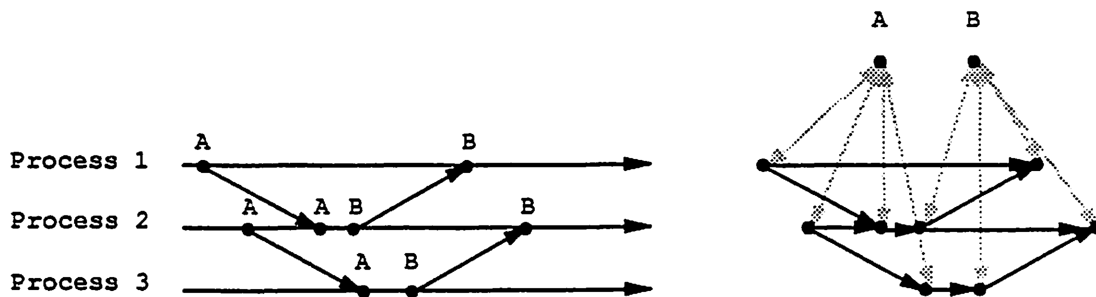


which, in turn, results in the reordering

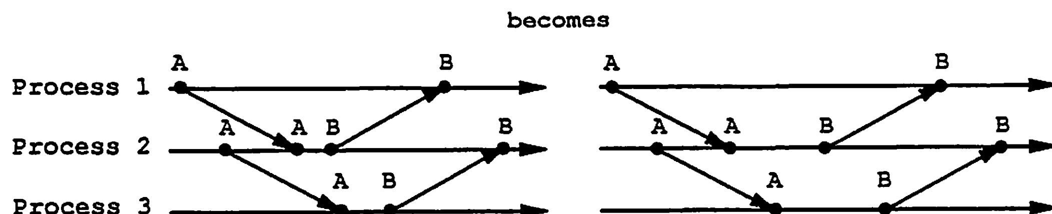


4.4.3 Example of a Recursive Perspective

A perspective view reduces asynchrony by changing the relative timing between abstract events but does not treat asynchrony within an abstract event. When the following system (shown with its graph),



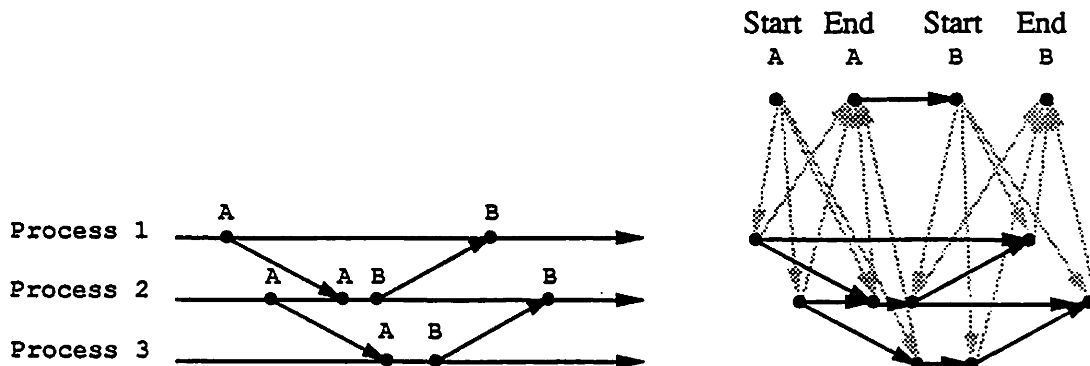
is reordered using a full perspective, it shows asynchrony remaining within the A and B events:



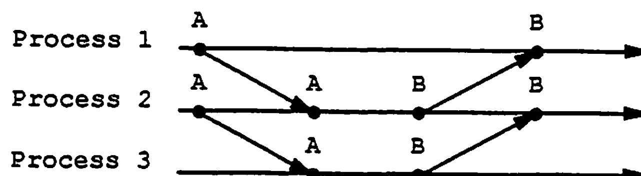
that is, the send events within the abstract events are *parallel* but they have not been moved to start at the same time.

However, since we allow abstract events to contain other abstract events, we can recursively apply the algorithm to sub-events within each abstract event. In the implementation, we do not actually apply the algorithm recursively but instead achieve the same effect by modifying the graph and running the algorithm as usual. The graph is modified so that instead of a single node to collect all of the dependencies in an abstract event, we use pseudo "start" and "end" nodes; the start node *happens before* all members of the associated abstract event and all members of an event *happen before* its end node. The bidirectional arcs between an element and its parent node are thus replaced with directional arcs from the start node to each element and from each element to the end node. When a *happened before* dependency is found between primitive events a and b, an arc is added between the corresponding end and start nodes of the topmost abstract events they belong to. (This usually adds no additional arcs, but in pathological cases this can force the time complexity of the algorithm to be $O(N \log N \, * \, E)$).

Using this "recursive" version of the algorithm, the above example results in the graph



and the reordering



Note that there are no nontrivial strongly connected components in this case. By not binding an entire abstract event into a connected component the algorithm has the freedom to assign different depths to the members of an abstract event while the associated start and end nodes insure that the members of an abstract event are all assigned depths that are less than the members of the following abstract event.

In a totally recursive perspective, such as the one used in the example above, there will never be any nontrivial strongly connected components. Our implemented algorithm is more general than shown here; it allows the user to select which abstract events are to be recursively reordered and which are not. In the general case the graph will contain both trivial and nontrivial strongly connected components.

4.5 Examples of the Utility and Generality of Perspective Views

Perspective views can be used to enhance visualizations of parallel systems. In this section, we demonstrate their application in three different types of visualization tools: tools producing process-time graphs such as Moviola [29]; tools facilitating user-controlled animations such as Voyeur [80]; and tools providing automatic animations such as Belvedere.

4.5.1 Process-Time Graphs

Several display tools[29, 38] produce process-time graphs showing processes along one axis and time along the other. We present two examples that demonstrate the use of perspectives in making these graphs more comprehensible; the first uses a full perspective and the second a partial perspective. It should be noted that we are primarily interested in visualizations for debugging purposes and that our temporal reorderings would not be useful in applications of process-time graphs to performance debugging.

Example 1. Successive Overrelaxation (SOR) [40]. In this iterative approximation to a partial differential equation (PDE), processes arranged in a square mesh repeatedly update their values as functions of their neighbors' values. Even

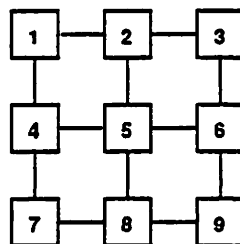


Figure 4.2 SOR mesh.

and odd processes alternate their execution: first odd processes send their current values to their neighbors while even processes read their neighbor's values and update their own; then the processes reverse roles. Processes send values in a counter-clockwise sweep beginning with the neighbor above them; processes read values in a complementary, clockwise sweep beginning with the neighbor below them. The resulting communication for the labeled mesh of Figure 4.2 is shown in the process-time graph of Figure 4.3 (we show process-time graphs in this section oriented the way Moviola displays them). Because of asynchrony, the communication patterns of these processes are difficult to see in the graph.

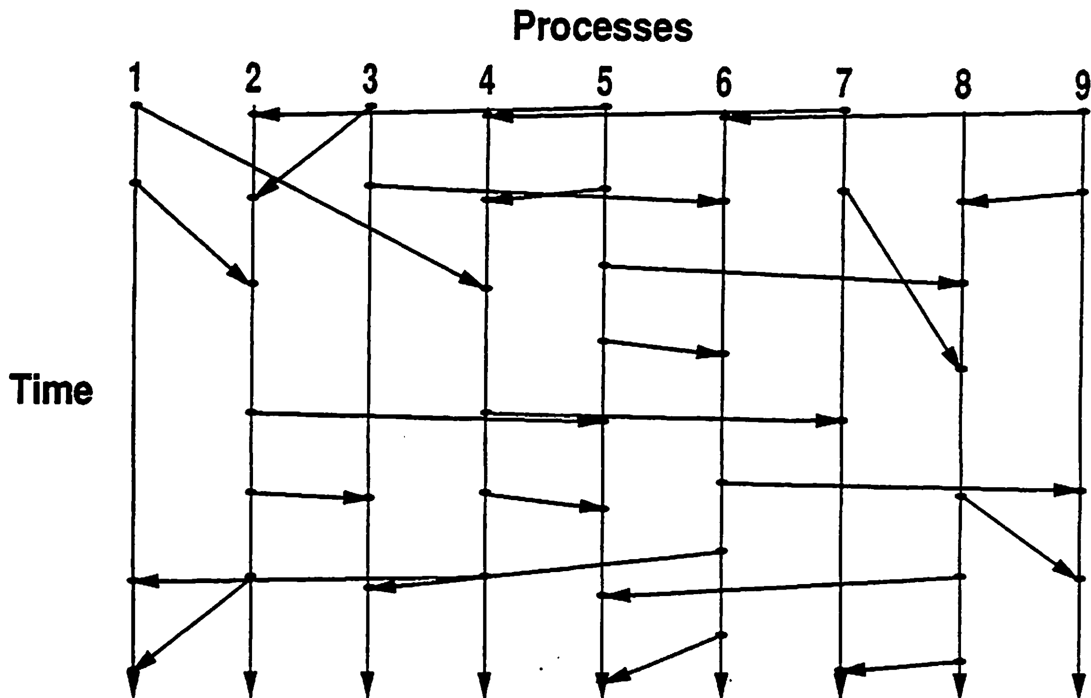


Figure 4.3 Primitive communication on a process-time graph for the SOR program.

To use our reordering tool, we first define a two-level hierarchy of abstract events: "messages" are defined as appropriately matched pairs of send and receives and are then used in defining abstract events of the form "north messages from even processes", "west messages from odd processes", *etc.* Because the processes send and receive in complementary sweeps, these abstract events are pairwise related by *precedes.* Thus, a recursive, full perspective successfully separates them into distinct bands as shown in Figure 4.4.

This visualization clearly shows the patterns of communication that the user intended. Because it is a consistent, full perspective, it preserves all relevant dependencies (that is, it does not change the *happened before* relation) and allows the user to view and debug his program at an appropriate level of abstraction.

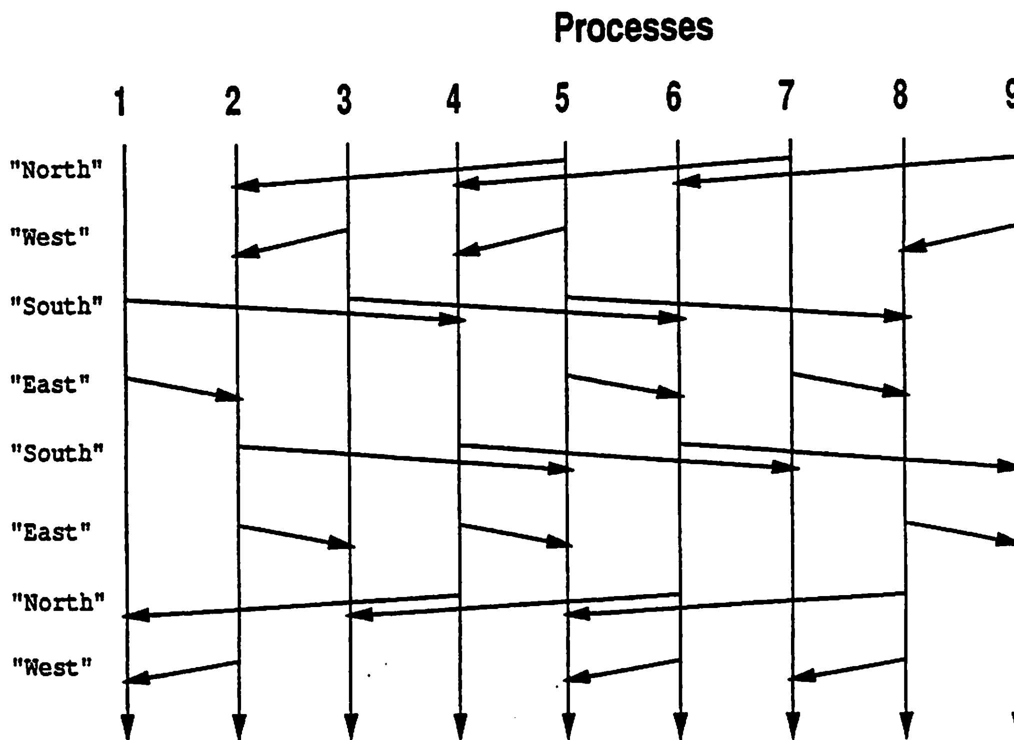


Figure 4.4 Logical communication on process-time graph from the SOR algorithm.

Example 2. Median Filter [70]. In this algorithm, each node repeatedly updates itself with the median of its neighbors' values. It differs from the SOR algorithm in that all processes execute simultaneously: each sends values to all of its neighbors, receives values from those neighbors, and then updates its own value. Both send and receive cycles use the same counter-clockwise sweep, beginning with the process above. Again, the patterns of communication are difficult to discern in the standard process-time graph, shown in Figure 4.5.

The abstract events of interest are similar to those used in the SOR example, events of the form "north messages from all processes", "west messages from all processes", *etc.* Unlike SOR events, however, these abstract events will not be related by *precedes* because each process overlaps its participation in many abstract

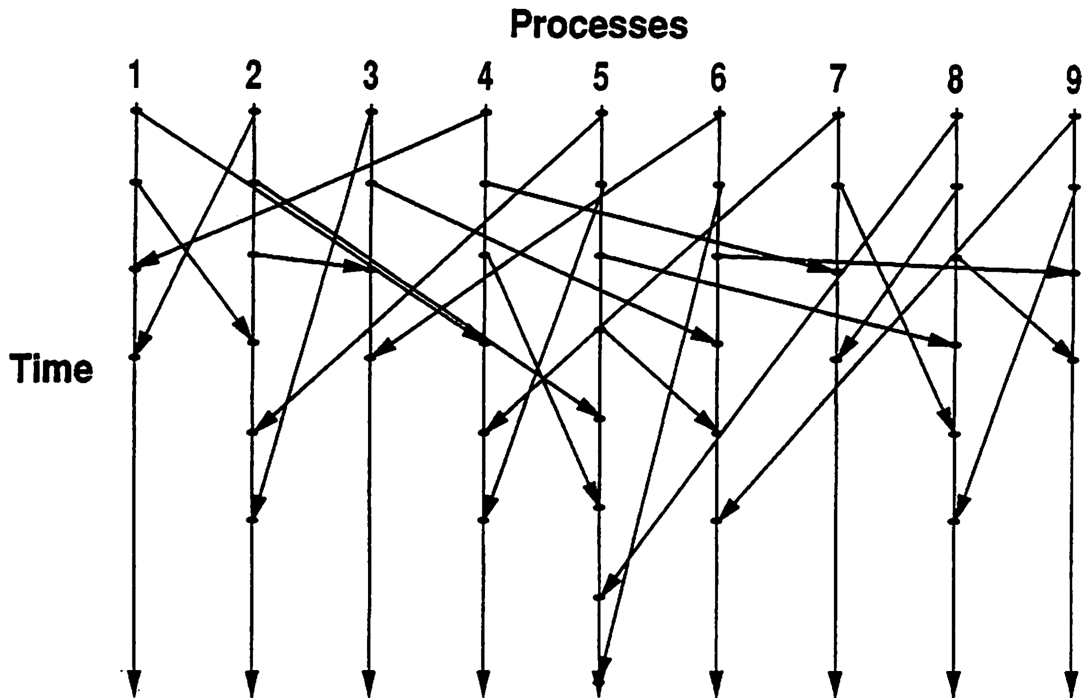


Figure 4.5 Primitive communication on process-time graph from the Median Filter algorithm.

events: first sending in all directions (to begin its participation in four events) and then reading from all directions (to end its participation in those events). Thus, for example, if we look at Processes 2 and 5, they exchange two messages, one as part of the "north message" event and one as a part of the "south message" event. These two events are related by *overlap* and they cannot be consistently separated in a full perspective.

To choose a partial perspective that will separate the events, we note that all processes send messages in the same counter-clockwise pattern. A partial perspective that includes only send events produces a process-time graph in which the abstract events are shown as bands of messages as in Figure 4.6. The visualization shows that at least part of the program's behavior is as the user intended. We can see that the proper number of messages are being sent, they are being sent in the

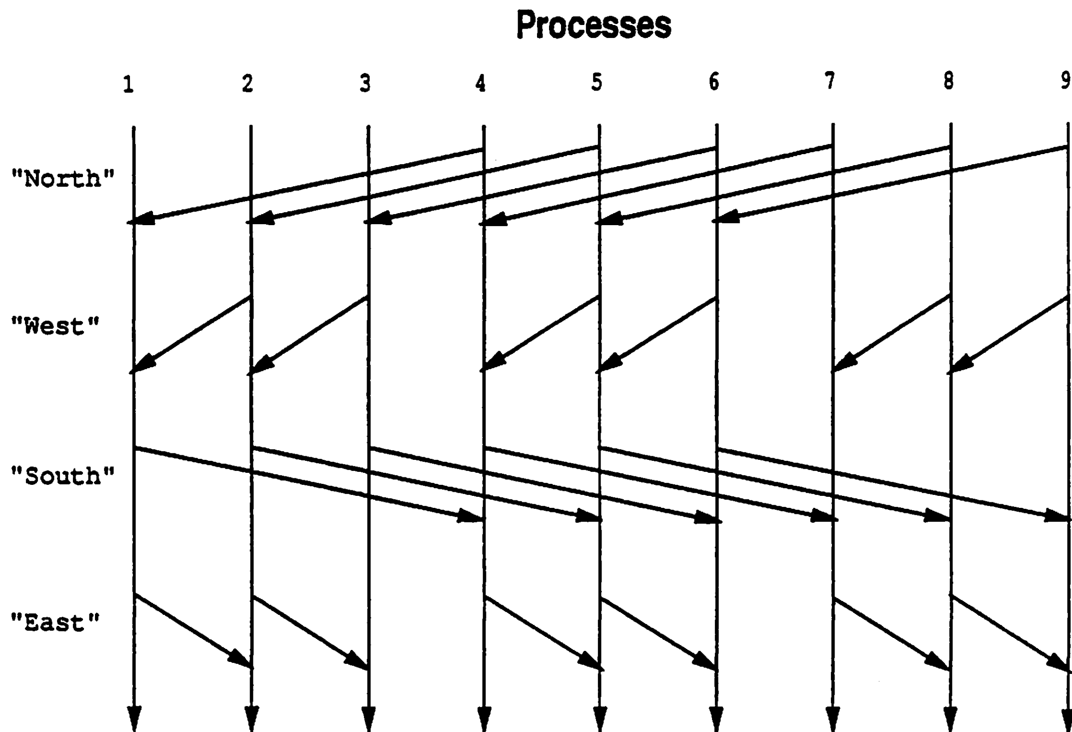


Figure 4.6 Logical communication on process-time graph from the Median Filter algorithm.

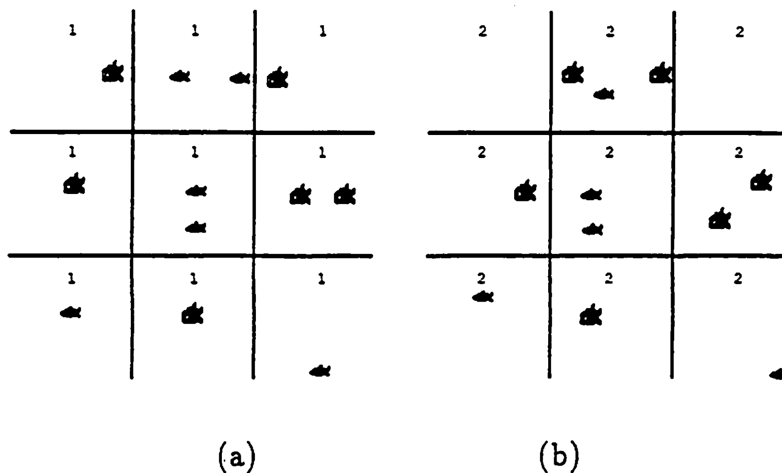
proper order, and to the correct recipients. Another partial perspective, based on the receive events, would be needed to allow us to see that the messages are being received in the proper order.

4.5.2 User-Directed Animations

User-directed animations can provide high-level, application-oriented visualizations of parallel programs. Unless special care is taken, however, even these animations will be difficult to understand.

Example 3. Sharks and Fish [80]. For this example, we use a load-balancing algorithm operating on an underwater simulation taken from a Voyeur paper [80]. The simulation contains sharks and fish that can move at most once per unit time;

sharks eat fish. Initially, the simulation might be configured as in Figure 4.7(a).[2]



(a) (b)

Figure 4.7 Logically consecutive time steps in the lives of sharks and fishes. Each process is labeled with its current time.

After a single logical timestep, it might be configured as in Figure 4.7b where both sharks in the first row have moved into the top middle square and the shark on the right has eaten a fish. The load balancing algorithm seeks to balance the number of animals under the control of each process.

Figure 4.7 displays the logical timestep for each process. The figure shows synchronized processes moving together from step 1 to step 2. If, however, the processes are not synchronized, the confusing display shown in Figure 4.8 might be seen. In that figure, some processes are at step 1 while others have advanced to step 2. The image is difficult to interpret; sharks in the top row, for example, are not displayed at all because during step 2 they have moved to processes shown at step 1. The problem is not in the code but in its visualization.

Either Voyeur or perspective views can be used to produce the coherent animation shown in Figure 4.7. In Voyeur, the user explicitly simulates a global clock by associating a generation number with each process action during execution.

[2]The figures in this section were not produced by Voyeur. They are hand simulations of Voyeur's actions based on the published example and conversations with the author.

Voyeur displays all process actions with the same generation number at the same time. In Belvedere, the user defines abstract events after execution that correspond to single movements of sharks and fish on all of the processes. Such events are ordered by *precedes*, and thus are automatically separated by a full perspective to produce the desired animation.

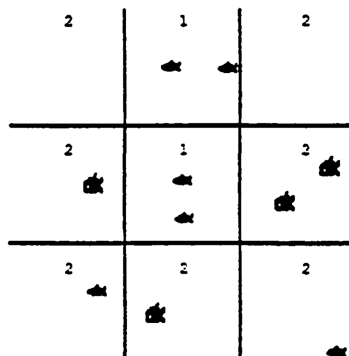


Figure 4.8 Actual execution of the Sharks and Fishes program. Processes at different time steps are executing concurrently.

4.5.3 Automatic Animations

Perspective views were developed and to enhance the automatic animations provided by Belvedere. In the following examples, we demonstrate the use of these reordered animations to aid comprehension of program behavior.

Example 4. Successive Overrelaxation. The cycles of odd and even execution of the processes in the SOR program should produce a "checkerboard" pattern of communication. A snapshot from the animation of the program in Figure 4.9 is difficult to interpret directly; in the figure we see that Processes `s12`, `s21`, and `s23` are participating in a "west messages from even processes" event, and Processes `s11`, `s13`, and `s24` are participating in a "north messages from even processes" event.

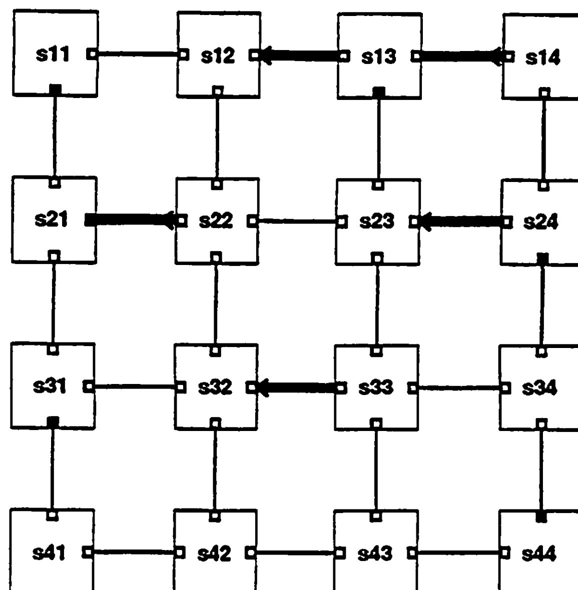


Figure 4.9 Primitive event animation of the SOR program.

The abstract communication events in this program are related by *precedes*. A full perspective view will thus show the expected checkerboarding pattern as shown in Figure 4.10.

Example 5. FFT on a Butterfly. The Fast Fourier Transform(FFT) algorithm on a Butterfly has a very regular communication structure in which each process reads its two input values, computes a new value, and sends the new value to two processes in the next lower row. The expected communication pattern is a wave of orderly activity moving through the butterfly from the top to the bottom. Even though processes perform identical actions and there are no conditional statements in any processes, the expected communication pattern is distorted by asynchrony as shown by the snapshots in Figure 4.11. Processes start computing simultaneously. Processes in the first row of the FFT send messages at two times to propagate their value to two recipients in the next row. However, processes in the following row read these messages in different orders depending on their position in the butterfly. Processes in the left half of the butterfly read messages in the order

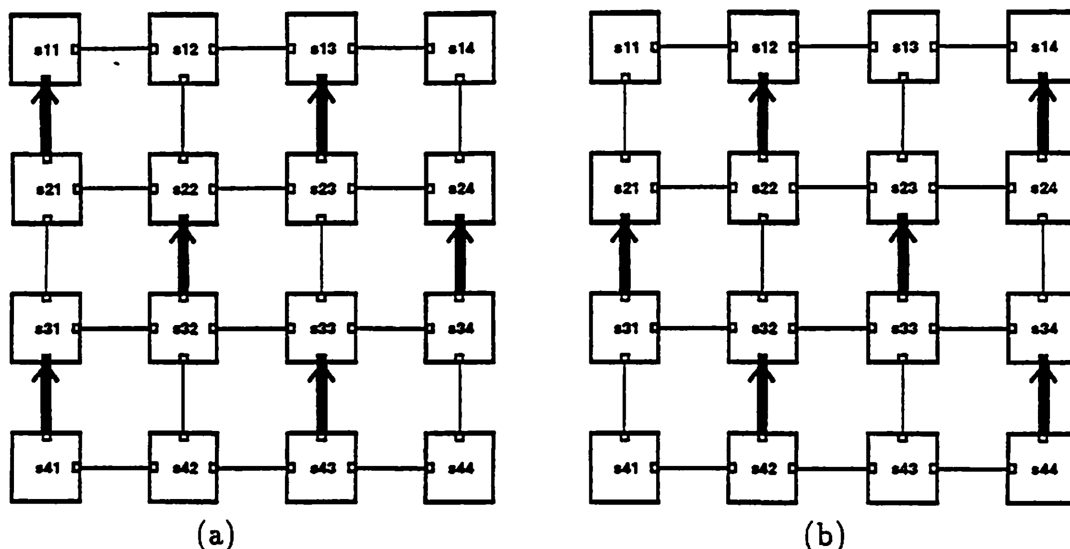


Figure 4.10 Successive overrelaxation with abstract events. Odd process communication events to the north (*a*) and Even process communication events to the north (*b*).

they were sent; processes on the right half then read in the reverse order. This sets up an asymmetry because the processes in the right half wait until the second message from the first row arrives before reading any messages. Skewing results and increases with each row. Figure 4.11(b) shows communication between rows two and three and between rows three and four at the same time. From a sequence of images such as those in Figure 4.11 it is difficult to follow the flow of the algorithm.

To see if the communication was occurring as expected we model the communication from one row to the next as two abstract communication events corresponding to the first and second message each process sends. Viewing the abstract behavior from the perspective of the "put" events shows the first and second waves of messages between each row as shown in Figure 4.12, and allows us to determine that the communication was proceeding as expected.

Example 6. Gridsort. In the previous examples, we have shown that asynchrony can cause activity from different phases of an algorithm to occur simultaneously

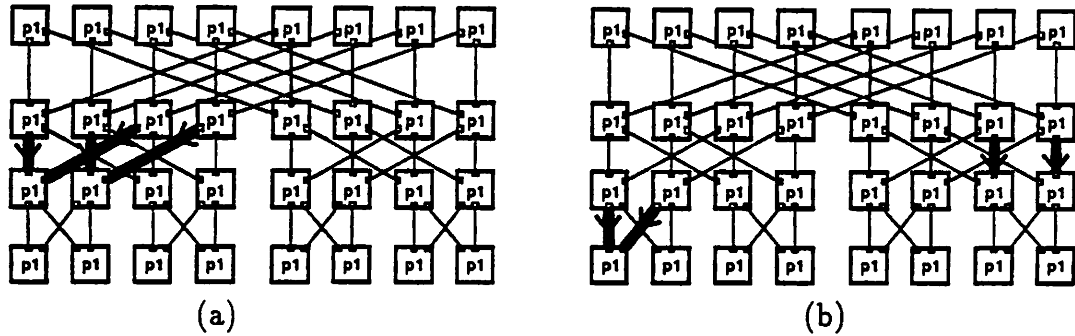


Figure 4.11 Low level communication behavior of the FFT. Communication near the start (a); and a short time later (b).

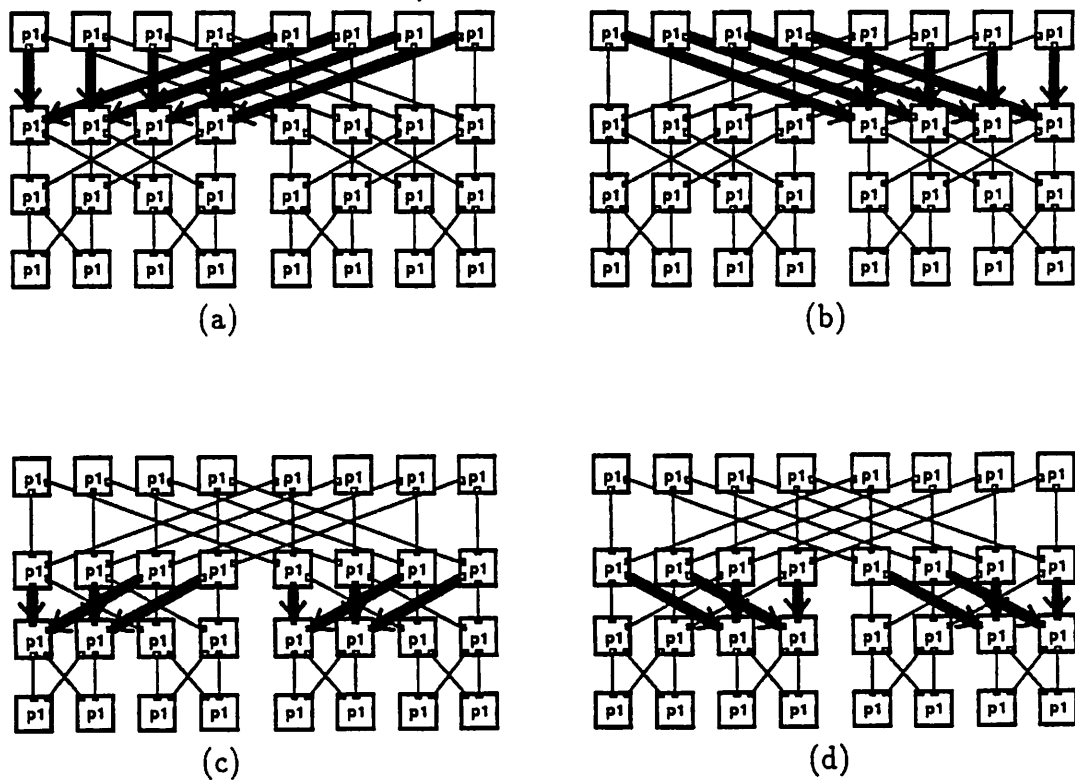


Figure 4.12 Abstract communication behavior of the FFT. First wave of messages between the first and second row (a); second wave of messages (b); first wave of message between the second and third row (c); second wave of messages (d).

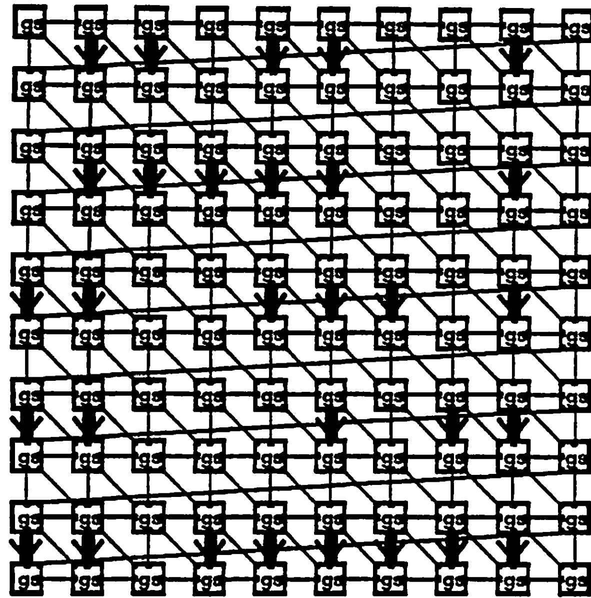
and make the low level visualizations of behavior difficult to understand. Here we show that asynchrony can prevents us from understanding the behavior of a single phase, even if it doesn't overlap with any other phases.

The gridsort algorithm[90] sorts a grid of numbers through a series of nearest neighbor exchange phases on a 2-D mesh. In the first phase processes in even rows compare and exchange values with the row above. In following phases processes exchange values between columns and along diagonals and then compare and exchange values starting from odd rows, columns and diagonals. The phases are repeated until the grid is sorted.

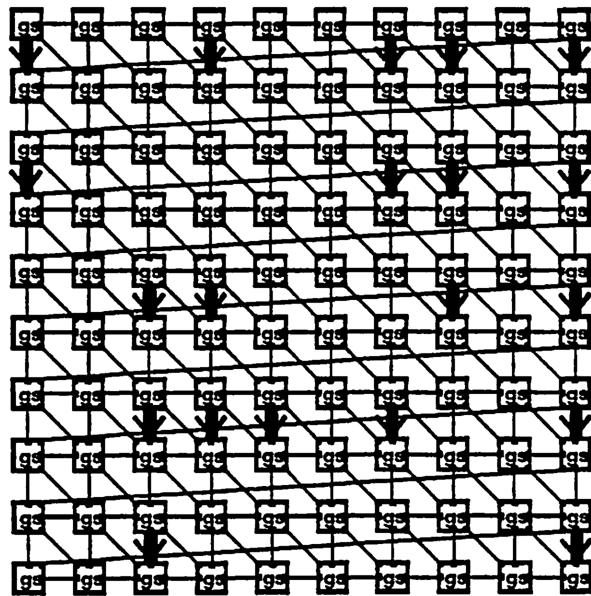
Figure 4.13 shows two snapshots from an even row exchange phase near the start of the program. Only behavior from the row swap is shown in these images but asynchrony has made it difficult to determine if all processes in the phase are participating in the swap. In fact, there are communications from processes in the row swap that are not shown in either snapshot. To improve the visualization we model the row exchange as an abstract event and use a full perspective view. This produces the two snapshots shown in Figure 4.14. From these images it is easy to see that all processes are participating in the exchange as expected.

4.6 Conclusions

Visualizing the execution of parallel programs is difficult because low-level primitive events do not reflect the programmer's view of the system. Program behavior is better understood at a higher level in terms of user-defined, abstract events. Visualizations of abstract events, however, are often obscured by concurrency and asynchrony. Perspective views enhance visualization by reordering behaviors so that abstract events can be seen as distinct units. For program executions that cannot be consistently reordered, partial perspectives can be used to transform the

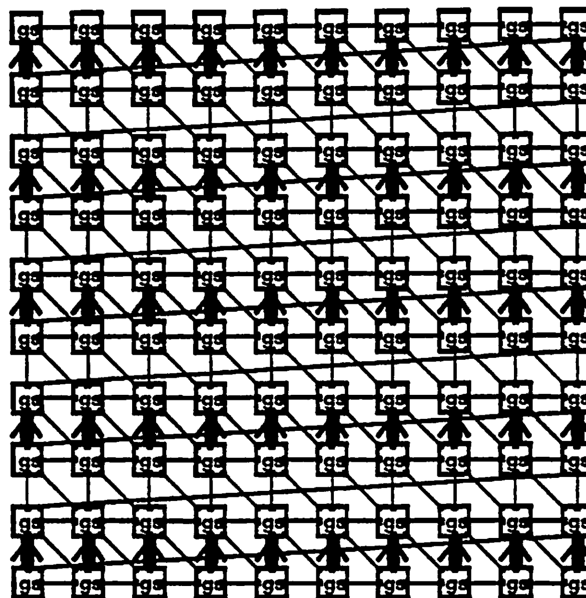


(a)

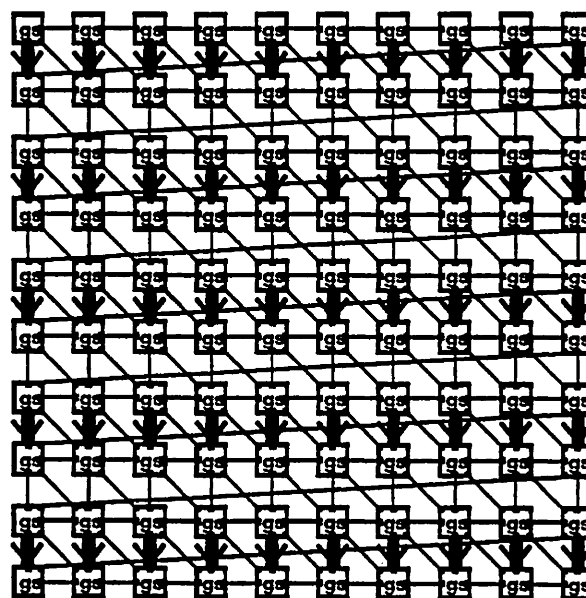


(b)

Figure 4.13 Two snapshots from a standard visualization of an even row swap in the Gridsort program.



(a)



(b)

Figure 4.14 Two snapshots from a perspective visualization of an even row swap in the Gridsort program.

system to one in which only user-selected temporal relationships are preserved. It is easy to create a variety of partial perspectives for the same behavior.

One concern about the use of perspective views is that because of the temporal remappings it is difficult for the user to interpret an animation because it may be difficult to tell which event orderings are "true" and which are "false". In practice we have not found this to be a problem. The user knows what ordering to expect because the ordering has been specified using the perspective set. Regardless, Belvedere has an animation style that renders events in the perspective differently from differently from those not in the perspective. In practice we have not found this animation style is needed to resolve confusion.

We have applied perspective views to the animations of parallel programs generated by our debugger and have found them to be effective in enhancing the depiction of high-level behaviors. These temporal remappings can also be used in conjunction with other types of visualization tools, as we have demonstrated. We have found that their use significantly aids the understanding of parallel program behavior.

CHAPTER 5

MODELING

In Chapter 2 we briefly reviewed modeling languages used for debugging. Here we provide a more detailed review of the modeling language issues and look at two such languages in depth: Bates and Wileden's Event Definition Language (EDL)[12] and Hseush and Kaiser's Data Path Expressions (DPE)[44]. With these languages as examples, we discuss problems with existing modeling languages and introduce a new modeling language.

A model is a specification of a set of events in terms of constraints on their attributes and relationships. A model is *matched* to a program behavior by searching the program behavior for events that meet the specified constraints. A set of events that meets the constraints is an *instance* or *match* of the model. A single model can have many instances.

Existing modeling languages are sensitive to changes in relative process speed. If the relative speed of the processes changes on different executions of a program, the results of matching may also change, even if the variations in relative process speed are invisible to the processes themselves. In addition, existing languages are sensitive to program errors. A program bug can cause events to be added to or deleted from the program's behavior, which can have the same effect as changes in relative process speed: the number of instances of a model can change and the grouping of events into instances can change.

We would like a modeling language to be as *specific* and *robust* as possible. Specificity means that changes in relative process speed that do not affect the

happened before relation do not change the instances of a model. Robustness is the property that the addition of an event due to a program bug can add new instances but not change the grouping of the original events into instances and that the deletion of an event can delete instances hierarchically containing that event but not otherwise change the original grouping.

To address these issues we have developed a new modeling language, PEDL. It uses the relationships **precedes**, **parallels**, and **overlaps** together with the concept of perspective views to permit models that are more specific and robust than those built in existing languages.

Section 5.1 examines issues in modeling languages. Section 5.2 introduces our approach and modeling language. Section 5.3 describes its implementation in an extended finite-state machine formalism. Section 5.4 presents examples showing the advantages of our language over other modeling approaches. Section 5.5 presents our conclusions.

5.1 Background

Most existing modeling languages are based on regular expressions[60, 43, 12, 17]. Regular expressions are used to describe sets of model instances, where each instance is a sequence of events. Existing modeling languages extend regular expressions by providing a means to express constraints in terms of event attributes. The following EDL model, for example, specifies that an instance of model X is a sequence of primitive events a, b, and c where the "color" attributes of the primitive

events must match:

X is a o b o c

filter

b.*color* = a.*color*

c.*color* = a.*color*

end

For parallel and distributed systems, the languages often add operators that explicitly model parallelism; we consider two such operators in detail later in this section. In addition, modeling languages often permit partial specifications of behavior which selectively ignore events. Thus for example, it is possible to model only the a,b, and c events in a program that produces d, e and f events as well. Finally, modeling languages may admit hierarchical descriptions as illustrated in Figure 5.1. In the figure, a model of the desired behavior, Z, is defined as a behavior consisting of the sequential composition of two other behaviors, X and Y; X and Y are defined as the sequential composition of primitive events a,b,c and d,e respectively. The example illustrates a simple hierarchical model of a sequential

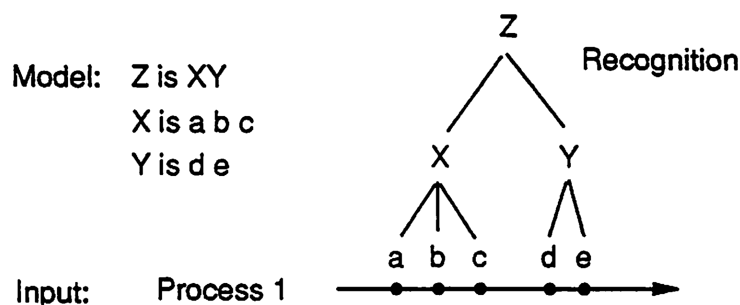


Figure 5.1 Hierarchical modeling.

behavior; for parallel behaviors the language must include a notion of ordering between abstract events.

Modeling languages take advantage of the relationship between regular expressions and finite state automata, using modified automata to find instances of models. We discuss how automata are modified and how they are used to find instances of models in the next section.

5.1.1 Comparing Modeling Languages

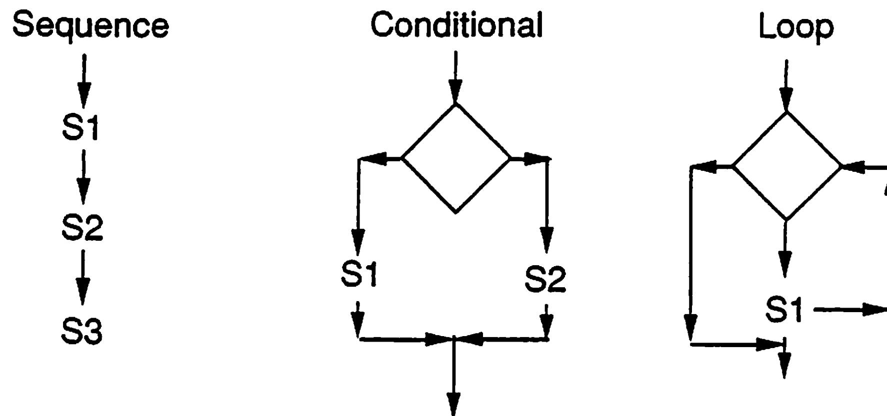
Each modeling language has a different set of operators and features. Comparisons based on the expressive power of the associated recognizers, that is, with respect to the Chomsky hierarchy, are not rewarding because after the required extensions to finite state automata, most recognizers are equivalent in power to Turing machines. Instead we rely on a qualitative comparison. We characterize modeling languages in terms of their features, using characteristics that include representations of program behavior, language operators, implementation formalisms, and methods for matching models to events.

Representation of Behavior. The representation of program behavior determines the kinds of operators that can be used to describe behaviors. Most often, behavior is represented as a string of events and described by the normal regular expression operators (Path Expressions[17], EDL[12], TSL[60] and Zave's language[94]). Alternatively, it may be represented as a partial order of events based on the *happened before* relation (Hseush and Kaiser[44]).

Language Operators. Standard regular expression operators model sequential behavior well because there is a natural correspondence between the operators and the standard control flow constructs as shown in Figure 5.2.

There is less agreement on how to model parallelism, in part because control constructs for parallelism are not standardized. We show examples of two

Control Constructs:



Models

S1 S2 S3 S1 | S2 S1 *

Figure 5.2 Modeling sequential program control constructs.

parallel operators, one using nondeterminism and one using *happened before*, later in this section.

Matching Interpretation. A model may be matched to a behavior in different ways. One view (used by Baiardi *et al.*[6] and by Zave[94]) is that the model must match *all* input events; no skipping of events is allowed. An opposing view (used in TSL[60] and EDL[12]) is that the model can match *any* of the input, skipping over intervening events as needed. The first view is similar to parsing; the second view is referred to as pattern extraction. The difference is illustrated in Figure 5.3 which shows a model and the behavior it matches under each interpretation.

With parsing, there is a single match of X and then an error. With pattern extraction, there are four matches of X; each primitive event can be matched many times and instances of X share primitive events.

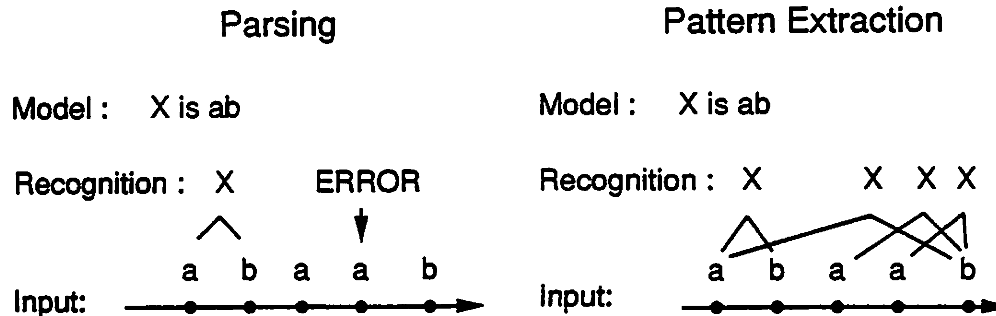


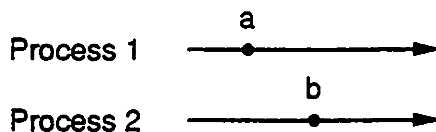
Figure 5.3 Parsing vs. pattern Extraction.

The matching mechanism affects error handling. Parsing approaches require that the input stream have a complete, unambiguous interpretation. When the automata cannot consume an input event, the automata fails and an error is signaled to the user. A pattern extraction approach offers no error handling capability because the recognizer attempts to find all possible instances of a model using multiple copies of the automata. When an automata cannot consume an input event, it is silently discarded.

Implementation. Automata extensions to implement language features must be implemented efficiently. Seemingly simple extensions to the automata can result in exponential size or time requirements.

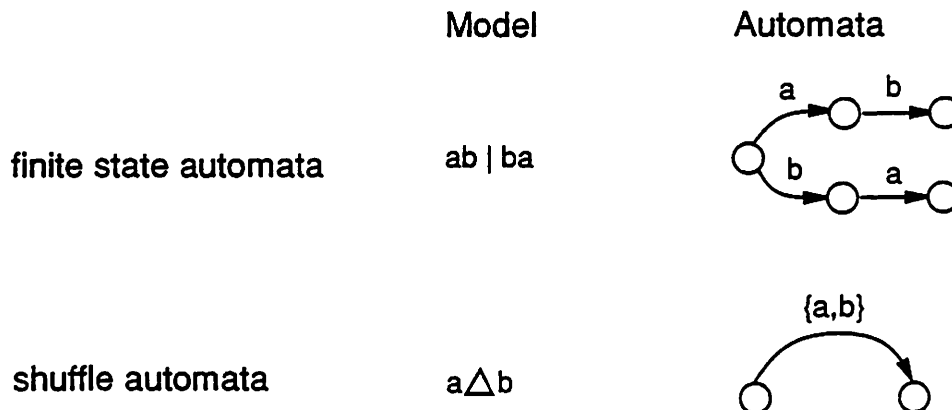
EDL is a string based language that uses the relative position of events in the string to determine ordering relationships between events. EDL supports the standard regular expression operators to model sequential behaviors and the *shuffle* operator to model concurrency. The shuffle operator ($\triangle$) is a string operator that matches any interleaving of its operands. For example, the expression $\mathtt{a} \triangle \mathtt{b}$ matches the string "ab" or the string "ba". The shuffle operator models concurrency by matching events independent of their relative order. For example, in the following

behavior,



the a and b events may occur in either order depending on the relative process speeds, resulting in either the string "ab" or the string "ba". The shuffle operator models this nondeterminism.

The shuffle operator is efficiently implemented as an extension to finite state automata — *shuffle automata* — in which arcs are labeled with sets of events instead of single events. A transition occurs only when the automata has seen all members of the set. In the following example,



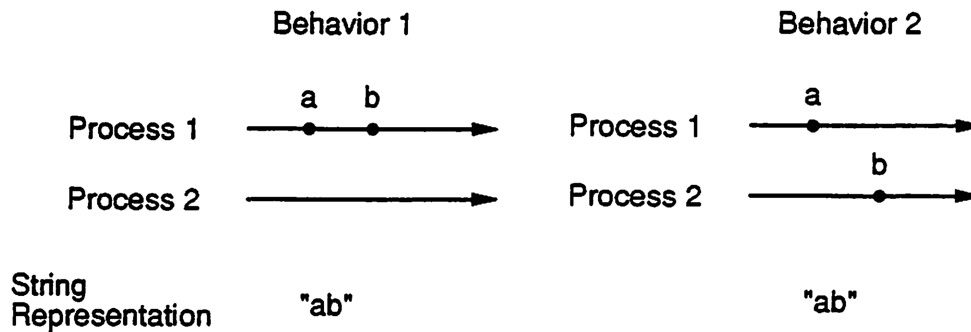
the finite state machine explicitly represents the possible interleavings of the strings "a" and "b". In the corresponding shuffle automaton there is a single transition which handles the shuffle of a and b. The shuffle automata will recognize either "ab" or "ba" because the events corresponding to the labels on the arc can occur in any order. Shuffle automata compactly encode the information necessary to recognize an interleaving of strings.

EDL uses hierarchical modeling which permits modeling expressions to contain both primitive and abstract events. Standard regular language based operators

operate only on single symbols and use the position of events in the string to determine ordering. To allow them to operate on abstract events Bates's EBBA system assigns a symbol and a position in the string to each abstract event as it is created. The symbol is the name of the model and the position is the position immediately following the rightmost event contained within the abstract event. This choice of position for abstract events also defines the notion of sequence between abstract events: given abstract events A and B, the sequential expression AB is true if B's position is greater than A's, or equivalently, B finished after A finished.

Bates's system uses a pattern extraction as discussed above but allows the user to control matching by restricting the sharing of events. A pure pattern extraction interpretation matches all possible instances of a model. Most of these instances are of no interest to the user. For example, there are many different instances of event X in Figure 5.3 but the user is probably interested only in some subset of the instances where the abstract events do not share primitive events. EDL provides two control mechanisms: a coarse mechanism that prevents groups of models from sharing events with each other, and a finer mechanism that controls the assignment of an event to a model when sharing is prevented. The default action when sharing is prevented is that the first instance of a model that can accept an event consumes that event.

Approaches that use strings to represent program behavior obscure some temporal ordering relationships. The following two behaviors, for example,



are different — they have different *happened before* relations — but their string representations are identical. Without using process attributes, it is not possible to build an EDL model that distinguishes between Behavior 1 and Behavior 2. This lack of discrimination can lead to models that are highly sensitive to process speed. For example, in Figure 5.4 the model "a followed by b" matches different events depending on the relative speed of Processes 1 and 2. Although it is possible that

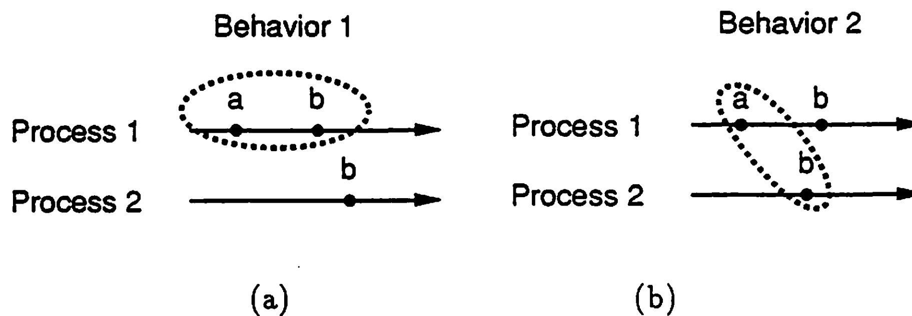


Figure 5.4 Model whose recognition is sensitive to process speed. Matching of the model "a followed by b" in a program behavior (a), and a slightly different execution of the same program (b), under a matching interpretation where only one instance of a model may consume an event. The match is shown with a dashed oval.

both matchings are acceptable, it is more likely that the user is trying to model the behavior as "a followed by b" because he or she believes that b must happen after a because of control or data flow considerations. In these cases the matching given

in Figure 5.4(b) would be unacceptable because there is no dependency between the events matched.

A similar problem exists with the shuffle operator. If, for example, we use "a $\triangle$ b" to model "a in parallel with b", we could again have both of the matchings shown in Figure 5.4. The matching in Figure 5.4(a) would probably be unacceptable to a user expecting "a in parallel with b". As another example, consider a program in which all processes are executing a loop; each process producing an "a" event on each iteration. For a two process system, we could model a single iteration of this program as an abstract event with the EDL model "$a_1 \triangle a_2$". (Subscripts are used in EDL to identify different instances of the same event type so that constraints can be applied individually.) The expected matching of the model is shown in Figure 5.5(a). An unintended, but equally valid matching is shown in Figure 5.5(b).

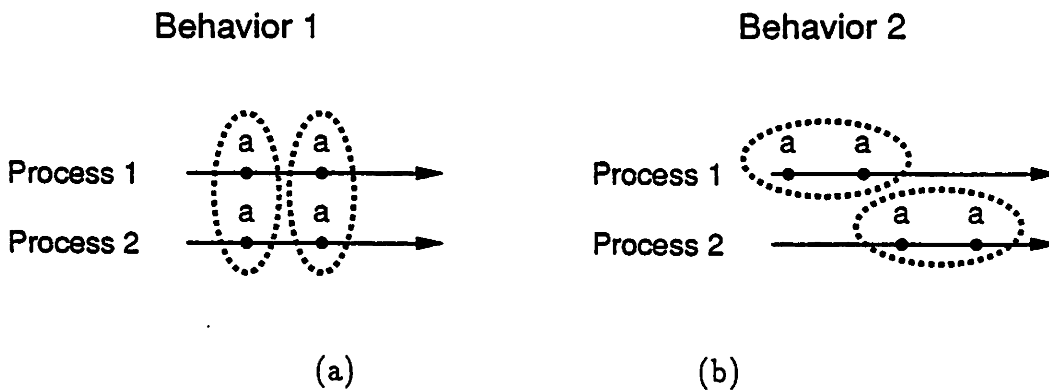


Figure 5.5 A matching of repetitive behavior that is sensitive to process speed. Matching of model "$a_1 \triangle a_2$" in a program behavior (a), and a slightly different execution of the same program (b).

For the previous examples, we could construct an EDL model that is not sensitive to process speed but we only if we were willing to add constraints on process identifiers, timestamps, and interprocess communication. The following EDL

model,

> *ab_sequence is* a ∘ b
>
> *filter*
>
> a.*processId* = b.*processId*
>
> *end*

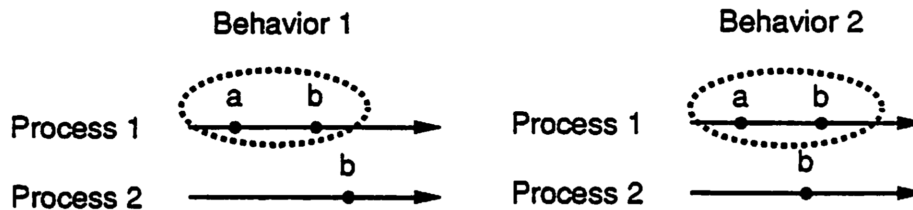
for example, matches the behavior of Figure 5.4(a) but not that of Figure 5.4(b) because it constrains a and b to be on the same process. In this case, the additional constraints were simple, but Figure 3.16 shows a more realistic example in which a large number of constraints were needed. Models that use such additional constraints are cumbersome. Constraints used to specify the lack of a dependency between events, for example, would have to describe all the ways the processes, timestamps, and messages cannot occur. In addition, constraints that use process identifiers explicitly inhibit scalability and reusability: new models are needed whenever the number of processes associated with the program changes and all levels of a hierarchical model must use the process identifier attributes of the lowest level events.

These problems can be addressed by basing the modeling language representation and operators on dependencies instead of global timestamps, as was done in Hseush and Kaiser's Data Path Expressions (DPEs). In a dependency-based language input behavior is represented as a directed graph where nodes represent events and edges represent *happened before* relationships. The standard regular expression operators are redefined in terms of this graph: *sequence* is defined as the existence of an edge between two events and Kleene star is defined as the closure of sequence. An operator added to represent parallelism defines parallel events to be *unordered*:[1] two events are *parallel* if there is no edge between them. The closure

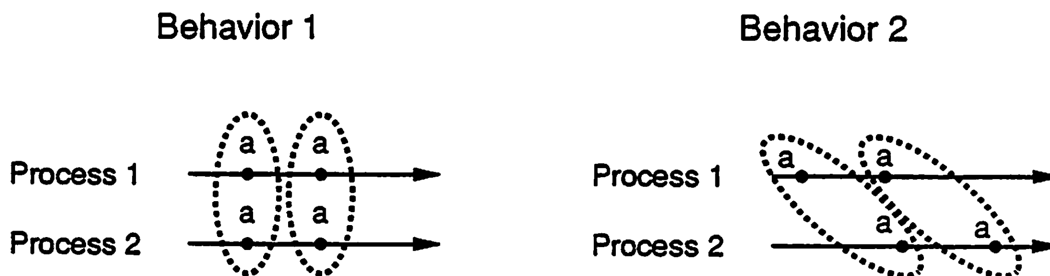
[1]In Section 5.3.1 we contrast this definition with the string based definition used in Petri Nets.

of the parallel operator is defined as a set of mutually *parallel* events such that no two events in the closure have an edge between them.

Modeling with operators based on a *happened before* yields models that are less sensitive to changes in process speed. More specifically, as long as the *happened before* relations stay the same, changes in process speed do not affect matching. For example, again using a pattern matching approach where only one instance of a model may consume an event, the model a *happened before* b matches the same events in the following behaviors despite the difference in relative process speeds:



In the example of Figure 5.5, the model "a in *parallel* with a" will match the intended behavior regardless of asynchrony:

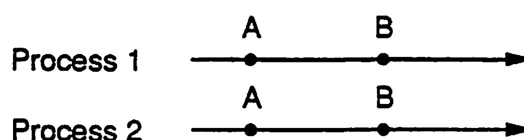


The instances of the model are the same in both behaviors because there are no dependencies between the first "a" event on Process 1 parallels and the first "a" event on Process 2 in both cases. The second events on each process also have no *happened before* dependencies between them.

The parallel closure operator has the same desirable properties as the parallel operator but provides a better means of modeling massively parallel behaviors.

Instead of writing a model such as "a_1 in *parallel* with a_2 in *parallel* with a_3 ...", where the number of "a" events must be specified in advance, the user can write a model using the parallel closure, such as "a_i *parallel**".[2] The model using parallel closure will match whatever degree of parallelism is actually expressed in the program's behavior. This frees the programmer from having to predict the degree of parallelism and scales when the degree of parallelism changes. We give an example demonstrating the utility of the parallel closure in Section 5.4.

We perceive two shortcomings of Hseush and Kaiser's system: its definition of ordering between abstract events and its implementation. In DPEs, two abstract events are sequential if and only if all of the primitive events of the first *happen before* all of the primitive events of the second. In terms of the graph representation, this requires an edge from each primitive event in the first abstraction to each primitive event in the second abstraction. This definition is too restrictive for modeling the kinds of behavior found in MIMD parallel programs: total pairwise orderings between the elements of abstract events rarely occur. For animation purposes, we would like to consider behavior such as

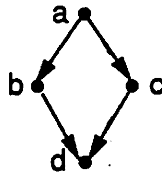


to be sequential: we would like to show first A and then B occurring. We would thus like to model this behavior as A followed by B.

A second shortcoming of DPEs is their implementation. Hseush and Kaiser proposed to implement DPEs in terms of finite state machines, called *predecessor automata*, that operate on directed graphs rather than strings. The input to a predecessor automaton is a sequence of edge expressions describing program be-

[2]This notation for the parallel closure operator is only for expository purposes.

Behavior as a graph:



Behavior as edge expressions:

(a .) (b a) (c a) (d c b)

Figure 5.6 A behavior represented as a graph and as edge expressions.

havior as a graph. An edge expression denotes a node and the predecessors of that node. For example, Figure 5.6 (taken from [44]) shows a behavior represented as a graph and as edge expressions. In the figure, the edge expression associated with node **a** is (**a** .), where the dot represents the absence of predecessors. Node **b** has predecessor **a** and is represented by the edge expression (**b a**). Node **d** has two predecessors and is represented by the edge expression (**d c b**). Transition arcs in a predecessor automaton are labeled with edge expressions: the machine transitions on an input if and only if the edge expression on the label of a transition arc exactly matches the incoming edge expression.

The construction of a predecessor automaton is analogous to the construction of a finite state machine from a regular expression. Small automata that recognize subexpressions are systematically glued together. Figure 5.7 shows a model and a predecessor automata for the behavior given in Figure 5.6. The automaton starts with a transition on an **a** event with no predecessors. Attached to this is a small square automata that recognizes **b** in parallel with **c**. To recognize parallel expressions, predecessor automata use a construction similar to the product of finite state machines[64]. The product automaton recognizes any interleaving of the languages accepted by its two constituent machines. The product automaton that recognizes **b** in parallel with **c** is represented by the square that has (**c a**) on the horizontal arcs and (**b a**) on the vertical arcs. The final part of the automata

Model:

a -> (b || c) -> d

Automata:

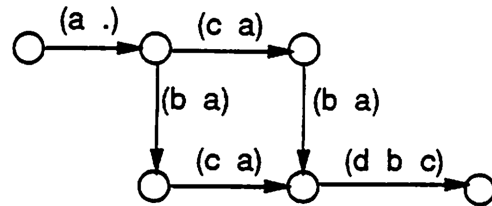


Figure 5.7 A model and an associated predecessor automata that implements the model.

recognizes the event **d** and is represented by a single transition labeled (**d b c**) at the rightmost end of the machine.

Although predecessor automata provide an implementation of the sequence and parallel constructs in a graph–based representation, they are inadequate for modeling highly parallel programs because there is no practical way to model a large number of parallel events. The implementation of the parallel operator uses a product of two machines which requires a number of states equal to the product of the number of states of the constituent machines. This leads to a state set size that is exponential in the length of the parallel expression to be recognized. The parallel closure operator is not currently implemented by predecessor automata.

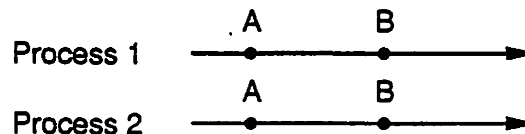
5.2 PEDL: Parallel Event Definition Language

We introduce a new modeling language based on perspective views and their associated temporal operators (**precedes** and **parallels**). It better models the behavior of parallel programs by providing a closer fit between the language operators and the types of abstract behavior users model. It can be efficiently implemented by a new kind of finite state automata, *parallel automata*, that maintains extra state information and defers decisions until execution time, allowing it to recognize parallel expressions without an exponential number of states in most cases.

PEDL does not attempt to address the parsing vs. pattern extraction issue; it uses a parsing interpretation with some small modifications.

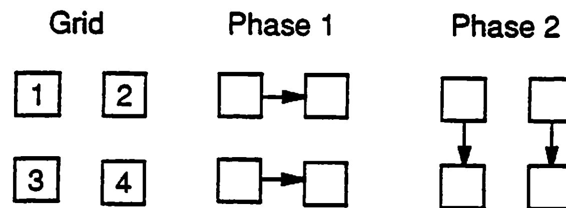
5.2.1 Language Operators

The language operators in PEDL are **precedes** ($\Rightarrow$), **precedes closure** (*), **parallels** ($\|$), and **parallels closure** ($\|^*$). Precedes and parallels are the operator versions of the temporal relations based on *happened before* introduced in Chapter 4. The difference between our operators and the DPE operators is that they are based on a different notion of "sequential" for abstract events. Our sequential operator, **precedes**, is less restrictive the DPE operator and models a larger class of abstract behaviors in highly parallel programs. We do not require a *happened before* relation between all pairs of events but instead require only that the abstract events be *consistently ordered*, that is, all *happened before* relationships between events must be from events in the first abstract event to events in the second. In the previous case,



we model the behavior as A **precedes** B. As another example, consider the system shown in Figure 5.8, in which processes in a grid first communicate horizontally ("East") in one phase and then vertically ("South") in the next. The user wishes to model this as an East phase (events labeled E) followed by a South phase (events labeled S). The relationship between the East and South events cannot be modeled with the DPE sequence operator because there is, for example, no ordering between the East event on Process 2 and the South event on Process 3. The relationship

Behavior on Process Array:



Behavior on Process-Time graph:

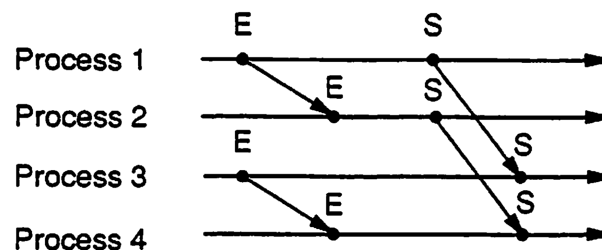


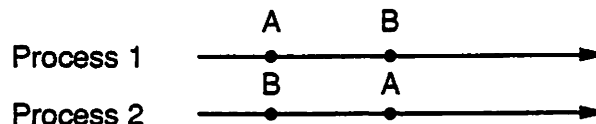
Figure 5.8 Communication phases on an application running on a small grid.

between the East events and the South events is easily modeled by our operator because there is a consistent ordering between them.

5.2.2 Modeling with Perspectives

Neither DPE's operators nor PEDL's operators can describe all possible relationships between abstract events. We deliberately chose not to add an operator to PEDL corresponding to the **overlaps** relationship because it is too general to be of use in modeling. Instead, we apply the idea of perspective views to allow a more precise description of the *happened before* relations that exist between overlapped abstract events. Several *perspectives* can be used simultaneously to describe

overlapped behavior from different viewpoints. In the following behavior,



abstract events A and B **overlap**. We can describe this behavior using two perspectives, one from each of the processes:

X *is* A,B
perspective
*.processId = 1 : A $\Rightarrow$ B
*.processId = 2 : B $\Rightarrow$ A
end

The model describes the behavior X as two events A and B where the ordering constraints between A and B are specified using perspectives. The events A and B must meet the constraints in both perspectives. Each perspective consists of two parts. The expression before the colon describes the set of events that compose the perspective. The PEDL expression after the colon describes the temporal ordering that must occur between candidate events. The example specifies that from the perspective of the events on Process 1, event A must **precede** B and from the perspective of Process 2, B must **precede** A.

Perspectives complete the PEDL language, allowing the specification of all possible relationships between abstract events. In principle, a user could specify the exact relationship between two abstract events as a long list of perspectives that specify the **precedes** (*happened before*) relationship between every pair of primitive events. It is unlikely that the user will do this; he or she is more likely to specify **precedes** relations between events that are logically related, such as the events from a single process shown in the example above.

We present additional examples showing the utility of modeling using perspectives in Section 5.4.

In the next section we show how a PEDL expression is translated into a recognizing automaton. A model that uses perspectives has an automata for each perspective; all automata associated with a model read the same input and all must accept that input for the model to match the behavior.

5.3 Implementing PEDL

We wish to implement the PEDL operators **precedes** ($\Rightarrow$), **precedes closure** (*), **parallels** ($\|$), and **parallels closure** ($\|^*$) with a reasonable degree of efficiency. Most existing automata are not suitable for implementing PEDL because they use a string-based representation of behavior that is not suitable for operators based on the *happened before* relation. Hseush and Kaiser's predecessor automata use *happened before* relations but are unsuitable for implementing PEDL for several reasons:

Exponential size. Predecessor automata use an exponential number of states to represent parallelism.

No parallel closure. Predecessor automata do not implement the parallel closure operator.

Not powerful enough. Predecessor automata make the decision to transition based only on the current state, the current input, and a predetermined set of predecessors. The more general notion of sequence in PEDL dictates that some of the predecessors of an event may not be known at compile time. For example, the expression $(a \parallel b) \Rightarrow c$, is true if a or b or both are predecessors of c, where the exact relationship between a, b, and c is not known at compile time. A related problem occurs with expressions involving the parallel

closure, which requires a variable number of predecessors. In the expression a $\|^* \Rightarrow$ c, any number of a events can match the expression a $\|^*$. Event c can have a data dependent number of predecessors. This expression can't be compiled into a predecessor automata because the number of predecessors can't be determined at compile time.

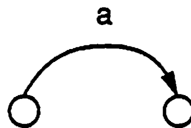
The following sections describe the our implementation of PEDL in terms of *parallel automata*. Section 5.3.1 introduces parallel automata and describes their run–time behavior. Section 5.3.2 describes the translation of PEDL expressions into parallel automata. Section 5.3.3 describes state changes during recognition. Section 5.3.4 addresses the complexity of parallel automata. Section 5.3.5 describes the compilation of nondeterministic parallel automata into deterministic parallel automata.

5.3.1 Parallel Automata

Parallel automata are constructed from standard finite state machines by changing the transition rules to include **precedes** relationships, and adding multiple threads of control and extra state information. Recognition happens in two phases. In the preprocessing phase, input events are read and a compacted form of the **precedes** graph is constructed using the process id and timestamp attributes on each event. Preprocessing is necessary because the **precedes** relation between two events cannot be computed until the entire input has been examined. In the recognition phase, the input is reread and each automaton consults the **precedes** graph in determining its transitions.

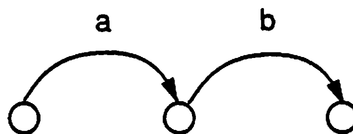
Like finite state machines, a transition is used to consume a single event. Unlike finite state machines, each machine maintains a *current context* that contains the immediate predecessors of the input event. Normally the context contains the event most recently consumed. (The current context after recognizing a **parallel**

expression is more complex; this case is discussed later in this section.) For a transition to apply, the current context must **precede** the input event; once the transition has occurred, the consumed event becomes the current context. For example, assuming the current context is empty, the following parallel automaton,



accepts a single a event. After taking the transition the current context will contain the event a.

An expression involving the **precedes** operator is implemented by the sequential composition of the automata recognizing its operands. For example, the following parallel automata recognizes the expression a $\Rightarrow$ b:



A transition on an a event places the a in the current context. The automaton will transition on a b event only when the a event in the current context **precedes** the b. This machine can distinguish between the behaviors in Figure 5.9, accepting the behavior shown in Figure 5.9(a) and rejecting the behavior shown in Figure 5.9(b).

To implement the **parallels** (||) operator we extend parallel automata so they can *split* and *merge*. At the start of a parallel expression the recognizer splits into child machines called *threads*. Each of the threads recognizes one part of the parallel expression. At the end of the parallel expression, the threads merge and a single machine continues. At the merge, the current context of the single machine is set to the union of the contexts of the child threads.

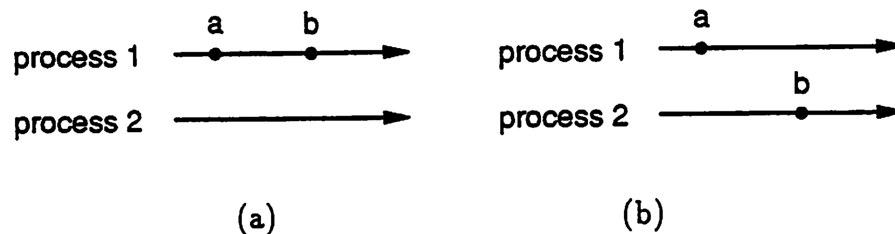
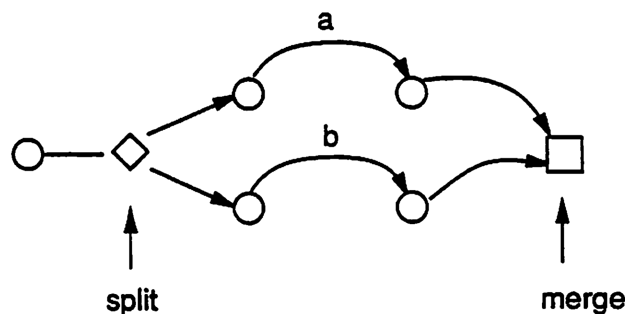


Figure 5.9 Sequential behavior (a) and parallel behavior (b).

Threads have their own context and finite state control but are not totally independent of each other: although all threads examine an input event, only one thread is allowed to consume it. Further, to insure that the semantics of the **parallels** operator is implemented, events consumed in one thread must be *unordered* with respect to events consumed by other threads. For example, the following machine,



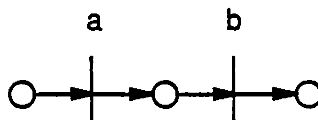
recognizes the expression a || b. The recognizer splits into two threads. One thread will only recognize an a event, the other only a b event. *These events can appear in the input in any order.* As each thread transitions it checks the activity of the other thread to insure that their respective events are **parallel**.[3] When both threads have transitioned they merge to complete the recognition. This

[3] This checking causes the time required to recognize n expressions in parallel with total length t to be at least $\Omega(nt)$. It relies on the computation of the precedes relation during preprocessing. The preprocessing phase requires time $O((E+M)P)$, where E is the number of events, M is the number of messages, and P is the number of processes.

construction will only accept a set of events that match the model a || b; thus it will recognize the behavior shown in Figure 5.9(b) but reject that of Figure 5.9(a).

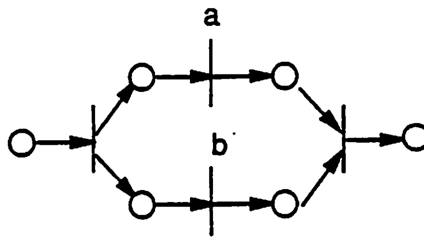
It is useful to contrast parallel automata with Petri nets[68]. Petri nets are a well studied model of concurrent systems. A Petri net consist of places, transitions, and tokens. Places are connected to transitions and transitions are connected to places by directed arcs. Tokens reside at places and move to other places via transitions. A transition is enabled when all places connected into the transition have a token. Any enabled transition can fire and when it does, tokens are removed from the input places and created at the output places. The state of the Petri net at any time is the set of places containing tokens.

Petri nets and parallel automata differ in how they interpret their input behavior. The standard interpretation of Petri nets as recognizers is to associate a symbol with each transition. If the next input symbol is associated with an enabled transition then the transition fires, the token is moved, and the symbol is consumed. This interpretation implicitly uses a string-based representation of ordering in the input behavior and does not insure that in a sequence of firings that each event **precedes** the next. Like other string based techniques, the following "sequential" Petri net recognizer,



would recognize both input behaviors in Figure 5.9 because both behaviors are represented as the same string, "ab". Similarly, the following "parallel" Petri net

recognizer



would also recognize both input behaviors in Figure 5.9 because it cannot distinguish between them.

5.3.2 Translation

We would like to produce recognizing automata from PEDL expressions consisting of event names and the temporal operators **precedes** ($\Rightarrow$), **precedes closure** (*), **parallels** ($\|$), and **parallels closure** ($\|^*$). We base our translation from language to automata on the standard compilation of regular languages to finite state automata[1], in which the expression is used to produce a nondeterministic automaton which is then converted into a deterministic automaton. We describe here how a nondeterministic parallel automata (NPA) is produced from a PEDL expression and the actions taken by a program interpreting a NPA as a recognizer. The potential for translating nondeterministic parallel automata into deterministic parallel automata is discussed in Section 5.3.5.

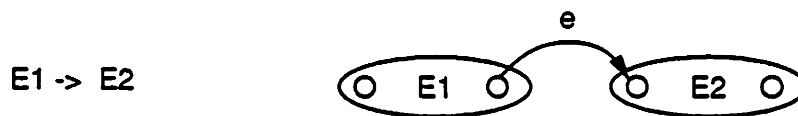
To translate a regular language expression into an NFA, we associate an automaton with each input symbol and operator in the language. During translation the expression is parsed bottom-up, and as each reduction is performed the automata associated with the reduction are combined. At the end of the parse, there is a single automaton that recognizes the entire expression. The translation of PEDL expressions is analogous.

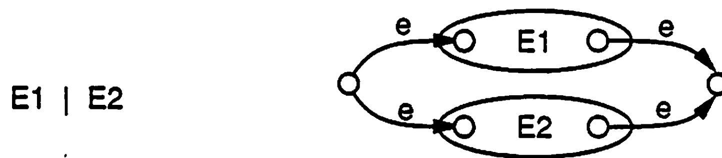
The machines associated with individual input symbols and PEDL operators are shown in Figure 5.10. Following the graphical notation used by Aho and Ullman[1] to describe finite state machines, a named oval containing two circles is used to represent a single entry/single exit automata that recognizes the named expression and an e is used to label an epsilon transition. The first four constructions are identical to those for regular expressions as presented by Aho and Ullman[1]. The last two constructions convert expressions using the **parallel** and **parallel closure** operators into automata. Both the **parallel** and **parallel closure** constructions contain a *split* transition, represented by a diamond (◇), and a *merge*, represented by a square (□). The diamond does not represent a state, but rather a place where a transition splits into multiple parts. The arc into a diamond is labeled with the symbol associated with the transition and the arcs leaving the diamond are left blank. The **parallel** construction contains a split transition which causes the recognizer to create two threads. One thread will recognize each of the **parallel** expressions. Following the recognition of each of its parts, the automata merges the two threads at a merge state. Only one thread continues past the merge state. The **parallel closure** construction contains a split transition in an a cycle of epsilon transitions. This specifies a nondeterministic, potentially infinite number of threads so that the automata can recognize any number of expressions in parallel. As with the **parallel** construction, the threads eventually merge at a merge state and only one thread continues past that state.

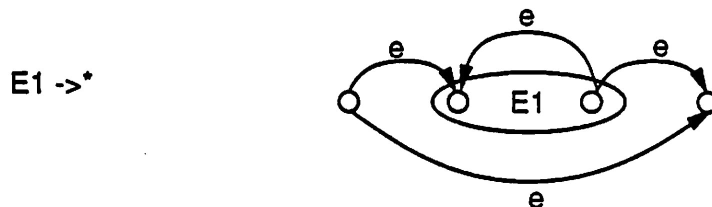
5.3.3 Interpretation of Nondeterministic Parallel Automata

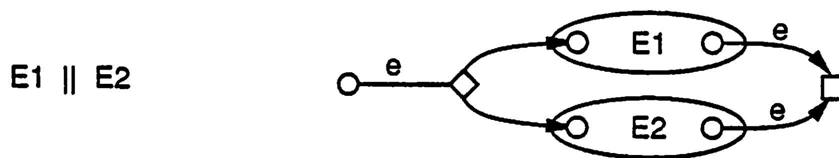
Behavior can be recognized by interpreting the actions of an automata on a stream of input. The recognizer maintains a current configuration. As each input is read, the recognizer changes configuration, aborts, or accepts as appropriate. To

For expressions "E1 and E2" where "e" represents epsilon :









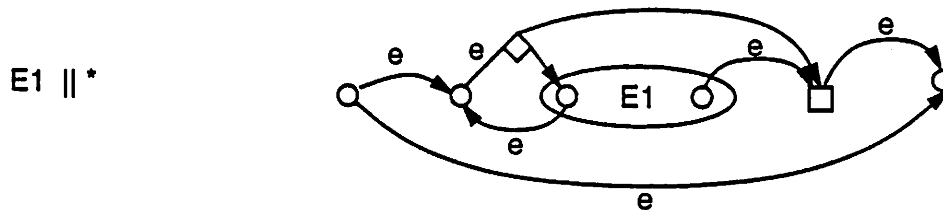
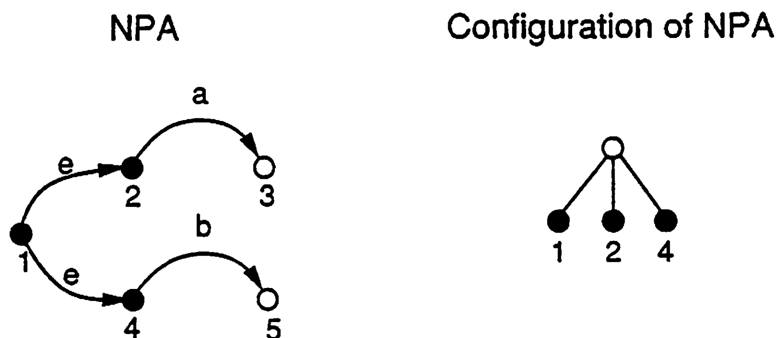


Figure 5.10 Converting PEDL expressions to Nondeterministic Parallel Automata.

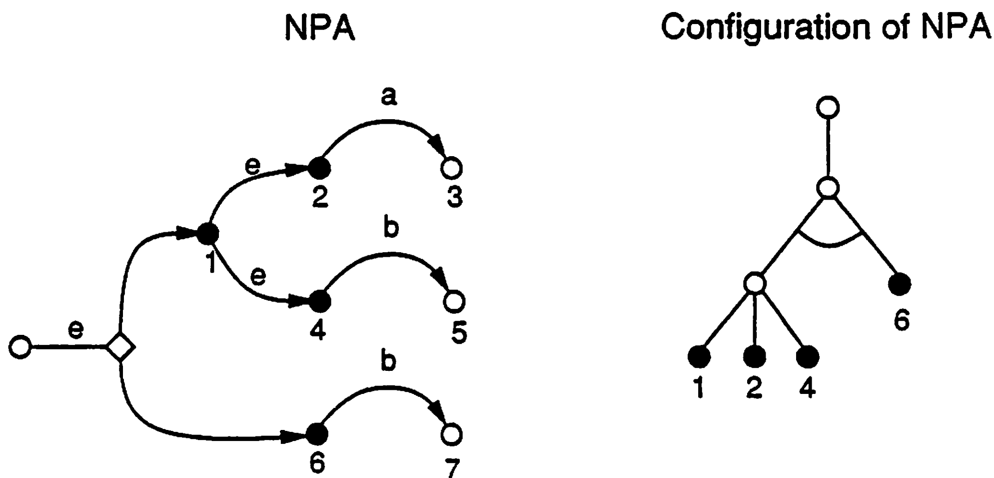
describe the interpretation of an NPA we again proceed by analogy with standard finite state machines.

An NFA consists of a number of states connected by directed, labeled arcs. The current configuration of the machine is represented by the set of states it could possibly be in. On each step of the interpretation an input symbol is read and all possible transitions from every state in the current configuration on that symbol are made. The new states, plus all states reachable from them by epsilon transitions, become the new configuration. If the new configuration contains an accepting state then the machine halts. If no transitions were possible then the machine halts with an error.

The interpretation of Nondeterministic Parallel Automata (NPA) is analogous but the existence of splits and merges requires a more complex representation. Instead of a set, the configuration of the machine is represented by an AND/OR tree; the root is an OR node and the remainder of the tree has alternating levels of AND and OR nodes. The leaves of the AND/OR tree represent the currently active states: OR nodes represent nondeterministic alternatives and AND nodes represent the states of parallel threads. For example, a simple NPA and the associated tree is shown below, using filled circles to mark states in the current configuration:

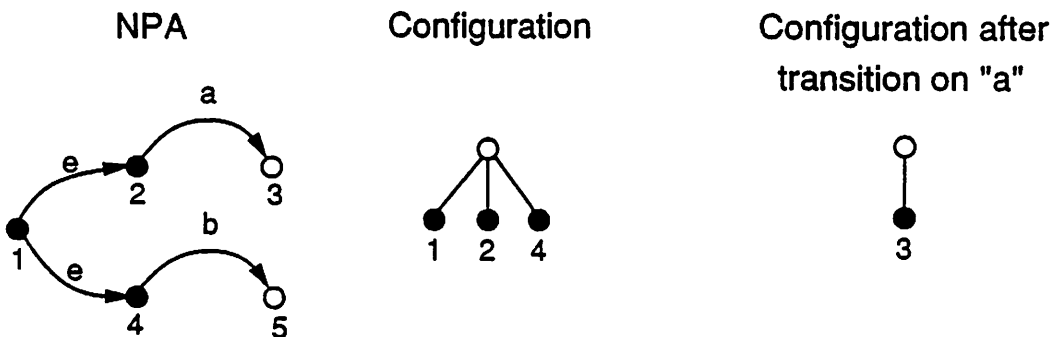


The root is an OR node, showing that the NPA could be in state 1, 2, or 4. For split transitions we need AND nodes, as in



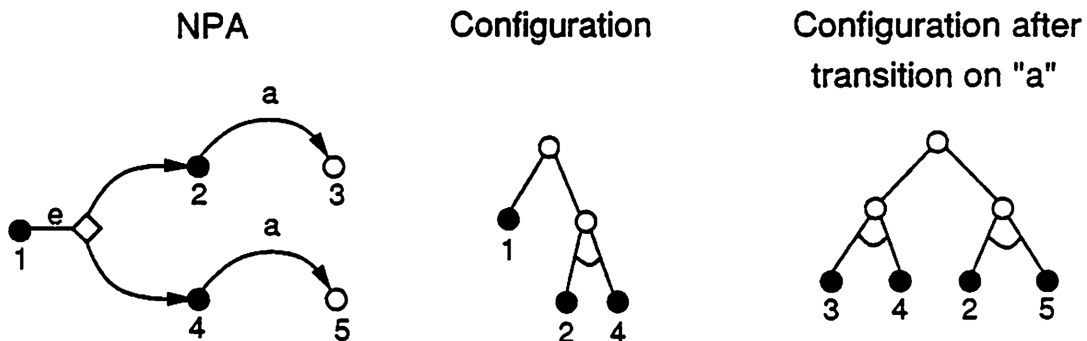
where the AND node is represented by a curve connecting its child nodes and it signifies that the NPA has one thread in state 6 and another in states 1, 2, or 4.

The action taken to update the current configuration tree on a transition depends on the number of transitions that occur on an input and whether the parent of the state where the transition originated is an AND or an OR node. Transitions from the child of an OR node are similar to transitions in an NFA: the new state replaces the original state in the representation and the other states are deleted. As an example, consider



where a transition from state 2 to state 3 occurs and states 1 and 4 are deleted. Transitions from the children of an AND node are more complex: if a transition

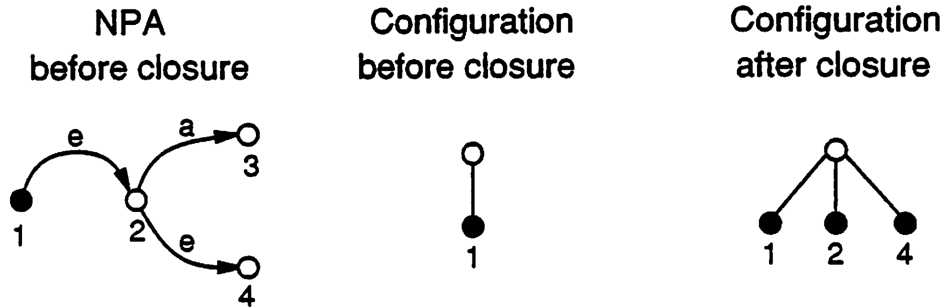
occurs from exactly one state (the normal case), one thread has consumed the input and the label on that state is changed; if no transitions occur, the parallel recognition has failed and the AND node and all of the child nodes are deleted; and if a transition from more than one of the nodes is possible, then a nondeterministic construction takes place representing the possibilities. For the nondeterministic construction, a copy of the subtree rooted at the AND node is made for each possible transition and the copies are joined together by a parent OR node. This construction is illustrated by



where transitions from state 2 to 3 and from state 4 to 5 are both possible. The resulting configuration shows an OR choice between the possibilities, with the transition from state 2 shown by the left subtree and the transition from state 4 shown by the right subtree.

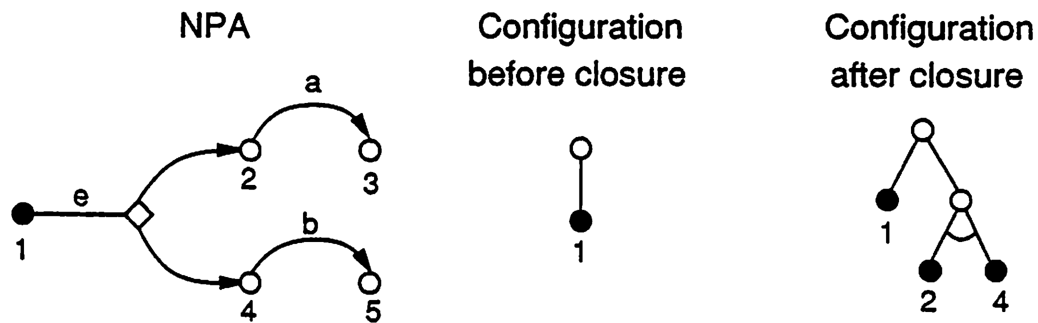
After every transition, new states in the configuration are *closed* by recursively following all epsilon transitions, adding visited states to the configuration as OR

node siblings. For example, the closure of state 1



includes both states 2 and 4 since they are reachable via epsilon transitions. Special actions occur on closures that encounter splits, epsilon-split-cycles, and merges.

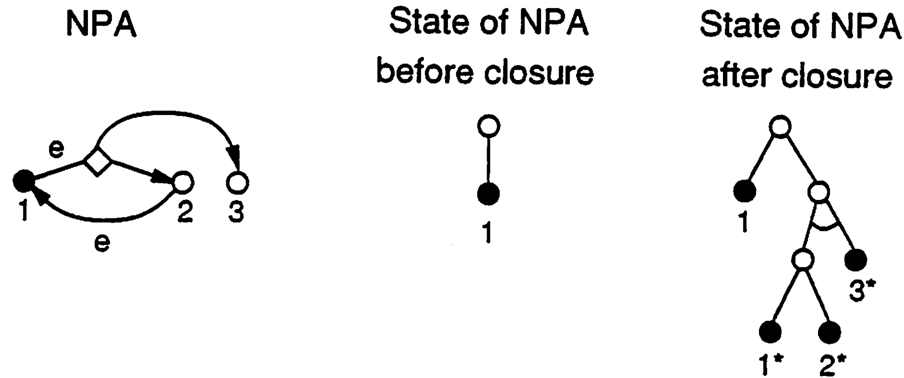
Split transitions are epsilon transitions that contain a split arc (◇). (They are created in the NPA construction for the **parallels** operator.) The action on a split is to generate an AND node with a child node for each thread created by the split; the action on a split generates two threads in the following example:



An epsilon-split-cycle is an epsilon-split within a cycle of epsilon transitions as occurs in the construction for the **parallels closure** operator. Epsilon-split-cycles pose a problem because our representation attempts to represent each thread explicitly and epsilon-split-cycles represent a potentially infinite number of threads. Our solution is to detect these cycles in the NPA and mark the affected states for special treatment when processing transitions.[4] The marking indicates a poten-

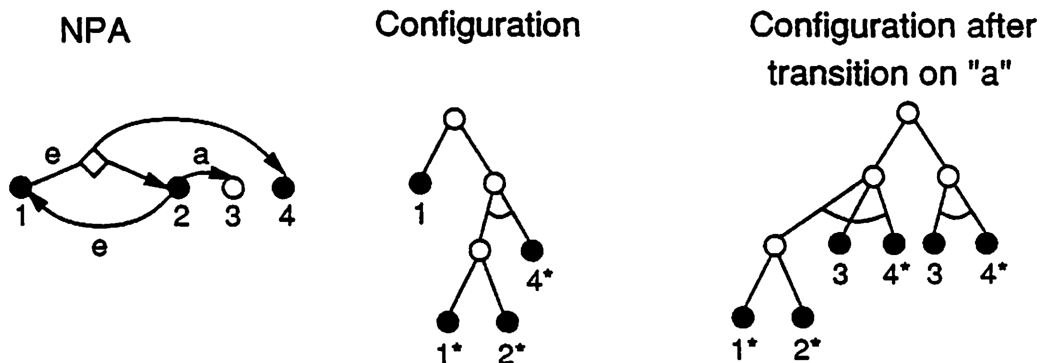
[4]Our solution is similar to that used to represent infinite markings in the reachability analysis of Petri nets[68].

tially infinite number of threads, as in the following example,



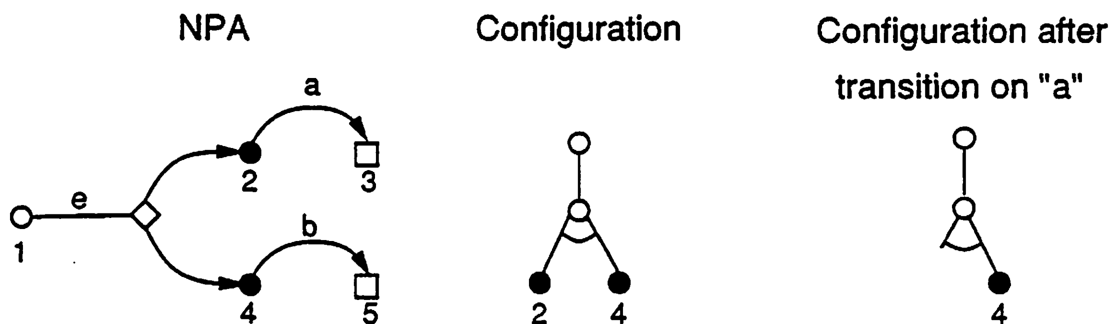
where the marked states starred.

A transition from a marked state represents a nondeterministic alternative: either one thread is in the marked state and it changes state or more than one thread is in the marked state and one thread changes state while the remaining threads do not. Both possibilities are explicitly represented, using two copies of the AND subtree containing the marked state that are connected with an OR node. In one copy, representing the case that only one thread is in the marked state, a normal transition from the marked state occurs. In the other copy, representing the case that more than one thread is in the marked state, a transition from the marked state occurs but an additional thread is created in the original marked state to represent the remaining threads. The construction is illustrated in the following example,



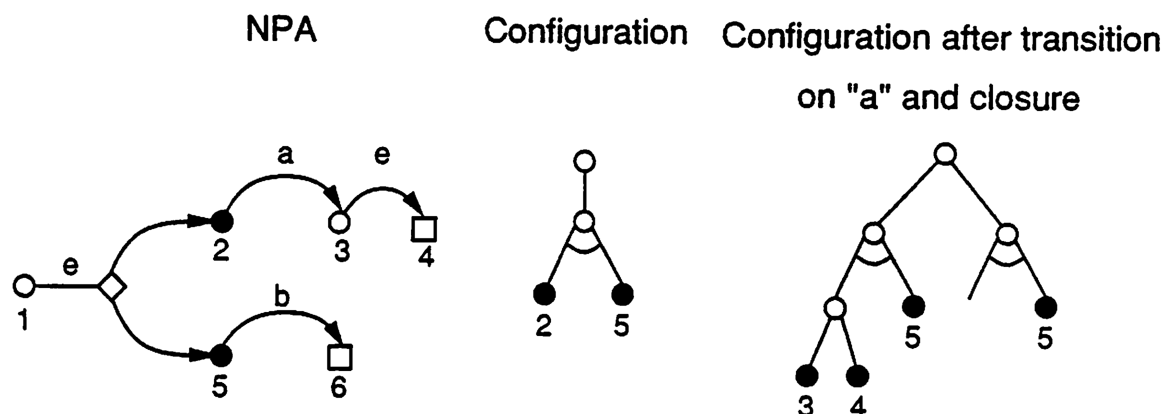
where a transition of one thread from marked state 2 is shown in the right subtree and a transition of one thread from state 2 with an additional thread remaining in state 2 is shown in the left subtree.

The action associated with a merge is to delete the merge state from the configuration because the thread that contains it is no longer active. Because splits and merges are always created in a nested fashion, a merge node will either be a child of the AND node that created it or a child of an OR node whose parent is the AND node that created the thread. If the merge node is the child of an AND node then a thread is terminating deterministically and the merge node is simply deleted. If the last child of the AND node is deleted, the child node replaces its parent, representing the single thread of control that continues past a merge state. The following example illustrates the deletion of a merge state from an AND parent:



The half arc is used to indicate an AND node with one child, to distinguish it from an OR node with one child. If the merge node's parent is an OR node then a thread is terminating nondeterministically and we must represent two possibilities: that the thread terminates or it that it remains active in a sibling state. To represent this situation the tree rooted at the AND node that created the thread (the grandparent of the current node) is duplicated. In one copy of the tree the thread remains. In the other copy the thread is deleted by removing the merge state and its OR siblings. The following example illustrates the deletion of a merge

state from an OR parent:



Finally, the interpreter halts when any of the states in the current configuration is an accept state.

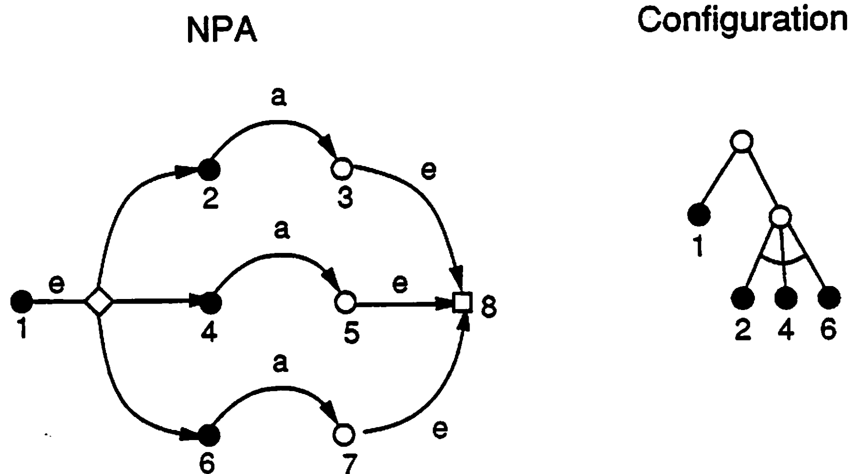
5.3.4 Complexity of Nondeterministic Parallel Automata

In the worst case, an NPA requires a number of states that is exponential in the length of the expression. However, the worst case is unusual and an exponential number of states can be avoided in most cases by a compiler optimization.

An exponential number of states can be required when interpreting some expressions involving the **parallel operator**. Consider the following model:

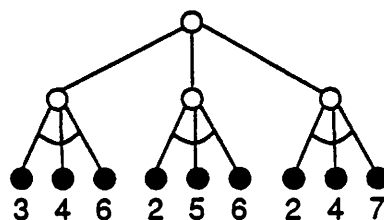
ThreeInParallel is $a_1 \| a_2 \| a_3$
end

At the start of interpretation, the NPA and its starting configuration would be:



From this configuration, an input event a could be consumed by any of the three threads. A copy of the old configuration, appropriately modified, is made to represent each possibility. The configuration after one transition on a would be:

Configuration after transition
on "a"



On a second a input, two transitions can now occur in each of the three subtrees, and copies are needed to represent each possibility. Thus, the number of states required for the entire expression is the factorial of the number of parallel subexpressions.

Only models of the form a || a || a ... or a ||* can potentially induce an exponential number of states because *every* input event can be consumed by *each* untransitioned thread. The state construction represents each possibility explicitly. We consider two cases. In the first and most common case, the constraints on

the events in the **parallel** expression are identical. We can optimize the number of states by noting that it does not matter which thread accepts an event. Rather than represent all possible thread transitions, the automaton can arbitrarily choose one thread to accept the event with no loss of generality. The optimized transition is represented by changing the state associated with the chosen thread without duplicating any state information. In the second case, the events in the **parallel** expression have different constraints associated with them. If the constraints are different then the NPA must explicitly represent each possible assignment of event to thread and use the corresponding exponential number of states. For example, the model

$UniqueParallelEvents\ is\ (a_1 \parallel a_2) \Rightarrow b$

  $filter$

    $b.color = a_2.color$

end

can not be optimized because the consumption of an a event by a particular thread affects the recognition of the following b event. We believe this case is rare because we have not seen it in practice and find it difficult to imagine when this construction would be needed.

Note that this optimization could not be applied to predecessor automata. Predecessor automata require a number of states that is exponential in the number of parallel expressions. Parallel automata only use an exponential number of states when the parallel expressions are identical, and the optimization reduces that in most cases.

5.3.5 Compilation to Deterministic Automata

Interpreting nondeterministic finite state automata can be slow. Every active state in the current configuration must be checked on every input symbol for a possible

transition. To improve recognition time a nondeterministic finite state automata can be translated into a deterministic finite state automata in which the current configuration of the automata is represented by a single state and there is a unique action from each state on each input symbol. Interpreting a deterministic finite state automaton requires only a unit time operation for every input symbol. The tradeoff for this decrease in execution time is an increase in space; the number of states required in the deterministic automaton can be exponentially larger than the number of states in the nondeterministic automaton.

Nondeterministic parallel automata can be translated into deterministic parallel automata, but the advantage of doing so is smaller than it is for compiling nondeterministic finite state machines. The compilation of a nondeterministic finite state automaton maps the current configuration of the machine, consisting of many states, to a single state in the deterministic machine. The compilation of a nondeterministic parallel automaton can not map the current configuration to a single state. Nodes in the current configuration that are children of an OR nodes, representing nondeterministic alternatives, can be collapsed onto a single state, but nodes that are the children of an AND node can not be collapsed because they represent the actions of independent threads. The deterministic NPA must maintain them as independent threads so that it can make sure that events that cause transitions in different threads are **parallel**. Children of AND nodes can not be collapsed onto the children of other AND nodes because the context of threads must be maintained independently. Thus, the speedup obtained by compilation is proportional to the fraction of collapsible nodes (OR Nodes) in the configuration. If half of the nodes in an average configuration are children of OR nodes, then on average we could only expect a deterministic parallel automata to need to check half as many states on each input event, and we could expect the

deterministic machine to be twice as fast as its nondeterministic counterpart. In the implementation under construction we do not intend to compile the automata.

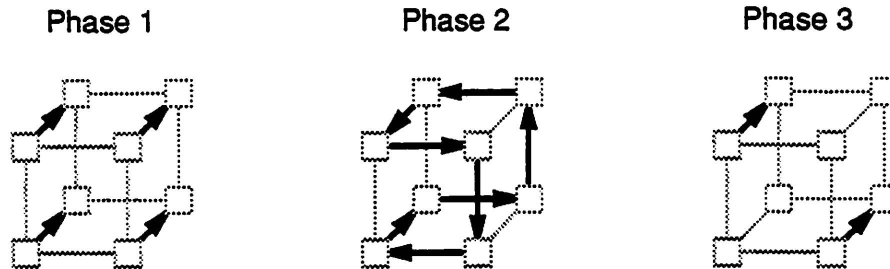
5.4 Examples

In this section we compare models written in PEDL with models written in Bates and Wileden's EDL language and Hseush and Kaiser's DPE language. EDL is a pattern extraction language and PEDL and DPE are both parsing languages. To compare them more equally we assume that the EDL models are using a "sharing" constraint so that each event is consumed by at most one model. Small syntactic differences between the modeling languages have also been removed in our examples.

5.4.1 Traveling Salesman Problem

This program provides an approximate solution to the traveling salesman problem by simulated annealing on a hypercube. Initially each city and its coordinates are assigned to a node. The current tour is defined by a fixed ring embedded in the cube. The algorithm operates by having pairs of cities along the salesman's tour compare and swap positions if doing so would shorten the overall tour length. The algorithm operates in three phases, shown in Figure 5.4.1, that are iterated under the control of the simulated annealing paradigm. In Phase 1, nodes on opposite ends of a given dimension of the hypercube compare positions. Nodes independently decide if swapping cities would improve the tour. Because of conflicts not all swaps are allowed to proceed; these conflicts are resolved in a synchronization phase with the hypercube configured as a ring shown in Phase 2. In the third phase the pairs of nonconflicting cities are exchanged.

Each phase of the behavior has a separate model. Figure 5.11 shows an EDL model of the first phase behavior which groups four consecutive "exchange" mes-



$Phase\ 1\ is\ Message_1 \triangle Message_2 \triangle Message_3 \triangle Message_4$

filter

$Message_i.name =$ *"exchange"*

$Bit(Message_1.sender, 1) = 0$

$Bit(Message_2.sender, 1) = 0$

$Bit(Message_3.sender, 1) = 0$

$Bit(Message_4.sender, 1) = 0$

$Bit(Message_1.receiver, 1) = 1$

$Bit(Message_2.receiver, 1) = 1$

$Bit(Message_3.receiver, 1) = 1$

$Bit(Message_4.receiver, 1) = 1$

end

Figure 5.11 EDL model of the first phase behavior of the Traveling Salesman program.

sages from the input. The model has constraints restricting the sender and receiver of each message to the proper face of the cube. This model accepts the first four "exchange" messages it comes across and has poor specificity: it does not insure that the matched messages are from the same iteration. There is a robustness problem as well. If a program bug causes extra or deleted "exchange" messages, the model would still group messages, four at a time, from whatever iterations they were in, until the end of input. A solution to this problem in EDL is to specify extra constraints on the attributes of the messages. In this example we could use a complex set of constraints on the process identifier of each message to insure that exactly one message gets sent from each process on each face, as shown in Figure 5.12. This technique was also used in example 3.16. As long as the program is correct, these extra constraints are sufficient to insure that the messages are all from the same iteration. However, if the program bug causes extra or deleted messages, the model could still match messages from different iterations. A message that is missing from an iteration would be replaced by a later message with the same name on the same process. An extra event in an iteration would be grouped with events from the following iteration. In both cases, abstract events would contain messages from two or more iterations.

The Phase 1 behavior can be modeled using PEDL as shown in Figure 5.13, where the model only matches four "exchange" messages that are unordered with respect to each other. This specification is simpler because it eliminates the need for extra constraints to limit messages to one message per process. The **parallel** operator limits the matched messages to one per process because messages sent from the same process are necessarily ordered; multiple messages from the same process are not **parallel**. The requirement that the messages be **parallel** also makes the model more robust. A missing message for example, will result in an abstract event with less than four messages and cause an error to be immediately

$Phase\ 1\ is\ Message_1 \triangle Message_2 \triangle Message_3 \triangle Message_4$

$filter$

$Message_i.name = \text{``}exchange\text{''}$

$Bit(Message_1.sender, 1) = 0$

$Bit(Message_2.sender, 1) = 0$

$Bit(Message_3.sender, 1) = 0$

$Bit(Message_4.sender, 1) = 0$

$Bit(Message_1.receiver, 1) = 1$

$Bit(Message_2.receiver, 1) = 1$

$Bit(Message_3.receiver, 1) = 1$

$Bit(Message_4.receiver, 1) = 1$

-- *make sure that only one message is sent per process*

$Message_2.sender! = Message_1.sender$

$Message_3.sender! = Message_1.sender$

$Message_3.sender! = Message_2.sender$

$Message_3.sender! = Message_1.sender$

$Message_4.sender! = Message_1.sender$

$Message_4.sender! = Message_2.sender$

$Message_4.sender! = Message_3.sender$

end

Figure 5.12 Better EDL model of the first phase behavior of the Traveling Salesman program.

$Phase\ 1\ is\ Message_1 \parallel Message_2 \parallel Message_3 \parallel Message_4$

filter

$Message_i.name = \text{“}exchange\text{”}$

$Bit(Message_1.sender, 1) = 0$

$Bit(Message_2.sender, 1) = 0$

$Bit(Message_3.sender, 1) = 0$

$Bit(Message_4.sender, 1) = 0$

$Bit(Message_1.receiver, 1) = 1$

$Bit(Message_2.receiver, 1) = 1$

$Bit(Message_3.receiver, 1) = 1$

$Bit(Message_4.receiver, 1) = 1$

end

Figure 5.13 PEDL model of the first phase behavior of the Traveling Salesman program.

reported. An extra message will result in an abstract event containing only the extra message and again cause an error to be reported.

An alternative PEDL model of the same behavior with different error handling properties is shown in Figure 5.14. This specification is simpler still because it eliminates the need to count messages and scales when the degree of parallelism in the program changes. However, program bugs are treated differently because the specification of the desired behavior is less constrained. If some processes do not send an "exchange" message it will be reflected by an abstract event with fewer than four messages. If some processes send extra "exchange" messages it will be manifested as an extra abstract event with less than four messages. These errors will not be flagged by the automata.

$Phase\ 1\ is\ Message_i\ \|^*$

$filter$

$Message_i.name = \text{"}exchange\text{"}$

$Bit(Message_1.sender, 1) = 0$

$Bit(Message_2.sender, 1) = 0$

$Bit(Message_3.sender, 1) = 0$

$Bit(Message_4.sender, 1) = 0$

$Bit(Message_1.receiver, 1) = 1$

$Bit(Message_2.receiver, 1) = 1$

$Bit(Message_3.receiver, 1) = 1$

$Bit(Message_4.receiver, 1) = 1$

end

Figure 5.14 Alternate PEDL model of the first phase behavior.

Phase 2, in which a sequence of messages is passed around the cube configured as a ring, is modeled in EDL as

$$Phase\ 2\ is\ Message_1 \circ Message_2 \circ Message_3 \circ Message_4 \circ Message_5 \circ Message_6 \circ Message_7 \circ Message_8$$

$filter$

$Message_i.name = \text{"}ring\text{"}$

end

which models Phase 2 as a sequence of eight consecutive "ring" messages ("$\circ$" is EDL's sequence operator). There is also a specificity problem with this model: messages from different iterations may be grouped together depending on the relative speed of the processes. The specificity of the model can be improved using

extra constraints as above. The PEDL model,

$$\begin{aligned} &Phase\ 2\ is\ Message_1 \Rightarrow Message_2 \Rightarrow Message_3 \Rightarrow Message_4 \Rightarrow \\ &\qquad Message_5 \Rightarrow Message_6 \Rightarrow Message_7 \Rightarrow Message_8 \end{aligned}$$

filter

$Message_i.name = \text{“ring”}$

end

matches a sequence of eight "ring" messages where each message must **precede** the next message. The model is more constrained than the EDL model but in some cases may still match messages from different iterations. The EDL model will group together messages from different iterations if eight consecutive messages in the input stream are not from the same iteration; this can happen if asynchrony causes processes to be executing different iterations of Phase 2. The PEDL model will group together messages from different iterations if eight consecutive message are not from the same iteration *and* the messages are related to each other by **precedes**; this can happen if the second to last process in the ring (process "n-2") is fast enough to send two messages to process "n-1" before the first message from process "n-1" is received at process "n". The PEDL model is also more robust in the presence of program errors. The EDL model will match the first eight "ring" messages regardless of what iteration they are from. In case of a deleted or extra "ring" message the PEDL model will generate an error if the "ring" message following the error is not related to the previous "ring" message in the input by **precedes**.

In Phase 3 of the algorithm, the proposed exchanges that do not conflict are swapped across a dimension of the hypercube in parallel. The Phase 3 model has a data dependent number of constituents and thus it cannot be modeled with EDL.

A model of the behavior using PEDL,

$$Phase\ 3\ is\ Message_i\ \|^*$$
$$filter$$
$$Message_i.name = \text{“swap”}$$
$$end$$

shows an indefinite number of **parallel** messages. This model is similar to the Phase 2 PEDL model that uses the parallel closure: it will match however many messages actually occur in **parallel**. Extra or deleted messages will result in abstract events containing more or fewer "swap" messages.

An EDL model of a single iteration of the traveling salesman program consisting of all three phases (assuming we could model Phase 3) would be:

$$Iteration\ is\ Phase\ 1 \circ Phase\ 2 \circ Phase\ 3$$
$$end$$

This model has severe robustness problems. A missing *Phase* behavior will be ignored and cause the other phases in the same iteration to be ignored because the recognizer will skip to the next behavior with the missing name. An extra *Phase* behavior will be silently ignored or cause *Iterations* to share *Phase* events, depending on the exactly how the sharing constraints are specified. Additionally, since this is a hierarchical model that includes the *Phase* models it will include the specificity and robustness problems of those models as well. The PEDL model of an iteration,

$$Iteration\ is\ Phase\ 1 \Rightarrow Phase\ 2 \Rightarrow Phase\ 3$$
$$end$$

will cause an error to be generated on either a missing or extra *Phase* behavior because the sequence of uniquely named events must exactly that specified in the model.

The DPE model of each of the three Phase behaviors is similar to the PEDL model and has the same properties. The DPE model of Phase 3, however, requires the parallel closure operator which is present in the language but is not implemented in predecessor automata. The Iteration behavior can not be described with DPEs because the phases are not related by the DPE sequence operator or any other DPE operators.

For each of these behaviors, the PEDL model has specificity and robustness properties similar to the DPE models and similar to or better than the EDL model. In addition, PEDL is able to model a behavior that EDL can not and a behavior that DPEs can not.

5.4.2 Median Filter

The median filter program presented in the previous chapter had two phases that were repeated iteratively: a first phase in which each process sends a message to its neighbors in a counter-clockwise fashion, followed by a second phase in which each process reads from its neighbors in the same counter-clockwise order. We would like to define an abstract behavior as all messages that were sent in the same direction in the same phase, for example, "all processes communicating with their north neighbor". This behavior is difficult to describe in DPEs and EDL because the relationship between the abstract events is complex. It cannot be modeled using DPEs because the ordering relationship between the messages does not correspond any of the DPE operators. The behavior can be modeled using EDL in a general fashion as a shuffle of all the messages in the same direction in the same iteration, but this model ignores the relationship between the events, resulting in specificity and robustness properties similar to those of the Phase 1 model above — the model can accidently group together behavior from different iterations because of variations in process speed or if there are extra or deleted

messages. Using the perspective capability of PEDL we can build a more specific, more robust model of this behavior because perspectives allow us to specify exactly the relationships between events that we are interested in. The following model of the "North" behavior,

$Norths$ *is* $Message_i$
  filter
    $Message_i.direction$ = *"north"*
  perspective
    $send : Message_i \; \|^*$
end

describes the *Norths* behavior as a set of messages whose *send* events are mutually parallel. Specificity and robustness are similar to the behavioral models of the traveling salesman program that used the **parallel closure**: there will be at most one message matched per process, and in the absence of program errors the messages will be all be from the same iteration. If there is an extra message in the iteration it will be reflected by an extra abstract event containing just the extra message; a deleted message will be reflected by an abstract event that contains one fewer message.

5.5 Summary and Conclusions

Modeling for debugging purposes is difficult because the modeling language must support abstractions of parallel program behavior, describe the expected behavior as precisely as possible, be robust in the presence of errors, and have an efficient, mechanically generated means of recognizing that behavior. Current modeling languages have shortcomings in some or all of these areas. In particular, existing

languages are not well suited to modeling the abstract, repetitive behavior of highly parallel programs.

We have presented a modeling language using temporal orders based on the *happened before* relation and introduced automata that recognize behavior specified using those operators. The *sequence*, *parallel* and *parallel closure* operators were shown to be an effective and more general mechanism for modeling behavior than existing alternatives. Perspective views, before only applied to animation, extend the descriptive power of the language so that it can precisely describe any relationship between abstract events. Existing languages can not describe all parallel behaviors.

PEDL is being implemented and we look forward to experimenting with its use. Much work remains to be done to provide an effective, specific, robust modeling language with good error handling capability. PEDL does not achieve this goal but we believe it does represent an improvement over existing languages.

CHAPTER 6

DEBUGGING

In this chapter we demonstrate the use of abstract visualizations in debugging programs. We are primarily interested in "parallel bugs" – those errors that occur because of (or affect) the interprocess communication patterns. These errors can be classified as (1) *sequencing errors* in which the expected communication patterns occurred but in the wrong sequence; (2) *missing communication errors* in which the expected patterns did not occur because some or all of their constituent events did not happen; or (3) *extraneous communication errors* in which the expected patterns occurred along with additional, unintended communications. In our examples, we show that abstract visualizations can be used in locating all three types of errors.

6.1 Case Studies

In each of the following case studies, we present an algorithm, an animation of its primitive behavior, a model of some or all of its abstract behavior, and an animation of that modeled abstract behavior, followed by a description of how we used the abstract animation to help locate a bug. These studies were all performed prior to the design of PEDL and thus used EDL as the modeling language. We do, however, describe PEDL models for each example and compare them to the EDL models.

6.1.1 Traveling Salesman

Our algorithm to perform simulated annealing of the traveling salesman problem on a hypercube was introduced in Section 5.4. Each annealing cycle of the algorithm has three steps: in the first, proposed swaps of values across a dimension of the cube are evaluated; in the second, a ring synchronization is performed to insure that accepted swaps do not conflict; and in the third, nonconflicting swaps are actually performed.

The animation of primitive events, shown in Figure 6.1(a), does not show the expected patterns of communication. As a result, we attempted to model each iteration step as a separate abstract event. Only the first two steps could be modeled with EDL because the third step involves an unpredictable number of parallel messages. To view the behavior of the program, we used abstract events for the first two steps and primitive events for the third step. A full perspective view showed each step at a separate time as in Figures 6.1(b-d). The clear image of the third step was produced fortuitously because the primitive events in the third step all *happen after* the abstract event representing the second step and are mutually **parallel** with each other; thus the abstract animation was the same as if the events of the third step had been grouped into an abstract event. Watching this animation, we detected a missing communication pattern in the third step of the iterations: swaps never occurred on some faces of the cube. This led us to discover a bug in the conflict resolution mechanism we were using, in which some proposed swaps were always accepted.

The PEDL model of this behavior could include the third step by using the **parallel closure** ($||^*$) operator. A perspective view using PEDL derived abstract step events would also display each step of the program at a separate time because the first step event **precedes** the second step event which **precedes** the third step event.

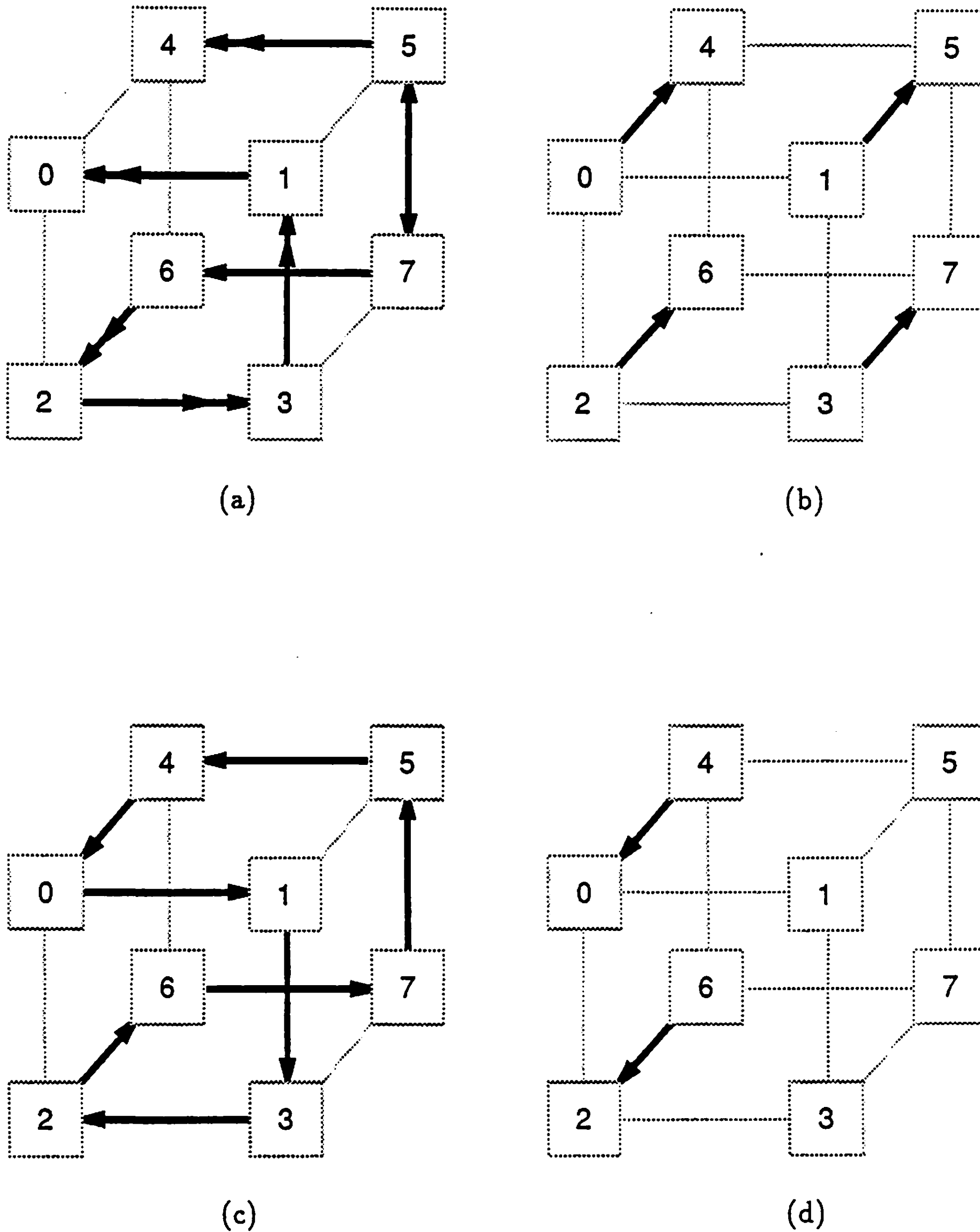


Figure 6.1 Traveling Salesman. Low-level communication events (a); abstract communication events in the evaluation step (b); abstract communication events in the ring synchronization step (c); and primitive communication events in the swap step (d). Doubled arrows indicate queued messages that have not been read by the receiving process.

6.1.2 Median Filter

In Chapter 3, we demonstrated that we could debug this algorithm using the database facilities and primitive event animations; here we find the same bug using abstract animations.

In the median filter algorithm, each process communicates with eight neighbors in a counter-clockwise rotation. On every iteration, a process first sends messages to all of its neighbors and then reads messages from all of its neighbors. The expected patterns of communication are hard to see in the primitive event animation in Figure 6.2(a), because of the large number of messages.

We defined a series of abstract events in the form "north messages from all processes", "west messages from all processes", "south messages from all processes", etc. The EDL model of the north behaviors was:

$$
\begin{aligned}
&Norths\ is\ Message_1 \triangle Message_2 \triangle Message_3 \triangle Message_4 \triangle \\
&\qquad Message_5 \triangle Message_6 \triangle Message_7 \triangle Message_8 \triangle \\
&\qquad Message_9 \triangle Message_{10} \triangle Message_{11} \triangle Message_{12} \\
&\ \ filter \\
&\qquad Message_i.direction = \text{"north"} \\
&end
\end{aligned}
$$

which collects together "north" messages in groups of twelve. Like the version of the median filter described in Section 4.5, the abstract events are related by **overlap** and a full perspective view could not improve their display. A partial perspective view from the perspective of the send operations was used. We expected a sequence of eight distinct abstract events in the counter-clockwise order that processes send messages, but we saw just the two complete events of Figure 6.2(b-c). These pictures suggested a bug causing the loss of send or receive event needed to form most of the eight abstract events. Suspecting that messages

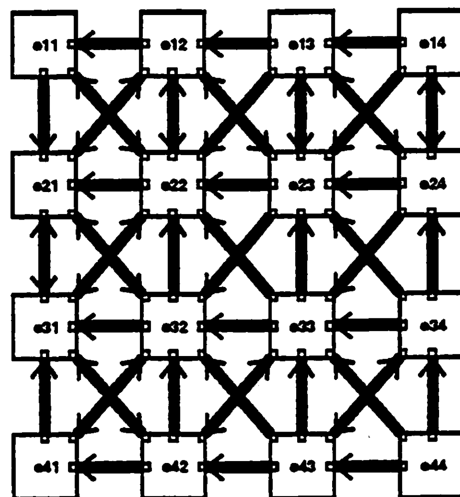
were being sent but not received, we decomposed the behavior using another partial perspective that included just the receive operations. This animation should have shown the abstract events in the counter-clockwise order that processes receive messages (different from the order that processes send messages). The actual animation, coincidentally identical to one illustrated in Figure 6.2(b-c), showed us that messages were being received properly, in sequence, until the "west messages" were not received. This led us to discover the bug, an incorrect initialization that prevented some east to west communication from occurring.

This behavior can be modeled with higher confidence using the PEDL perspective models presented in Chapter 5. The PEDL models using the **parallel** operator would have produced the same abstract events and result in the same visualization as the EDL models. However, it would provide us with more confidence that the abstract behavior was correctly matched, that is, that the matched behavior all occurred in the same iteration, because of the constraining effect of the **parallel** operator.

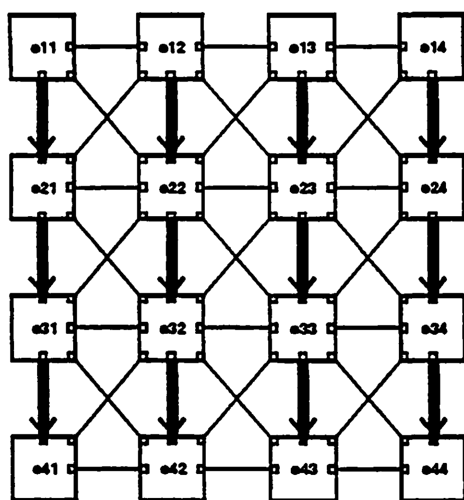
6.1.3 Dictionary Search

In this algorithm[65], a database of key-ordered records is stored in a hypercube.[1] Each node in the cube holds a contiguous section of the database. Queries entering from an external host are routed within the cube to the proper node using a binary search technique, the desired information is looked up on the node, and then routed back to the host. The host does not wait to receive the result of a query before sending out another. In the primitive event animation shown in Figure 6.3(a), multiple queries are active simultaneously and it is difficult to understand whether or not each query is proceeding as intended. For debugging purposes we wanted to view the progress of each query as it moved through the cube.

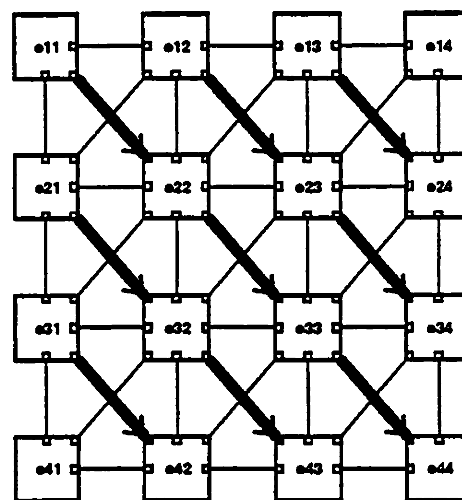
[1] Our implementation is a simplified version of the cited algorithm.



(a)



(b)



(c)

Figure 6.2 Median Filter algorithm. Low-level communication events (a); logically sequential abstract communication phases "south"(b) and "southeast"(c).

We defined an abstract event to be all of the communication generated on behalf of a query. Each message leaving the host was given a unique query identifier so that the host program can recognize the query when it returns. This identifier was carried by all messages belonging to the query, and was used to trivially group together the messages belonging to query by grouping all messages with the same identifier. This was accomplished using the following EDL model,

$$
\begin{array}{l}
Query\ is\ Message_i^* \\
\quad filter \\
\quad\quad Message_i.query_id = Message_1.query_id \\
end
\end{array}
$$

which specified that all messages grouped have the same identifier as the first message accepted by the model. A PEDL model would be almost identical and perform similarly, since the languages differ primarily in their temporal specification operators and the models group together messages using a value constraint on the query identifier.

Figure 6.3(b) shows a snapshot using a *duration* animation, in which all of processes and channels used by the primitive events in an abstract event are highlighted from the start of the earliest primitive event to the end of the latest primitive event. Highlighting processes and channels even when they are not being used helps the user see the abstract event as a single entity. Unfortunately, the snapshot shows the whole cube highlighted, indicating that all processes and channels are part of some query currently executing.

A full perspective view of the abstract events will not show us each query at a separate time because the abstract events overlap. Because queries visit different numbers of nodes, processes do not service queries in the same order. To separate the queries, we chose the set of send events on the host process as our perspective. Using a *traced duration* animation in which messages are left displayed until the

abstract event they belong has finished, the perspective view produced clearly separated queries as shown in Figure 6.3(c-d).

The query depicted in 6.3(c) was processed correctly, but that of 6.3(d) was not. A query packet should be sent across a dimension of the cube in each direction at most once; if it is returned, we know the target node is not in that half of the cube. In Figure 6.3(d), we can see a packet crossing between the front and back of the cube twice. Noticing the extra communication helped us uncover a bug — the packets returning to the host were initially being sent in the wrong direction from some processes. We present a more detailed version of this example in Appendix B.

Although we only needed a single perspective view to help us find this bug, we can use this example to illustrate how multiple perspectives on the same behavior might be useful. Suppose that instead of the routing bug discovered above, some process had a bug that caused it to discard queries after successfully servicing some. A perspective view, using the host send events as the perspective, would establish that several of the queries entered that process but never exited. It would reveal little about the actions of that process, however, because the queries would be shown in the order that they were issued from the host. A perspective view using the receive events on the suspect process would show the queries in the order that they were received and serviced, helping to establish the history and context of the process at the time of the error. Of particular interest would be the last query successfully serviced and the first query unsuccessfully serviced.

6.1.4 Dynamic Programming

Dynamic programming on a triangular array of processors [36] examines all possible solutions to a problem by combining pairs of partial solutions. Messages flow through the array from the processes on the main diagonal to the process at the

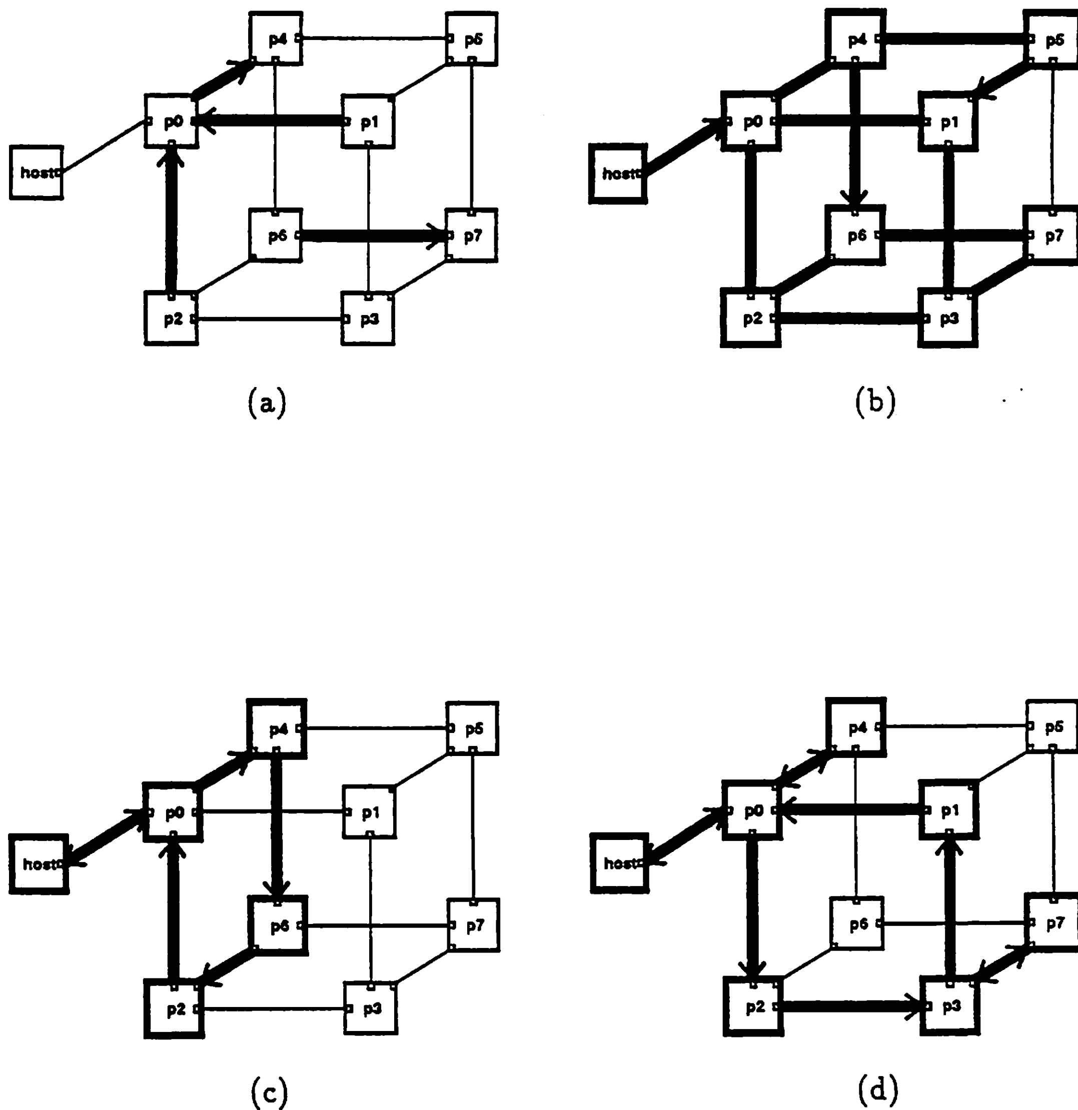


Figure 6.3 Snapshots of Belvedere's animation of the Dictionary Search. Low-level communication events (a); high-level events with several simultaneous search requests (b); a perspective view of high-level events showing the path taken by an individual request (c); erroneous communication between p3 and p7 (d).

upper right of the array. A snapshot from an animation illustrating this communication is shown in Figure 6.4(a). Each process combines pairs of incoming values to produce the partial solution to a larger subproblem; in our example, Process d15 computes the final solution using pairs of values from Processes d12 and d25, Processes d14 and d45, and Processes d13 and d35. Values that are to be paired at a process must arrive at that process together; to accomplish this, they are routed over paths of "fast" and "slow" channels. In our example, the value from Process d12 travels to Process d13 on the fast (upper) track and to Processes d14 and d15 on the slow (lower) track.

The correct pairing of values is crucial to the program but it is not discernible in the animation of primitive communication events shown in Figure 6.4(a). In order to debug our program we wished to model its abstract pairing behavior. Unfortunately, there was insufficient information available in the attributes of the messages to write a model of pairing behavior. Our workaround was to modify the program so that messages carried more information about their logical communication behavior. We added fields to each message identifying the original sender and eventual receiver of the message. This fields were carried by each message as extra values and were accessible to the modeling language. We then described the intended behavior with two families of models. The first family,

$$
\begin{array}{l}
\mathit{Logical_Channel}_{sender,receiver} \text{ is } \mathit{Message}_i^* \\
\quad \mathit{filter} \\
\qquad \mathit{Message}_i.\mathit{origin_id} = \mathit{sender} \\
\qquad \mathit{Message}_i.\mathit{destination_id} = \mathit{Message}_1.\mathit{destination_id} \\
\mathit{end}
\end{array}
$$

grouped communication along a single logical channel into an abstract event associated with the original sender and eventual receiver. The model is parameterized by a sender and receiver id which designate the ends of the logical channel. The

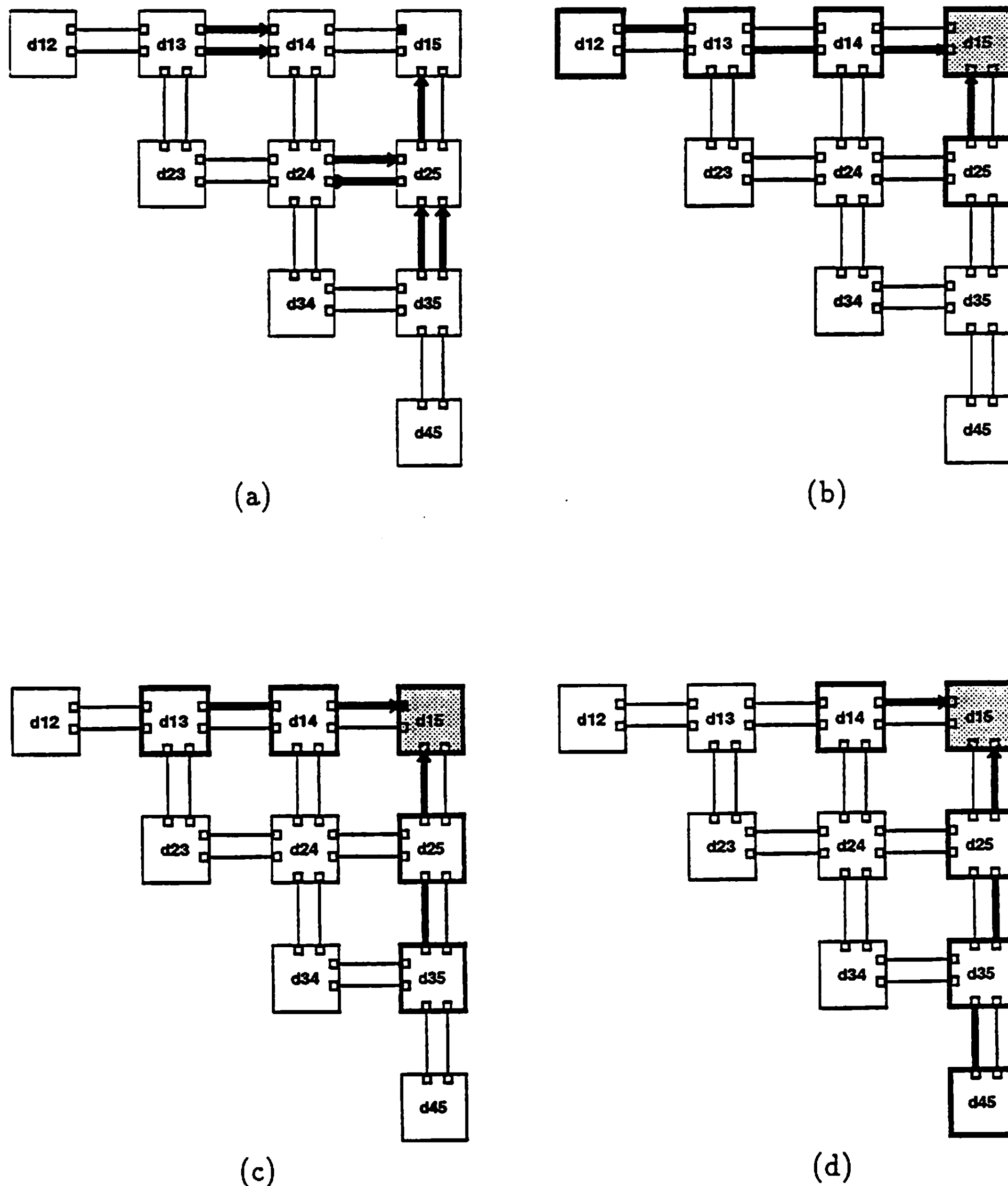


Figure 6.4 Dynamic Programming. Low-level communication events (a) and logical paths followed by paired values from Processes d12 and d25 (b), Processes d13 and d35 (c), and Processes d14 and d45 (d).

parameterized specification gives a separate model for each logical channel. The second model,

$Channel_Pair_{row,col}$ *is* $Logical_Channel_1 \circ Logical_Channel_2$

filter

$Logical_Channel_1.Message_1.destination_row = row$

$Logical_Channel_1.Message_1.destination_col = col$

$Logical_Channel_2.Message_1.destination_row = row$

$Logical_Channel_2.Message_1.destination_col = col$

$Logical_Channel_2.Message_1.origin_row =$

$Logical_Channel_1.Message_1.origin_col + 1$

end

grouped logical channels into pairs that are supposed to deliver their messages to a common destination at the same time. The model is parameterized by the row and column coordinates of the process that is the common destination of the logical channels.

Consecutive snapshots taken from the animation of *Channel_Pairs* delivering their values to Process d15 from the perspective of Process d15 are shown in Figures 6.4(b-d), where the expected logical path and pairing behavior of the algorithm is easily seen. Also evident is a sequencing error : the values from the pair d13 and d35 should have arrived before the values from either of the other two pairs, but they did not. This led us to a buffering problem in the internal processes that resulted in messages being forwarded out of order.

6.1.5 Fast Fourier Transform

For this program, we mapped the FFT's butterfly communication structure onto a rectangular mesh. On a 4 × 4 grid, the FFT algorithm takes four communication steps. Values are exchanged first between adjacent columns, then between columns that are two apart, then between adjacent rows, and finally between processes that are two rows apart. In the second and fourth exchanges communications are explicitly routed through intervening processes.[2]

In debugging the FFT, we modeled the logical patterns of swaps as described above. The second exchange, for example, was modeled as:

$$
\begin{array}{l}
Step_2_{row} \ is \ Message_1 \circ Message_2 \\
\quad filter \\
\qquad Message_i.row = row \\
\qquad Message_i.direction = Message_1.direction \\
\qquad Message_2.sender = Message_1.receiver \\
\qquad Message_2.original_sender = Message_1.original_sender \\
end
\end{array}
$$

We viewed the behavior from the perspective of Process `fft22` as shown in Figure 6.5 and found an error in the second exchanges: the images only showed communication between columns one and three but not between columns two and four. These missing communications led us to uncover a bug in the an east/west port specifications of processes in column three.

After that error was fixed, however, the program was still incorrect. Animation showed that all of the intended interactions were occurring, but subtracting the modeled behavior from the total behavior resulted in the extraneous communica-

[2]Our program and the models of its behavior are not scalable to different size grids. It is possible to write a more complex program and more complex models that are scalable.

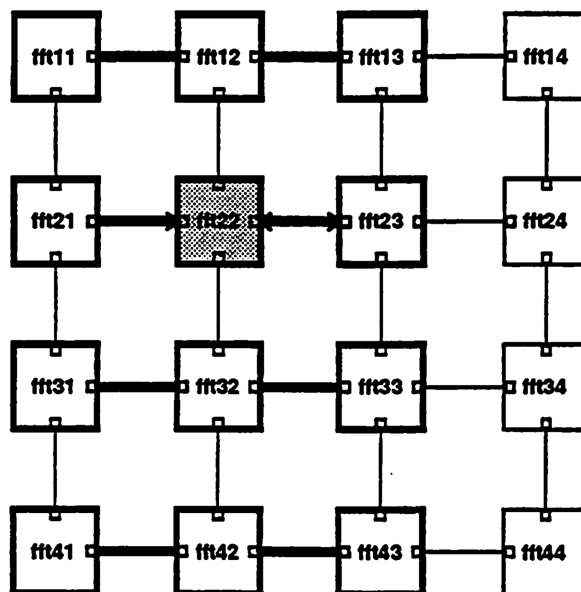


Figure 6.5 Communication between columns one and three in phase two of the Fast Fourier Transform.

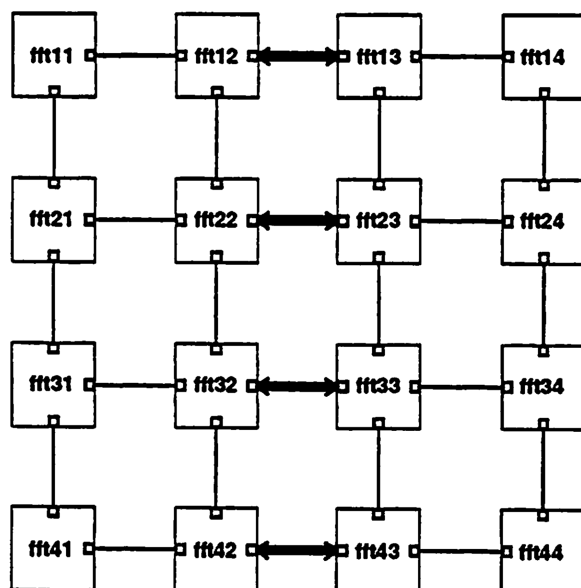


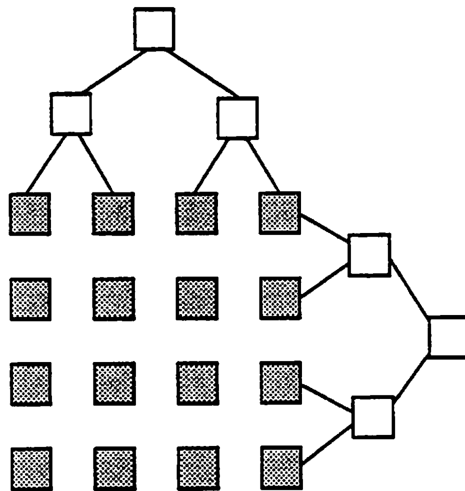
Figure 6.6 Extraneous communication during phase two of the Fast Fourier Transform.

tions shown in Figure 6.6. This led us to a redundant section of code in some of the processes.

A PEDL model of the second exchanges would have produced visualizations but would have provided better error checking. The PEDL model would have automatically detected the extra communication errors during recognition.

6.1.6 Connected Components

The algorithm given by Ullman to find the connected components of a graph[87] uses processes configured in a modified mesh-of-trees architecture. The original mesh-of-trees architecture consists of a rectangular grid of processes augmented with a binary tree over each row and a binary tree over each column. The following picture, for example, shows the construction with a tree over just the first row and the last column:



The complete graph would include 6 additional trees. For this algorithm, we modify the architecture so that the roots of the corresponding row and column trees coincide, as shown in Figure 6.7.

To find the connected components of the graph, the algorithm assigns each node to a component and then successively merges components that are connected. Each node of the graph is associated with a root process maintains the node's compo-

nent assignment. An adjacency matrix representation of the graph is stored in the mesh, one element per process. On each iteration of the algorithm, the component identifiers for each node are broadcast down the row tree attached to that root and temporarily stored in the corresponding row of the mesh. Each column then contains the current component ids of all nodes, and the smallest component connected to a node is found by reducing the ids of connected components up the column trees using a "minimize" combining function at each process in the tree. After one iteration, each root node contains the smallest component id of all components that are connected via a path of length one from the original component. The algorithm iterates until the maximum possible component size has been considered.

We were able to use the debugger in locating an error in this program even though the program produced correct communication patterns. Process perspective animations showed both `row broadcast` and `column aggregate` abstract behavior occurring as expected, shown in Figure 6.7. By viewing the abstract behavior from a variety of processes, we were able to determine that the overall communication flow was occurring as expected. Since the program produced incorrect answers even though the message patterns were correct, our next step was to consider the values passed during interprocess communications. Tracing data values in parallel algorithms is difficult. The asynchronous execution of processes means that several iterations of the algorithm may be represented on the screen at one time. It is difficult to interpret the values carried by the messages because they may belong to different iterations. A perspective view solves this problem by displaying all of the messages that belong to the same phase at one time. A full perspective with a traced animation that displays message values is shown in Figure 6.8.

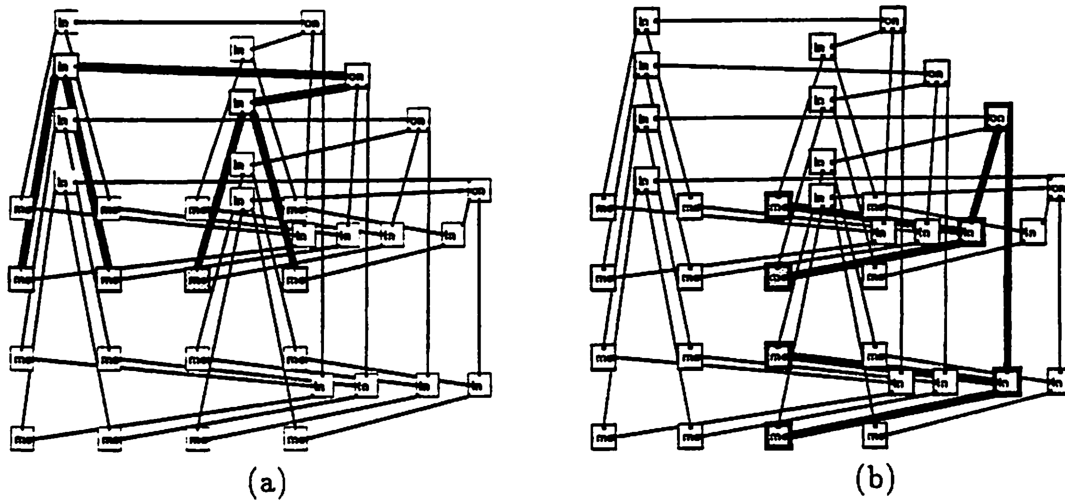


Figure 6.7 Connected Components. Animation of a `row broadcast` event and a `column aggregate` event from the process perspective of `mesh process (2,3)`.

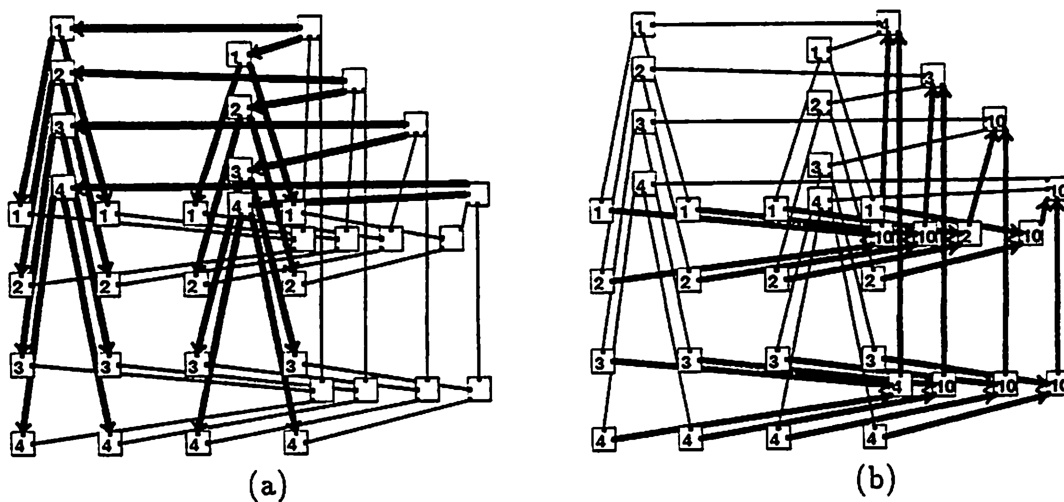


Figure 6.8 Connected Components. Animation of a common perspective showing a portion of a `row broadcast` event (a) and a `column reduction` event (b).

In Figure 6.8(a), the animation has been shown immediately after a `row broadcast`. The values shown are the values of the message last read. We can see from the figure that the initial component id's, (1,2,3, and 4) have been broadcast into the first four rows of the mesh. Figure 6.8(b) shows the minimum id's being propagated up to the roots ("10" is the truncated printing of a large value used to represent that the nodes are not connected). We were able to isolate the problem to one phase by observing that at the start of the phase the values were correct but they were incorrect by the start of the next phase. Locating the errant phase led us to a programming bug in which the wrong value was passed to a subroutine.

6.2 Conclusions

We have presented several examples showing that perspective visualizations can be used to debug communication-related errors in asynchronous, highly parallel programs. The examples were chosen to exhibit a variety of communication patterns and errors. In each example, asynchrony or complexity prevented the user from understanding the primitive behavior of the program. When viewed at a higher level of abstraction, the behavior could be understood and bugs were easily detectable as deviations from expected behavior.

In all of the examples PEDL could equal or better the ability of EDL to model the behavior. In one case (Traveling Salesman) PEDL could model a behavior that EDL could not. In most cases the power of the languages was similar but PEDL would have provided more confidence that the recognizer was matching the intended behavior.

The examples also show some limitations of the approach. First, we have only addressed bugs that affect patterns of interprocess communication. More abstract animations, such as the user-defined animations produced by Voyeur, and less abstract animations, such the process-time performance animations produced by

Moviola are useful aids to finding different kinds of bugs. A full visualizing debugger would need to incorporate animations at all of these levels. Such a debugger is envisioned by Leblanc in [53]. Second, the technique requires that primitive events can be grouped into abstract events according to the user's desires. In practice this means that primitive events must have attributes that enable them to be identified as members of a group by the recognizer. In our examples these attributes took the form of fields in messages that allowed messages to be identified from their contents. In most of our examples, messages had the attributes that we needed, but in some cases the availability of attributes was fortuitous because the programmer had tagged the messages as a matter of coding style but tagged messages were not necessary to the program (Dictionary Search, FFT), and in one case the attributes had to be added and the program rerun (Dynamic Programming). To use our techniques, programmers will often need to tag data so that it can be grouped and sometimes these tags will have to be added during debugging because the attributes needed may depend upon the abstractions desired.

CHAPTER 7

CONCLUSION

In this chapter we summarize the work, present our conclusions, and consider areas for future work.

7.1 Summary and Contributions

Asynchronous parallel programs are hard to debug. Asynchronous execution distorts the logical relationships between processes, forcing users to confront and understand low-level details. Because a highly parallel programs can consist of hundreds or thousands of independent processes, the amount of detail can be overwhelming.

To help users understand and debug parallel programs we have combined a modeling technique with a visualization technique. The modeling technique helps users group related program actions into a single abstract action. The visualization technique displays program behavior in terms of abstract actions, showing each abstract action as a distinct visual unit. In these abstract visualizations, behavior is easier to understand and bugs are easier to find.

Specifically, this dissertation makes the following contributions:

A Visualization Technique: We have described a general technique, *perspective views*, for improving arbitrary visualizations of parallel program behavior by reordering the visualization according to user-supplied abstractions. This technique can largely compensate for the distortion introduced by the asyn-

chronous execution of programs. It can also be used to decompose complex, hard-to-understand visualizations into a series of simpler visualizations.

A Modeling Technique: We have described a language, *PEDL*, that is better suited to creating abstractions of asynchronous highly parallel programs. By using language operators that more closely model the kinds of behavior found in parallel programs, this technique is able to find abstractions of program behavior using smaller, less error-prone descriptions.

We observe that debugging techniques for asynchronous parallel programs are facilitated by the use of a logical event ordering. Most existing debuggers use physical time to order events. Debugging program behavior in terms of physical time is difficult because there is a large number of independent clocks and asynchrony can change the relative timing of events on each execution. In contrast, a logical event ordering reflects the control and communication dependencies specified by the programmer, providing a uniform way of ordering events that is independent of relative process speeds. Because the ordering is based on dependencies specified by the user, modeling and displaying behavior in terms of the ordering are natural.

We also observe that abstraction works best in the later stages of debugging when the program is "nearly" correct. For example, with our techniques the user debugs by displaying the program behavior according to a graph representing the event ordering and looking for extra, missing, or changed elements of the graph. If the graph has been substantially modified by program errors then the abstract event recognizer will not be able to match the behavior and the debugger will not be able to improve the visualizations. Thus, abstraction based techniques are not appropriate to all stages of debugging.

We have previously noted other limitations of our techniques — the reliance on modeling technology, the availability of tagging information to the modeler, the

limitation of the screen size, *etc.* In the following section we suggest approaches to some of these problems.

7.2 Future Research

The techniques developed in this dissertation use events, temporal orderings, and visualization as a basis for debugging and visualizing parallel program behavior. Future directions for research will complete the work on Belvedere and PEDL, broaden the use of events captured from the program, consider other temporal orderings, and extend the range of the visualization techniques.

Following are some specific directions:

- *Implementing PEDL.* The implementation of PEDL in terms of Nondeterministic Parallel Automata is not yet complete. It should be completed and tested to evaluate PEDL as a modeling language and NPAs as an implementation of PEDL.

- *Visualization of very large programs.* Our techniques assume that it is possible to individually animate each process on the screen. This is not practical for programs containing millions of processes. We expect that very large programs will require the development of more sophisticated spatial abstractions of communication graphs.

- *Extend modeling to other parts of the program.* In order to focus on the "parallel" parts of a program we have based our examples exclusively on the events corresponding to the sending and receiving of messages. Extra events, such as those describing control flow and data flow events, would allow us to visualize other aspects of program behavior. These events would also lead to more precise models of communication behavior.

- *Use other partial orders as the basis of perspective views.* The *happened before* relation preserves the control flow ordering of events. This is not the only choice. Data flow is particularly intriguing because the *happened before* relation used in perspective views enforces dependencies that the user may not care about. For example, a program may send a data value to its north and east neighbors. These messages are sequential using *happened before* (because they are ordered by the sending process) but unordered from a data flow point of view. The user probably thinks of these message sends as "parallel" and might prefer to see them displayed simultaneously. Data flow could also be incorporated in a modeling language.

- *Automatic modeling.* Some kinds of models could be generated automatically from the program text. Models of blocks, conditionals, loops, and procedures are obvious abstractions of control flow that can be automatically generated from a compiler. If the language supports parallel flow constructs (such as a parallel for loop) or data flow constructs (such as canisters[7]) these could automatically generate models as well.

- *Language improvements.* Although PEDL has some ability to describe dynamic behaviors, it is primarily designed to model behaviors that can be predicted at compile time. As programmers write more complex parallel programs they will make more use of dynamic communication patterns and behaviors and need the ability to model them.

APPENDIX A

Complexity of the Perspective Algorithm

The time needed to compute a perspective view is $O(N \log N + E)$ where N is the total number of events in the system and E is the number of dependencies between the events.

To arrive at this result we analyze each of the four steps of the algorithm in turn:

1. *Construct a graph representation of events and dependencies.*

   The constructed graph has N nodes and E edges representing events and the dependencies between events, respectively. Dependencies have three sources: (1) the connection between an abstract event and its subevents; (2) the implicit *happened before* dependencies between primitive events; and (3) the constraints supplied from the display system.

   Dependencies from each source are processed to produce edges in the graph as follows:

   (a) *Connect each abstract event to its immediate child events.*

   The record describing each abstract event contains pointers to the records representing the immediate children of the event. For each pointer we create a bidirectional edge in the graph between the abstract event and the corresponding child event. For a system with N events and E_1 pointers to child events the complexity of this step is $O(N + E_1)$, because we visit each abstract event and each pointer once.

The total number of pointers is at most N^2 because there is at most one parent-child pointer between each pair of events.

(b) *Add a minimal set of* happened before *dependencies.*

In this step we add a set of edges sufficient to generate the *happened before* relation for the events in the perspective. Edges are only added between events in the perspective. First, an edge is created between each event and the next event on the same process. Second, for each message an edge is created between the event on the sending process immediately prior to the sending of the message and the event on the receiving process immediately following the receipt of the message. For a system with E_2 messages the complexity of this step is $O(N + E_2)$, because there is one edge added to the graph per node and one edge added per message. The number of messages is at most $N/2$ because each message has two events associated with it, the send event and the receive event.

(c) *Add visualization dependencies.*

Each visualization dependency causes an edge to be added to the graph. Given E_3 dependencies, the complexity of this step is $O(E_3)$ because each dependency is processed once. The number of visualization dependencies is at most N^2 because there is at most one dependency between each pair of events.

The complexity of step 1 is the sum of the complexities of the substeps, $O(N + E_1 + E_2 + E_3)$. This expression can be simplified by expressing it in terms of the number of edges in the graph. The total of $2E_1 + E_2 + E_3$ is the number of times step 1 attempts to add an edge to the graph. This total can be at most $3E$, where E is the number of edges in the graph, because each of

the three substeps adds a given edge to the graph at most once. Substituting $3E$ for $E_1+E_2+E_3$ gives a complexity of step 1 of $O(N+E)$.

2. *Find strongly connected components.*

   Tarjan's algorithm[83] finds the strongly connected components of a graph in time $O(N+E)$.

3. *Assign depths to components*

   The depth of each component is the length of the longest path to each component, counting along strongly connected components. To compute this length we first compact the graph so that each strongly connected component is represented by a single node; the compacted graph is acyclic. The longest path to each component can then be computed by the critical path method, normally used for scheduling jobs ordered by a graph of dependencies. The critical path method schedules a job to start when the jobs it depends on have completed; if we assign each job a unit execution length, then the time a job is scheduled to start is the length of the longest path of dependencies to that job. The critical path method schedules jobs in time $O(N+E)$[25].

4. *Reorder events.*

   Reordering the events has two substeps. Events are first sorted by depth and then a pass over the events in sorted order assigns new timestamps. Sorting can be accomplished in time $O(N \log N)$. The pass to assign new timestamps is $O(N)$.

No algorithm step requires more than $O(N \log N)$ or $O(N+E)$ time, so summing these complexities and simplifying gives the complexity of the entire algorithm, $O(N \log N + E)$. To compute the complexity in terms of N we observe that

the upper bound of E is the maximum number of unique edges in a directed graph, $2N^2$, because there can be at most two directed edges between each pair of nodes. Substituting this for E, the complexity of the algorithm in terms of N becomes $O(N^2)$.

APPENDIX B

DETAILS OF THE DICTIONARY SEARCH EXAMPLE

This appendix provides more details of the dictionary search debugging example presented in Chapter 6. Included here are additional snapshots from the animation, the text of the EDL model, and a listing of the process code used in the example.

In debugging our program, our first step was to check the overall message behavior using Belvedere. Snapshots from the primitive event animation are shown in Figure B.1. The animation showed communication from multiple queries, making it difficult to see if each query was proceeding as intended.

To see the communication associated with a single query, we used the model shown in Figure B.2; it differs from that given in Chapter 6 only in low-level details. The first line includes a definition of a "Message" abstract event similar to the definition given in Chapter 3. The model defines a "Query" as one or more "Messages" followed by the End of Stream token (EOS). The EOS token is used to tell the automata when to stop accepting Messages. The "with" clause declares that a Query has two local variables or attributes, queryId and temp. The "aftermatching" clause sets the local variable queryId to the query identifier carried by the first message accepted by the model. The "filter" clause extracts the query Id from the body of the second and subsequent messages and compares it to the queryId extracted from the first message. Only messages with the same queryId are accepted by the model.

The recognizer created abstract events named "Query" each of which contained all messages carrying the same query identifier. To view these abstract events, we

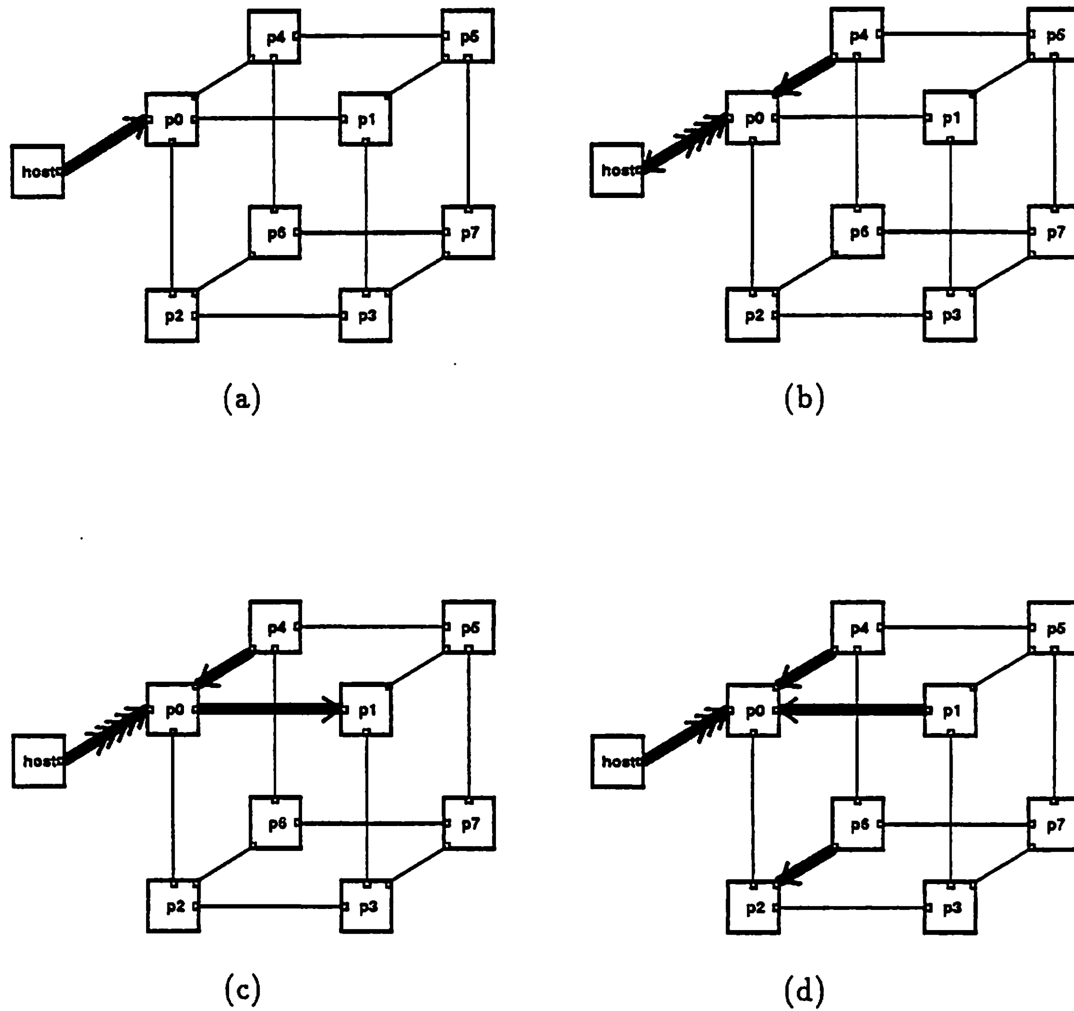


Figure B.1 Primitive event animation of the Dictionary Search.

```
#include "messages.h"
Query (nonsharedself) is Message[1] ' (Message[2]*) ' Eos
  with
      int queryId;
      int temp;
  end
  filter
      Message[2]: sscanf(Message[2].MessagePSA.MessagePut.mp_value,
                          "%d", &temp),
                  temp==queryId
      end
  end
  aftermatching
      Message[1]: sscanf(Message[1].MessagePSA.MessagePut.mp_value,
                          "%d", &queryId);
      end
  end
end
```

Figure B.2 EDL model of an abstract Query behavior in the Dictionary Search.

used a perspective view using the send events from the host. From this perspective the Queries are related by **precedes**, and the perspective view shows each in its entirety before showing the next. We used a traced animation to produce the snapshots shown in Figure B.3. They show the progress of one query which has now been separated from other queries.

The animation showed us a bug in Figure B.3(f) where the query crossed into the back plane of the cube for the second time. This led us to suspect the routing algorithm. The code running on each process, written using the Simple Simon Environment, is shown in Figures B.4, B.5, B.6, and B.7. The bug in Figure B.7 occurs in the part of the code that initiates the route of the query back to the host after it has successfully found a value on a node. The mistaken routing code always sends the query across the largest dimension of the cube whether it needs

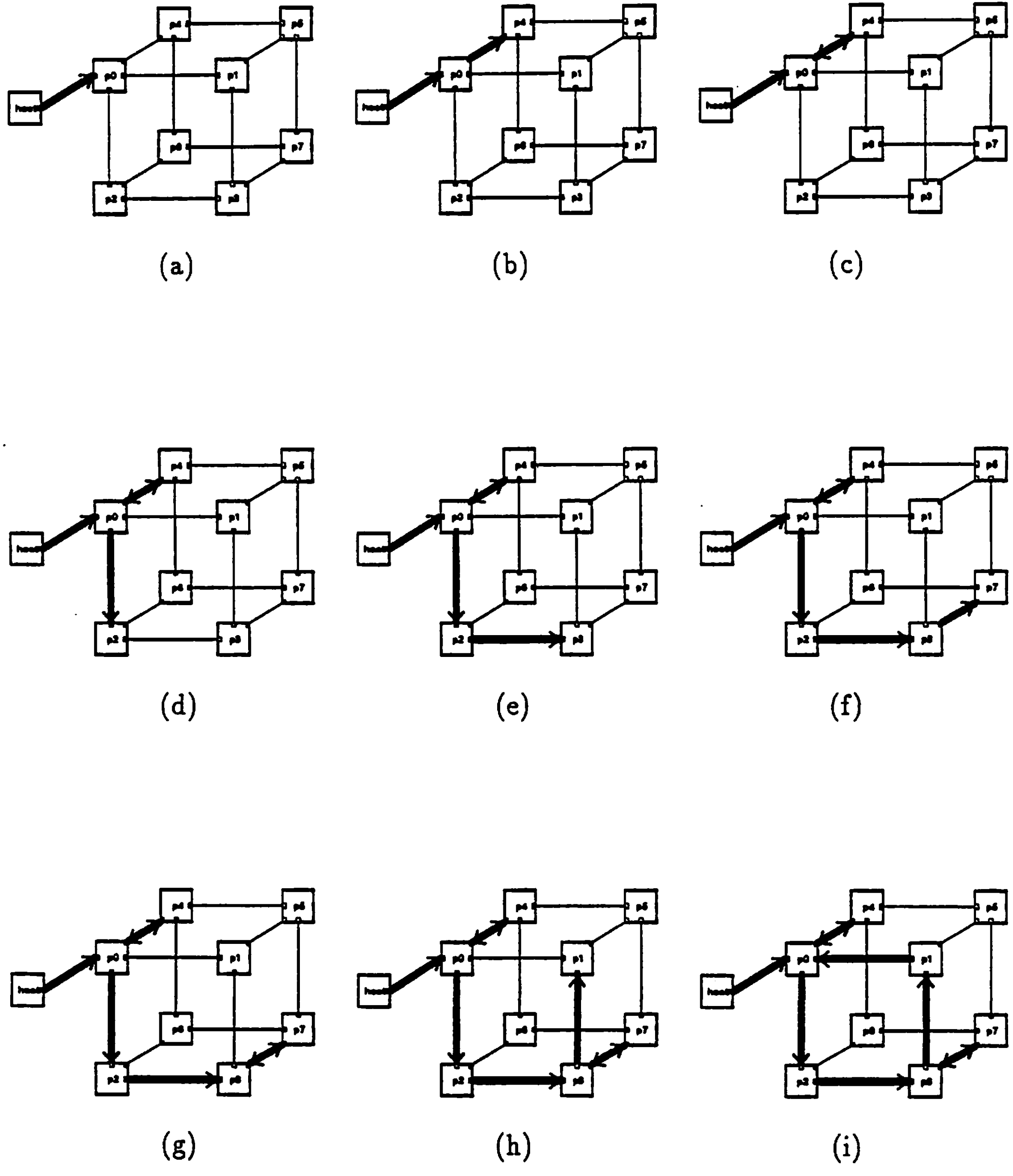


Figure B.3 Abstract event animation of a single query in the Dictionary Search.

to or not. The revised code, shown in Figure B.8, looks for the correct dimension before initially routing the return message.

```
#include "/user/simon/include/simon.h"
#include "localdefs.h"
#include "dict.h"
#include <math.h>

PID(arglist)
argstruct arglist;
{
    int argc;
    char **argv;
    int cubeId;
    /* min and max values of key on this process */
    int myLowKey = 1000000; /* infinity */
    int myHighKey = -1;
    int nKeys =0;
    dictEntry dictPart[MAX_DICT_PART_SIZE];
    int i;
    int outDim;
    boolean loadingEntry;
    char *Dim[DIMENSION];  /* port array */
    dictMessage msg;

/* start */
    /* pick up arguments */
    getargs(arglist,argc,argv);
    getiarg(argv,1,cubeId);

    for(i=0; PortList[i]; i++){
        copen(PortList[i],"rw");
    }

    Dim[0]=DIM0;
    Dim[1]=DIM1;
    Dim[2]=DIM2;
```

Figure B.4 Process code for the Dictionary Search, Part 1.

```
    /* copy our part of global dictionary to our part; this simulates
       the loading phase */
    for(loadingEntry=TRUE,i=0; i<MAX_DICT_PART_SIZE && loadingEntry;i++){
        dictPart[i] = GlobalDict[cubeId][i];
        loadingEntry = (dictPart[i].name != NULL);
        myLowKey  = MIN(myLowKey, dictPart[i].key);
        myHighKey = MAX(myHighKey, dictPart[i].key);
        nKeys++;
    }
    go();
    clock_on();

    /* each process serves find requests */
    while (TRUE){
        /* check for a message on each channel */
        for(i=0; PortList[i]; i++){
            if (qlength(PortList[i])){
                GET("*",PortList[i],param(msg));
                if (msg.msgType == RETURN){
#                   ifdef host
                        PUT("*",host,param(msg),"msg",msgPrintable(msg));
#                   else
                        /* have to route it somewhere */
                        while( msg.routingDim < DIMENSION &&
                               !(cubeId & (1<<msg.routingDim))) {
                            msg.routingDim--;
                        }
                        if ( (cubeId & (1<<msg.routingDim))){
                            outDim = msg.routingDim;
                            INDEXED_PUT("*","DIM",outDim,Dim[outDim],
                                        param(msg),
                                        "msg", msgPrintable(msg));
                        }else{
                            ERROR("dict","routing error");
                        }
#                   endif
```

Figure B.5 Process code for the Dictionary Search, Part 2

```
}else if (msg.msgType == REQUEST){
  if (msg.isLeftMessage){
      msg.isLeftMessage = FALSE;
      msg.lowProcess = cubeId+1;
      msg.routingDim--;
      outDim = msg.routingDim;
      INDEXED_PUT("*","DIM",outDim,Dim[outDim],
                  param(msg),"msg", msgPrintable(msg));
   }else if
     ((msg.key < myLowKey  && msg.lowProcess == cubeId)||
      (msg.key > myHighKey && msg.highProcess == cubeId)){
          printf("key not in list, process %d\n",cubeId);
          msg.msgType = RETURN;
          msg.routingDim = DIMENSION-1;
          strcpy(msg.name,"NOT FOUND");
          /* have to route it somewhere */
          while( msg.routingDim < DIMENSION &&
                !(cubeId & (1<<msg.routingDim))) {
              msg.routingDim--;
          }
          if ( (cubeId & (1<<msg.routingDim))){
              outDim = msg.routingDim;
              INDEXED_PUT("*","DIM",outDim,Dim[outDim],
                          param(msg),
                          "msg", msgPrintable(msg));
          }else{
              ERROR("dict","routing error at not found");
          }
   }else if (msg.key < myLowKey){
      msg.highProcess = cubeId-1;
      msg.isLeftMessage = TRUE;
      outDim = msg.routingDim;
      INDEXED_PUT("*","DIM",outDim,Dim[outDim],
                  param(msg),"msg", msgPrintable(msg));
   }else if (msg.key > myHighKey){
      msg.lowProcess = cubeId+1;
      msg.routingDim--;
      outDim = msg.routingDim;
      INDEXED_PUT("*","DIM",outDim,Dim[outDim],
                  param(msg),"msg", msgPrintable(msg));
```

Figure B.6 Process code for the Dictionary Search, Part 3.

```
                }else{ /* it's local */
                    for(i=0; i<nKeys; i++){
                        if (dictPart[i].key == msg.key){
                            /* got it */

                            /******** B U G G Y  C O D E *******/

                            msg.msgType = RETURN;
                            msg.routingDim = DIMENSION-1;
                            strcpy(msg.name,dictPart[i].name);
                            outDim = msg.routingDim;
                            INDEXED_PUT("*","DIM",outDim,Dim[outDim],
                                        param(msg),
                                        "msg", msgPrintable(msg));

                            break;
                        }
                    }

                    if (i==nKeys){ /* not found */
                        printf("key not in list, from process %d\n",
                               cubeId);
                        msg.msgType = RETURN;
                        msg.routingDim = DIMENSION-1;
                        strcpy(msg.name,"NOT FOUND");
                        outDim = msg.routingDim;
                        INDEXED_PUT("*","DIM",outDim,Dim[outDim],
                                    param(msg),
                                    "msg", msgPrintable(msg));

                    }
                }
            }
        }
    }
  }
}
```

Figure B.7 Process code for the Dictionary Search, Part 4.

```
}else{ /* it's local */
    for(i=0; i<nKeys; i++){
        if (dictPart[i].key == msg.key){
            /* got it */

             /******** REVISED  CODE*******/
             msg.msgType = RETURN;
             msg.routingDim = DIMENSION-1;
             strcpy(msg.name,dictPart[i].name);
             while( msg.routingDim < DIMENSION &&
                   !(cubeId & (1<<msg.routingDim))) {
                 msg.routingDim--;
             }
             if ( (cubeId & (1<<msg.routingDim))){
                 outDim = msg.routingDim;
                 INDEXED_PUT("*","DIM", outDim,Dim[outDim],
                       param(msg),"msg",msgPrintable(msg));
             }
             break;
        }
    }
    if (i==nKeys){ /* not found */
        printf("key not in list, process %d\n",cubeId);
        msg.msgType = RETURN;
        msg.routingDim = DIMENSION-1;
        strcpy(msg.name,"NOT FOUND");
        outDim = msg.routingDim;
        INDEXED_PUT("*","DIM",outDim,Dim[outDim],
                  param(msg),"msg", msgPrintable(msg));

    }
}
}
}
}
}
}
```

Figure B.8 Revised process code for the Dictionary Search.

BIBLIOGRAPHY

[1] Aho, A. V. and Ullman, J. D. *Principles of Compiler Design.* Addison-Wesley, 1977.

[2] Allen, T. R. and Padua, D. A. Debugging FORTRAN on a shared memory machine. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 721–727, 1987.

[3] Appelbe, W. F. and McDowell, C. E. Integrating tools for debugging and developing multitasking programs. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 78–88, May 1988. Proceedings also published as SIGPLAN Notices, 24(1), January 1989.

[4] Avrunin, G. S., Dillon, L. K., Wileden, J. C., and Riddle, W. E. Constrained expressions: Adding analysis capabilities to design methods for concurrent software systems. *IEEE Transactions on Software Engineering*, SE-12(2):278–292, February 1986.

[5] Baeker, R. and Sherman, D. Sorting out sorting. 16mm color sound film, 1981. Excerpted in ACM SIGGRAPH Video Review #7, 1983.

[6] Baiardi, F., De Francesco, N., and Vaglini, G. Development of a debugger for a concurrent language. *IEEE Transactions on Software Engineering*, SE-12(4):547–553, April 1986.

[7] Bailey, D. A. *Specifying Communication for Massively Parallel Ensemble Machines.* PhD thesis, University of Massachusetts, Amherst, MA 01003, 1988. Also COINS Technical Report 88–83.

[8] Bailey, M. W. Simple simon database documentation. Documentation produced in the Parallel Programming Environments Group, Computer and Information Sciences Dept, University of Massachusetts, Amherst, MA, 01003, under the direction of Professor Janice E. Cuny., 1987.

[9] Balzer, R. M. EXDAMS – EXtendable Debugging and Monitoring System. In *Proceedings of AFIPS Spring Joint Computer Conference*, pages 567– 580, 1969.

[10] Bates, P. C. *Debugging Programs in a Distributed System Environment.* PhD thesis, University of Massachusetts, Amherst, MA 01003, 1986. Also COINS Technical Report 86-05.

[11] Bates, P. C. Debugging heterogeneous distributed systems using event-based models of behavior. *SIGPLAN Notices*, 24(1):11–22, January 1989.

[12] Bates, P. C. and Wileden, J. C. Event definition language: An aid to monitoring and debugging complex software systems. In *Proceedings of the Fifteenth Hawaii International Conference on System Sciences*, pages 86–93, January 1982.

[13] Brindle, A., Taylor, R., and Martin, D. A debugger for Ada tasking. Technical Report ATR-85(8033)-1, Aerospace Corporation, 1985.

[14] Brown, M. H. *Algorithm Animation.* PhD thesis, Brown University, 1987. Tech report No. CS-87-05, also available from MIT press.

[15] Brown, M. H. Exploring algorithms with Balsa-II. *Computer*, 21(5):14–36, May 1988.

[16] Brown, M. and Sedgewick, R. A system for algorithm animation. *Computer Graphics*, 18(3):177–186, July 1984.

[17] Bruegge, B. and Hibbard, P. Generalized path expressions: A high level debugging mechanism. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, pages 34–44, 1983. Also SIGPLAN Notices, 18(8), August 1983, and Software Engineering Notes, 8(4), August 1983.

[18] Chandy, K. and Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[19] Clarke, L. A. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, September 1976.

[20] Couch, A. *Graphical Representations of Program Performance on Hypercube Message-Passing Multiprocessors.* PhD thesis, Tufts University, Medford, Massachusetts, 1988. Appears as technical report 88–4, Department of Computer Science, April 1988.

[21] Couch, A. L. *Seecube User's Manual.* Department of Computer Science, Tufts University, 1988.

[22] Cuny, J. E., Bailey, D. A., Hagerman, J. W., and Hough, A. A. The Simple Simon programming environment: A preliminary report. In *Proceedings of the Twenty-Fifth Annual Allerton Conference on Communication, Control, and Computing*, pages 238–247, 1987.

[23] Duisberg, R. A. Animated graphical interfaces using temporal constraints. In *Proceedings CHI'86 Human Factors in Computing Systems*, pages 131–136. ACM, 1986.

[24] Emrath, P. A. and Padua, D. A. Automatic detection of nondeterminancy in parallel programs. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 89–99, 1988. Proceedings also published as SIGPLAN Notices, 24(1), January 1989.

[25] Even, S. *Graph Algorithms*, pages 138–139. Computer Science Press, Inc., 1979.

[26] Feldman, S. I. and Brown, C. B. IGOR: A system for program debugging via reversible execution. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 112–123, 1988. Proceedings also published as SIGPLAN Notices, 24(1), January 1989.

[27] Fidge, C. J. Partial orders for parallel debugging. *SIGPLAN Notices*, 24(1):183–194, January 1989.

[28] Flynn, M. J. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[29] Fowler, R. J., Leblanc, T. J., and Mellor-Crummey, J. M. An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors. *SIGPLAN Notices*, 24(1):163–173, January 1989.

[30] Fujimoto, R. M. SIMON: Simulator of multicompter networks. Technical Report UCB/CSD 83/140, UC Berkeley, 1983.

[31] Gait, J. A debugger for concurrent programs. *Software Practice and Experience*, 15(6):539–554, June 1985.

[32] Garcia, M. E. and Berman, W. J. An approach to concurrent systems debugging. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 507–514, 1985.

[33] Garcia-Molina, H., Germano, Jr., F., and Kohler, W. H. Debugging a distributed computing system. *IEEE Transactions on Software Engineering*, SE-10(2):210–219, March 1984.

[34] Goldszmidt, G. S., Katz, S., and Yemini, S. Interactive blackbox debugging for concurrent languages. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 271–282, May 1988. Proceedings also published as SIGPLAN Notices, 24(1), January 1989.

[35] Gordon, A. J. and Finkel, R. A. TAP: A Tool to Find Timing Errors in Distributed Programs. In *Workshop on Software Testing*, pages 154–163, July 1986.

[36] Guibas, L. J., Kung, H. T., and Thompson, C. D. Direct VLSI implementation of combinatorial algorithms. In *CALTECH Conference on VLSI*, pages 509–519, January 1979.

[37] Haban, D. and Weigel, W. Global events and global brreakpoints in distributed systems. In *Twenty-First Annual Hawaii International Conference on System Sciences*, pages 166–174, January 1988.

[38] Harter, P. K., Heimbigner, D. M., and King, R. IDD: an interactive distributed debugger. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 498–506, 1985.

[39] Helmbold, D. and Luckham, D. Debugging Ada tasking programs. *IEEE SOFTWARE*, 2(2):47–57, March 1985.

[40] Hockney, R. W. and Jesshope, C. R. *Parallel Computers: Architecture, Programming and Algorithms*. Adam Hilger, Bristol, England, 1981. SOR algorithm as presented by Michael J. Quinn, *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, pp. 140–142, 1987.

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

[42] Hough, A. A. and Cuny, J. E. Perspective Views: A technique for enhancing parallel program visualization. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II–124–II–132, 1990.

[43] Hseush, W. and Kaiser, G. E. Data path debugging: Data-oriented debugging for a concurrent programming language. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 236–247, May 1988. Proceedings also published as SIGPLAN Notices, 24(1), January 1989.

[44] Hseush, W. and Kaiser, G. E. Modeling concurrency in parallel debugging. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 11–20, March 1990. Proceedings also published as SIGPLAN Notices, 25(3), March 1990.

[45] Joyce, J., Lomow, G., Slind, K., and Unger, B. Monitoring distributed systems. *ACM Transactions on Computer Systems*, 5(2):121–150, May 1987.

[46] Kelley, Jr., J. E. and Walker, M. R. Critical path planning and scheduling. In *1959 Proceedings of the Eastern Joint Computer Conference*, 1959.

[47] Kung, S. Y. *VLSI Array Processors*, pages 215–216. Prentice-Hall, 1987.

[48] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.

[49] Lamport, L. The mutual exclusion problem: Part I–A theory of interprocess communication. *Journal of the Association for Computing Machinery*, 33(2):313–326, April 1986.

[50] LeBlanc, R. J. and Robbins, A. D. Event-driven monitoring of distributed programs. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 515–522, 1985.

[51] LeBlanc, T. J. Shared memory versus message-passing in a tightly coupled multiporcessor: A case study. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 463–466, 1986.

[52] Leblanc, T. J. and Mellor-Crummey, J. M. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[53] Leblanc, T. J., Mellor-Crummey, J. M., and Fowler, R. J. Analyzing parallel program executions using multiple views. *Journal of Parallel and Distributed Computing*, 9:203–217, 1990.

[54] LeDoux, C. H. and Stott Parker, Jr., D. Saving traces for Ada debugging. In *Ada in use, Proceedings of the Ada International Conference*, pages 97–108, May 1985.

[55] Lehr, T., Segall, Z., Vrsalovic, D. F., Caplan, E., Chung, A. L., and Fineman, C. E. Visualizing performance debugging. *Computer*, 22(10):38–51, October 1989.

[56] Lin, C. and Snyder, L. A comparison of programming models for shared memory multiprocessors. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II–163–II–170, 1990.

[57] Linton, M. A. A debugger for the Berkeley Pascal system. Master's Report, Computer Science Division, University of California at Berkeley, June 1981.

[58] London, R. L. and Duisberg, R. A. Animating programs using Smalltalk. *Computer*, 18(8):61–71, August 1985.

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

[60] Meldal, S., Luckham, D. C., and Haberler, M. A. Specifying ADA tasking using patterns of behavior. In *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences*, pages 129–134, January 1988.

[61] Miller, B. P. and Choi, J.-D. Breakpoints and halting in distributed systems. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 316–323, 1988.

[62] Miller, B. P. and Choi, J.-D. A mechanism for efficient debugging of parallel programs. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 141–150, May 1988. Proceedings also published as SIGPLAN Notices, 24(1), January 1989.

[63] Miller, B. P., Macrander, C., and Sechrest, S. A distributed programs monitor for berkeley UNIX. *Software Practice and Experience*, 162(2):183–200, February 1986.

[64] Milner, R. A calculus of communicating systems. In Goos, G. and Hartmanis, J., editors, *Lecture Notes in Computer Science(92)*. Springer–Verlag, 1980.

[65] Omondi, A. R. and Brock, J. D. Implementing a dictionary on hypercube machines. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 707–709, 1987.

[66] Pan, D. Z. and Linton, M. A. Supporting reverse execution of parallel programs. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 124–129, May 1988. Proceedings also published as SIGPLAN Notices, 24(1), January 1989.

[67] Pancake, C. M. and Utter, S. Models for visualization in parallel debuggers. In *SuperComputing '89*, pages 627–636, 1989.

[68] Peterson, J. Petri Nets. *ACM Computing Surveys*, 9(3):223–252, September 1977.

[69] Plattner, B. and Nievergelt, J. Monitoring program execution: A survey. *Computer*, pages 76–93, November 1981.

[70] Pratt, W. K. *Digital Image Processing*. John Wiley and Sons, 1978.

[71] Quinn, M. J. *Designing Efficient Algorithms for Parallel Computers*. McGraw-Hill, Inc., 1987.

[72] Reiss, S. P. Graphical program development with PECAN program development systems. *SIGPLAN Notices*, 19(5):30–41, May 1984.

[73] Roman, G.-C. and Cox, K. C. A declarative approach to visualizing concurrent computations. *Computer*, 22(10):25–36, October 1989.

[74] Rubin, R. V., Rudolph, L., and Zernik, D. Debugging parallel programs in parallel. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 216–225, May 1988. Proceedings also published as SIGPLAN Notices, 24(1), January 1989.

[75] Schiffenbaur, R. D. Interactive debugging in a distributed computational environment. Master's thesis, MIT, September 1981. Computer Science Department, Tech report number MIT/LCS/TR-264.

[76] Segall, Z. and Rudolph, L. PIE: A programming and Instrumentation Environment for Parallel Programs. *IEEE SOFTWARE*, 2(6):22–37, November 1985.

[77] Sequent Computer Systems, Inc. *DYNIX Pdbx Debugger User's Manual.* 15450 SW Koll Parkway, Beaverton OR 97006, 1986.

[78] Smith, E. T. Debugging tools for message-based, communicating processes. In *Proceedings of the 4th International Conference on Distributed Computing Systems*, pages 303–310, May 1984.

[79] Snodgrass, R. Monitoring in a software development environment: A relational approach. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 124–131, April 1984. Also SIGPLAN Notices, 19(6), June 1984.

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

[81] Stasko, J. T. Tango: A framework and system for algorithm animation. *Computer*, 23(9):27–39, September 1990.

[82] Stone, J. M. A graphical representation of concurrent processes. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 226–235, May 1988. Proceedings also published as SIGPLAN Notices, 24(1), January 1989.

[83] Tarjan, R. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2), 1972.

[84] Taylor, R. N. A general purpose algorithm for analyzing concurrent programs. *CACM*, 26(5):362–376, 1983.

[85] Taylor, R. N. and Osterweil, L. J. Anomaly detection in concurrent software by static data flow analysis. *IEEE Trans. on Software Engineering*, pages 265–278, May 1980.

[86] Teitelman, W. *Interlisp Reference Manual.* Xerox PARC, Palo Alto, CA, October 1978.

[87] Ullman, J. D. *Computational Aspects of VLSI*, pages 160–164. Computer Science Press, Rockville, Maryland, 1984.

[88] Utter, S. and Pancake, C. M. A bibliography of parallel debuggers. *SIGPLAN Notices*, 24(11):29–42, Nov 1989. Version 1.0.

[89] Utter, S. and Pancake, C. M. A bibliography of parallel debuggers. Technical Report CTC89TR17, Cornell Theory Center, Cornell University, Ithaca, NY 14853-5201, Sep 1990. Version 2.0. Also available in file pub/debugger.bib, accessible through anonymous ftp from eagle.cnsf.cornell.edu.

[90] Weems, Jr., C. C. *Image Processing on a Content Addressable Array Parallel Processor.* PhD thesis, COINS Department, University of Massachusetts, Amherst, MA 01003, 1984. Also COINS Technical Report 84–14.

[91] Wileden, J. C. and Avrunin, G. S. Toward automating analysis support for developers of distributed software. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 350–357, 1988.

[92] Wittie, L. D. Debugging distributed C programs by real time replay. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, pages 57–67, 1988. Proceedings also published as SIGPLAN Notices, 24(1), January 1989.

[93] Zave, P. A Distributed Alternative to Finite-State-Machine Specifications. *ACM Transactions on Programming Languages and Systems*, 7(1), January 1985.