Integrating Event- and State-Based Approaches to the Debugging of Parallel Programs

A Dissertation Presented

by

Joydip Kundu

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

Doctor of Philosophy

May 1996

Department of Computer Science

Integrating Event- and State-Based Approaches to the Debugging of Parallel Programs

A Dissertation Presented

by

Joydip Kundu

Approved as to style and content by:

Janice E. Cuny, Chair

Charles C. Weems Jr., Member

Krithivasan Ramamritham, Member

C. Mani Krishna, Member

David Stemple, Department Chair
Computer Science

Acknowledgements

The successful completion of this dissertation is made possible by the help and generosity of a number of people. First and foremost, I am indebted to Jan Cuny for her help and guidance. She was always available to point out the broader scheme of things whenever I lost sight of it. Her constant encouragement ensured that I not lose heart during months of apparent meandering that inevitably accompanies any research. The years I spent under her supervision have been the most satisfying professionally and academically.

I would also like to thank my committee: Chip Weems, Krithi Ramamritham and C. Mani Krishna for their patient reading of the manuscript and for suggesting ways to improve it. Chip Weems had always been generous with his time and advice whenever I needed it. His comments related and unrelated to the thesis have always been illuminating, and provided me with a better understanding about the field of distributed computing in general. C Mani Krishna introduced me to the issues of Computer Architecture that first attracted me to the field of parallel computing, and Krithi Ramamritham made me aware of the core issues of distributed computing in general.

Al Hough was patient in sharing his insights on Belvedere. I enjoyed collaborating with him during the initial design of Ariadne's modeling language. His inputs on the overall scope of the experiments were invaluable. Bruce Leban made many useful suggestions regarding the modeling language, and pointed out several incompleteness in the draft. His insights into LaTeX saved me hours of formatting. Allen Malony of University of Oregon provided useful feedback on how to incorporate performance modeling in Ave/Ariadne, and was very supportive throughout.

Chris Wilson, also from University of Oregon, helped me determine the algorithmic complexity of the modeling task.

This research allowed me to enjoy the warmth and hospitality of two places some 3000 miles apart. My stay at Amherst was made eventful by the company of my friends. Boijayanta Bezbarua, Kumar Varadaraju, Uday Hegde, Asis Nasipuri, and Rajesh Jha were there when I needed them the most. A.V.S. Sastry, Srinath Duvuru, Jan Telle, Ferenc Racogzi, Darryl Brown, Bernd Mohr and Takunari Miyazaki made my brief stay at Eugene enjoyable. Pallab Midya was available over the telephone from Champagne whenever I needed him.

Special thanks also goes to Ilya Gertner of Encore Computer Corporation, for the stimulating discussions on applications of event-based modeling techniques. Jim Sullivan had been extremely accommodating during my stay at Digital Equipment Corporation whenever I needed to take time off for writing the thesis.

Sharon Mallory and Glen Stowell of University of Massachusetts and Jan Saunders and Lauradel Collins of University of Oregon took care of the administrative details that allowed me to concentrate on the research.

Finally I would like to thank my parents Girija and Kalyani Kundu for their unwavering support, and my wife Gargi Chakrabarty for being so patient with me during the most hectic days of the research and the writing of the dissertation.

ABSTRACT

INTEGRATING EVENT- AND STATE-BASED APPROACHES TO THE DEBUGGING OF PARALLEL PROGRAMS

MAY 1996

JOYDIP KUNDU, B.Tech(Hons), INDIAN INSTITUTE OF TECHNOLOGY, KHARAGPUR

M.S.E.C.E, UNIVERSITY OF MASSACHUSETTS, AMHERST

M.S., UNIVERSITY OF MASSACHUSETTS, AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Janice E. Cuny

This dissertation presents a comprehensive solution to the problem of debugging of parallel programs. The proposed strategy employs an event-based modeling phase at the highest level where gross patterns of process interactions are investigated for anomalous behaviors. The results of the modeling phase are used to guide the investigation of the local states of the processes by stopping the computation at a consistent global state specified in terms of abstract events. The primary contributions of the dissertation are: (1) A simple modeling language that is capable of providing precise feedback on an error, (2) A scalable visual feedback mechanism that allows the user to control the balance between the scale and the precision of the feedback with the help of a query language, and (3) A novel abstract event-based breakpoint specification scheme that integrates two disparate strategies of debugging parallel programs.

Parallel programs are hard to debug. State-based techniques lack mechanisms to accommodate a sparsity of consistent global states, a large volume of trace data and asynchronous execution that characterize parallel program behaviors. Event-based techniques provide abstraction mechanisms to contend with large quantities of data, and use logical time to filter out the perturbations due to asynchrony. They, however, do not provide sufficient feedback on match failures, they do not scale well for massive parallelism, and they cannot track errors back to the source

code. Our approach is to integrate the two techniques in a meaningful whole: event-based modeling for its abstraction facilities, and state-based techniques to explore local process behaviors in detail. The user is thus presented with a unified debugging framework where an anomaly detected at the level of abstract behaviors can be traced back to the offending source code. The need for such an integrated environment has long been recognized, and this dissertation introduces a method by which such a comprehensive solution can be implemented.

Table of Contents

List of Figures

Chapter 1

Introduction

The debugging of parallel programs is often difficult. State-based techniques [92, 73] allow the user to directly examine an execution to an arbitrary level of detail and to relate errors to source code constructs. They lack, however, mechanisms to accommodate a sparsity of consistent global states, a large volume of trace data, and asynchronous execution; as a result they are quite ineffective in practice. Event-based techniques offer an alternative in which user-specified models of intended program behavior are compared to actual program behavior as captured in execution traces [8, 9, 39, 41]. The use of event traces in conjunction with replay mechanisms avoids issues of global state consistency [18, 58] and reproducibility; the use of abstraction makes it possible to contend with large quantities of data; and the use of logical time transformations filters out perturbations due to asynchrony [21, 51]. Current event-based debugging tools, however, have three major limitations: they do not provide sufficient feedback on match failures, they do not scale well for massive parallelism, and they do not facilitate tracking errors back to the source code.

This dissertation is based on the premise that parallel debugging requires a multi-level strategy. The strategy we propose employs event-based techniques [8, 9, 53, 39, 41] at the highest level where gross patterns of process interactions are investigated. As debugging proceeds and the focus of attention narrows, the behavior of progressively smaller parts of the program are analyzed in progressively finer detail, using a combination of event- and state-based techniques. Finally, at the lowest level, when the error has been isolated to specific sections of sequential code, traditional state-based techniques [44, 92, 73] are used. This integrated strategy effectively addresses the inadequacies of both approaches: it provides powerful abstraction mechanisms for understanding complex behaviors that purely state-based approaches lack, and it provides the ability to examine state information at an arbitrary level of detail that standard event-based debuggers lack. But how can such an integration be achieved? And how do we address the remaining deficiencies of event-based debuggers, namely their lack of detailed feedback and scalability ? These issues are addressed in this dissertation. Specifically, we design a simple modeling language that can be used to specify complex behavior without losing the ability to detect and characterize errors. We introduce a scalable visualization technique that works in tandem with a query interface to support a hierarchical exploration of the program behavior and provide the user with an ability to control the scalability and the preciseness ratio of the visual feedback. Most importantly, we provide an integrated debugger implementation where the results of the event-based modeling can be used to stop the computation at globally consistent states and guide the interactive phase of the debugging to relate the bug to the offending source code construct.

For the remainder of this chapter, we discuss issues in parallel debugging, describe contributions of the thesis, and give an overview of the integrated environment.

1.1 Issues related to Parallel Program Debugging

Often we try to solve a complex problem by enhancing the solution used to tackle a simpler version of the same problem. This would suggest that one can adapt the standard break/examine/repeat strategy used in sequential programs to accommodate parallel program debugging. This strategy requires the program be executed repeatedly and stopped at various points in the execution to examine its states. This technique is, however, fundamentally incompatible with parallel program debugging due to the following reasons.

- *Nondeterminism and Irreproducibility:* Non-reactive sequential programs (those that do not deal with real-time events) are deterministic, and their behavior is reproducible. Given the same set of inputs a sequential program will follow the same state transitions and produce the same outputs every time it is executed. Parallel programs, on the other hand, have multiple threads of control that may interact with each other in a non-deterministic manner. In a shared memory program, the access order to shared variables may depend nondeterministically on the load of the system. Similarly, in a message passing program, a process may non-deterministically receive one message of many available in its message queue. As a result, different executions of a parallel program may take different state transitions. This irreproducibility means that repeated executions of the same program with the same input may not manifest the same bug consistently.

  One way to handle irreproducibility is to log the resolution of all non-deterministic accesses and then to ensure that such accesses are resolved the same way during re-execution. This is known as *replay.* Often the efficiency of replay mechanism is increased by saving the program state at periodic intervals in a process known as *checkpointing.* In this way, the user does not need to restart the program from the very beginning to explore its states. However the implementation of logging and checkpointing adds extra computational overhead, which can alter the program behavior. This is known as *probe effect* [29] and gives rise to non-transparency.

- *Non-transparency:* Sequential debuggers are transparent with respect to non-reactive programs, that is, program monitoring does not change the behavior of the program. Parallel debuggers cannot guarantee such transparency. Monitoring a single process can change the order in which shared resources are accessed, and hence alter the program behavior. There is no software solution to the probe effect. But use of additional hardware can achieve transparent monitoring [80], albeit at a prohibitive cost.

- *Specification and Detection of Consistent Global States:* Parallel programs contain multiple loci of control and the core behavior of a parallel program is determined by the inter-dependence among the multiple loci through interprocess communications. A global state of such a computation can be thought of as a set of local states, but a simple enumeration of individual loci of control does not guarantee a meaningful global state. For example, a local state of process A where the process has received message M from process B, and one on process B where the process is about to send message M to process A both make sense individually, but a global state that contains both these local states is meaningless – it can never exist in any execution of the program and is thus inconsistent. Specifying consistent global states in presence of multiple processes is difficult: it is even harder to stop the computation at such a consistent state once it is specified. Manabe and Imase [58] have proved that in order to stop a parallel program at certain states, the program may need to be replayed twice.

- *Asynchrony:* In absence of a global clock, processes in a asynchronous parallel program do not execute in lock-step. Temporal relationships between events in a parallel program are only partially defined. To a central observer, events in different processes may, as a result, interleave in unexpected ways. Most of such interleavings are arbitrary and irrelevant: program correctness depends only on the event orderings imposed by data dependencies captured by Lamport's *happened before* relationship. To be effective, a debugger must be be able to filter out the perturbations due to temporal asynchrony. In event-based approaches, this is done by considering only the *happened before* relationships among the events determined either by program order[1] or by interprocess communication, and ignoring the order implied by physical time.

- *Large State Space:* Even if the user is willing to tolerate the additional overhead of monitoring, state-based techniques are not suitable for parallel programs. Multiple threads of control produce an enormous state space – a large part of which may not be meaningful to the user. One approach toward accommodating large state space is that of abstracting out the low level details of the program by concentrating on the salient features of the behavior. Such a modeling approach forms the basis of event-based debugging, where the user specifies the core characteristics of the program behavior in terms of events collected during execution. State-based approaches provide no such abstraction mechanism, and thus are not amenable to hierarchical exploration of program behavior.

1.2 Approach

Event-based debugging techniques are generally better suited to handle parallel program behavior: replay mechanism can be incorporated to ensure reproducibility of executions, abstract behavioral modeling based on events can accommodate large data set and avoid the issues related to state consistency, and the use of logical time based on Lamport's *happened before* relationship can contend with the arbitrary event interleaving of asynchronous executions. Widespread use of such techniques has been impeded, however, by three major inadequacies: they do not provide sufficient feedback on match failures, they do not scale well for massive parallelism, and they do not facilitate tracking errors back to the source code. We are primarily interested in providing a solution that will overcome these

[1] A program order is the order imposed on events occurring on a process due to the sequential nature of the process.

inadequacies. We incorporate replay technique as part of our debugging environment to ensure reproducibility, and we incorporate the notion of logical time in our modeling language to make it impervious to asynchronous executions. The three inadequacies that we attempt to overcome motivate our research to focus on three broad areas. The goal of providing precise feedback on match failures required that we handle a large state space and incorporate the correct notion of time in the modeling language. It led us to explore simple but effective abstraction and filtering mechanisms that give the debugger the ability to accommodate complex behavior without losing its error detection capability. The goal of accommodating massive parallelism required us to address the issue of scalable visualization techniques that support hierarchical exploration of program behavior. Finally, the goal of tracking errors back to the source code constructs required analysis of consistent global states, their specification, and their relationship with abstract events. We have adopted a hybrid approach comprised of a post-mortem modeling phase followed by an interactive debugging phase. The novelty of this approach is that the interactive phase is guided by the findings of the post-mortem modeling, and the user is free to switch back and forth between the two phases. The post-mortem modeling requires collection of event traces; the same trace file is used for replay during the interactive phase of debugging, allowing the debugger to stop the program in a meaningful global state.

Our approach can be applied to both shared-memory and message-passing programs. Most of the debugging experiments are performed on a message-passing parallel machine due to the availability of that machine. The typical steps involved in our integrated approach is outlined in Figure 1.1. The user starts with a parallel program instrumented to log messages or the order of the shared variable accesses for future replay purposes. Initial instrumentation is limited to communication events to keep the overhead at a minimum. He/she then writes a behavioral specification which is matched with the collected event trace. The user can collect a more detailed execution trace by reinstrumenting the program and running it under the replay manager and then specifying a more detailed behavioral model. The replay manager consults the initial log to ensure that re-executions resolve interprocess interactions in the same way as in the original run. If the user wants to examine the program states, he/she can then use the event-based model to set a consistent global breakpoint before/after the occurrence of an abstract event. Our debugger ensures that the user's request result in a consistent global state, and annotate the execution log with breakpoint markers that stops individual processes when run under the replay manager at states that as a collection correspond to a consistent global state. The user can then use the commercial vendor-supported debugger available for the particular machine to examine individual local states and find the source code construct(s) responsible for the error. Once the user has accounted for all errors in this fashion and corrected the source code constructs responsible for them, he/she can proceed to performance tuning, the next phase of parallel program development.

1.3 Contributions

This dissertation addresses the problems of debugging asynchronous, massively parallel programs by introducing an integrated environment that allows the user to take advantage of event-based modeling and scalable visualization to guide global state exploration of the program. Its contributions are three-fold:

1. It demonstrates that it is indeed possible to design a modeling language that can be used to specify complex behavior without losing the ability to detect and characterize errors. We present a simple modeling language that supports sophisticated filtering mechanism, comprehensive logical time manipulation facility, and a powerful functional query interface that aids the user to specify and explore complex program behavior.

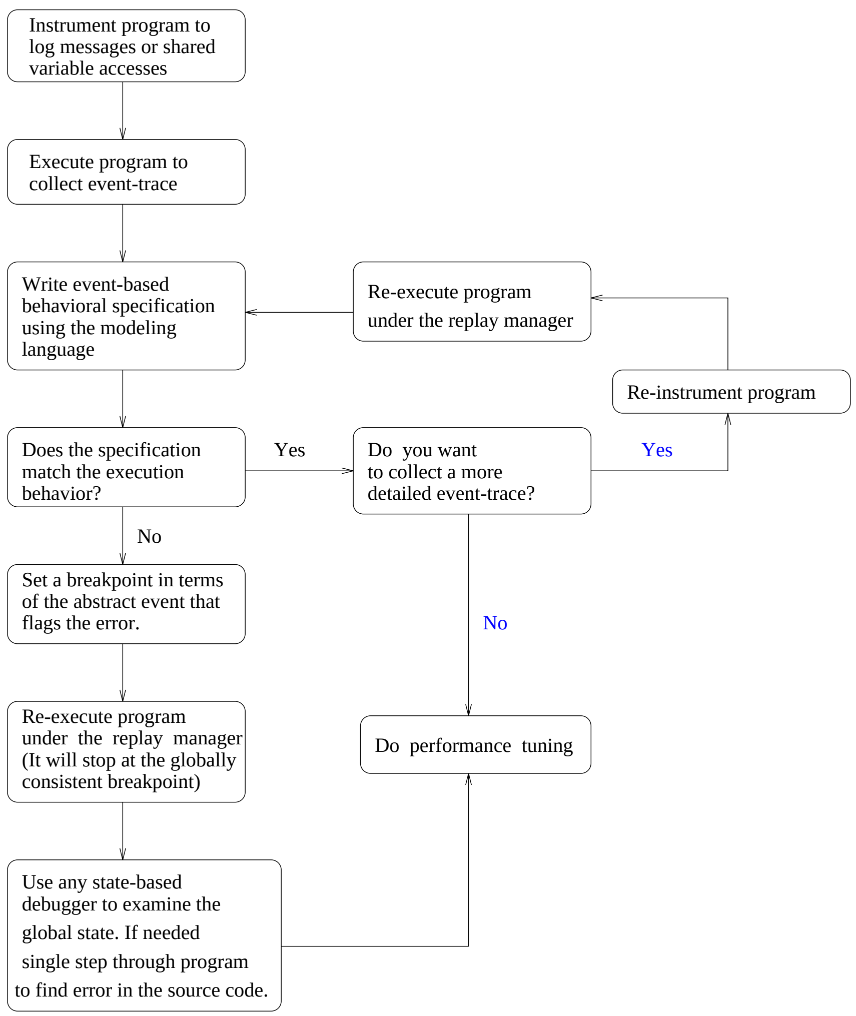


Figure 1.1. Overview of the integrated debugging approach.

2. It introduces a scalable visualization technique that not only enables the user to understand the program behavior, but also acts as a query interface. The user retains the ability to control the balance between scalability and precision of the visual feedback with the help of the query language.

3. It provides the design and implementation of an integrated debugging environment, where the event-based model is used in conjunction with the scalable visualization of the behavior to specify a globally consistent breakpoint. This approach allows the user to re-execute the program under the control of the trace file in order to examine the program state to an arbitrary level of detail. The event-based component of the environment makes the powerful abstraction mechanism available to the user to help narrow the focus of investigation to the offending part of the behavior. This behavioral approach of state selection bridges the gap between the two approaches – it enables the use of a powerful abstraction mechanism so lacking in purely state-based debuggers, and it allows us to track an error back to the offending source code construct that has so far been unavailable with purely event-based debuggers.

1.4 Outline of the Dissertation

Chapter 2 is an overview of the current status of parallel debugging. We also discuss related topics of sequential debugging, program animation systems and performance monitoring systems as they address some common issues of large data set, asynchronous execution and scalable visualization although under different contexts.

Chapter 3 introduces the modeling language and the spreadsheet-oriented query facility based on the scalable visualization technique of Ave/Ariadne.

Chapter 4 contains six debugging examples to illustrate the query and modeling capabilities of Ave/Ariadne, and it also presents the simple filtering scheme used in Ariadne's modeling.

Chapter 5 discusses the behavioral approach to consistent breakpoint specification. The greedy algorithm that we use to stop a program at an user-specified consistent global state is also introduced along with a proof of its correctness.

Chapter 6 summarizes the results of the dissertation, and includes directions for future research and our conclusions.

Chapter 2

Related Work

A bug is an unwanted property of a program that manifests itself as an anomalous behavior. At the onset of debugging, only its manifestation is apparent and not its underlying cause. The task of debugging is to relate the the observable anomalies (the manifestations) to the underlying errors in the source code constructs. In this chapter we will review both theoretical and experimental techniques for analyzing and debugging parallel programs.

Any empirical approach requires the ability to perform repeatable experiments. We first review sequential debuggers where the ease of reproducibility of non-reactive programs has been used in the cyclic break/examine/repeat method. As with any experiment, debugging requires analysis of data collected regarding the state of the programs. We discuss two related research domains where analysis of a large amount of data is required; program animation systems are covered in Section 2.2 and performance monitoring systems in Section 2.3. In both cases, we examine whether some of the described techniques can be imported to our domain of parallel program debugging. Section 2.4 discusses the theoretical approaches used for parallel program analysis, and finally, in Section 2.5, we investigate the empirical approaches used for debugging of parallel programs.

2.1 Sequential Debuggers

The most common technique for debugging sequential programs is that of break/examine/repeat where the program is stopped at different states in its execution path, and those states are examined for correctness. This is the technique used in most commercial sequential debuggers. The debuggers provide three basic functionalities to the user: *state selection* by setting a breakpoint at a particular statement or by single stepping through statements; *state visualization* through examining the values of different variables in the program; and *state modification* through setting variable values. State selection via a higher level mechanism such as predicates on variable values is also available in debuggers like Ladebug [10]. Although these debuggers allow the user to examine behavior at the source level, user interaction is still at a low level of abstraction.

For large and complex sequential programs, a debugger should provide support for correlating the occurrences of several logically related breakpoints into a single, and more abstract occurrence. Few commercial debuggers provide such a service. Commercial debuggers are also inadequate in the variety of actions that they support once a breakpoint has been reached — the options being limited to examining single variable value at a time; facilities such as printing an aggregate structure like a linked list in its entirety is not provided. Moreover debugging requires viewing a program execution from a variety of view points. To this end, the standard services provided by the break/examine/repeat methodology is inadequate. Several non-commercial debuggers have attempted to address these issues.

Reiss in his PECAN [77] programming environment allows multiple views of the abstract syntax tree representation of the program. These views include structured flowcharts, module interconnection diagrams, symbol tables and data history. Execution can be animated by highlighting the specific part of the flow graph display where the thread of execution currently resides or has visited previously. Balzer[7] in his EXDAMS debugger also records all variable accesses to provide the user with a data history that includes a tree-oriented display showing how a current value of a variable is obtained from the values of other variables. This *flowback analysis* technique has also been used in parallel program debugging by Choi and Miller [62]. Similar event-based technique of logging all variable accesses has also been used in PROVIDE [65] to animate complex program states.

The most high level event-based approach to sequential debugging is imported from the domain of parallel debugging. Olsson, Crawford and Ho in their Dalek system [70] incorporate many ideas found in the event-based parallel debugging field in order to build a high level sequential debugger. Dalek takes a dataflow approach to event recognition. Primitive events are generated via instrumentation of the source program: a high level event recognition algorithm can be triggered when any of the constituent primitive events is generated. The debugger is programmable, the user can even change the matching algorithm. Dalek can be thought of as a sequential variant of Event Based Behavioral Abstraction toolkit (EBBA) [9] developed for parallel programs. EBBA is described in Section 2.5.3.

2.2 Program Animation Environments

Program animation environments are mainly geared toward educational purposes, where the animation of interesting properties of the algorithm can illuminate the working of a program. This is very similar to the goal of debugging systems where interesting properties of a program are investigated for clues as to the cause of an anomalous behavior. It is not surprising therefore to find animation systems trying to cope with issues related to event collection, modeling, visualization and logical time that are central in the context of parallel debugging environments. These similarities between the two domains have led to the development of several abstract animation systems [39, 50] geared expressly toward debugging of parallel programs.

Animation systems require the user to specify the events to be animated, select or design animation objects that will form the animation, and define a mapping between the program objects and the animation objects. There are two main types of animation systems – those that follow an annotative approach, and those following a declarative approach.

Annotation-based animation systems collect interesting events by instrumenting the program, and then specify the animation scheme in terms of the collected events and animation icons. Annotation-based parallel program animating systems include Voyeur [85], Belvedere [39] and J2 [50]. Voyeur's architecture resembles that of BALSA [12], an animation system developed for sequential programs; and it supports a class hierarchy of views to create animations at different levels of abstraction. Voyeur, however, does not handle asynchrony properly its notion of time being limited by the assumption of the existence of a global clock.

The first sophisticated animation system that provides abstract visualizations for parallel program debugging was Belvedere [39]. Belvedere extends Lamport's *happened before* relation to abstract events and animates the program execution in logical time to filter out perturbations due to asynchrony. EDL [9] is used as the modeling language. Belvedere also introduces the concept of *perspective views* where the user can choose to animate the behavior from a number of different perspectives. Belvedere, however, does not address the issue of scale that arises with massive parallelism.

Leban addresses the issue of scalability in a comprehensive manner in his J2 system [50]. J2 animates the program execution in terms of the representative behavior of the program by combining processes exhibiting similar behaviors into a group. J2 has been shown to accommodate fixed communication patterns scalably. It is not clear, however, whether this technique can be successfully applied to data-dependent behaviors: whether for representatives for such behaviors can automatically be determined, and, more importantly, whether the automatic grouping of events in terms of representatives will be useful for data-dependent behaviors. This work is still under progress.

All annotation-based animating system share a common characteristic: programs need to be annotated afresh and recompiled if the user wants to modify the animation. In contrast, Roman and Cox in their Pavane animation system [78] propose a declarative approach. Pavane is for animating programs that run on Swarm operating system, which has a shared tuple-space like Linda [3]. Instead of annotating the program, the user declares the animation events that need to be displayed when the global state change satisfies the user-defined criteria. Thus, unlike the annotation schemes, the original program does not need to be recompiled in order to build a new animation. A declarative approach, however, is difficult to implement in a distributed message passing environment where consistent global states are harder to detect.

Except for J2, none of the animation systems address the issue of scalable visualizations, and are thus incapable of handling massive parallelism. The modeling languages used in the context of animation systems are not designed for providing detailed feedback in case of an error in the execution. Often if a model does not match the actual execution an animation system simply fails to animate the execution without providing precise reason for such a failure. Moreover parallel programs display a variety of complex behaviors: it is not trivial to decide on suitable animations to highlight particular nuances of the behavior. One is left with the choice of using animation system that targets a particular facet of the program execution (like Belvedere's emphasis on communication patterns) or insist that the user be sophisticated enough to design the animations. Animation systems can be successfully used as a part of an integrated debugging environment and can aid in debugging of programs but cannot function well as stand-alone debugging systems.

2.3 Performance Monitoring and Visualization

Performance monitoring and visualization systems help users to find bottlenecks in the program and suggest ways to modify the program so as to achieve efficient execution. Performance monitoring systems must collect, analyze and display a large amount of data. Thus the issues of scalability and analysis of behavior faced by correctness debuggers are also addressed by these systems, although in a different context. Performance debuggers deal with global time which imposes a total order among the "interesting" events. This total order makes it easy to provide a mechanism for abstraction and filtering.

Inter-process communication plays a pivotal role in parallel programs. A standard approach for transparently monitoring distributed program behavior is to instrument the interprocess communication library. Each instrumented system call to send a message, generates a duplicate message to central a filter process that contains a user-specified template to determine whether the message is to be stored in the trace. This architecture has been used by Miller, Macrander and Sechrest [63], and in the Jade system by Joyce *et al.* [45] to collect event traces for performance analysis.

The other aspect of performance monitoring deals with the analysis of the large data set collected by program instrumentation. Several debuggers, including Seeplex [19], SIEVE [81], and Chitra [1] address the issue of scalable visual feedback as an aid to data analysis in the context of performance debugging. Scalability is achieved in Seeplex [19] and Chitra [1] by grouping related behaviors together into categories. SIEVE [81] allows the user to focus his/her attention on a particular portion of the event trace through selection of nodes in the static call graph of the program. Trace data is highly structured, and the user can specify the structure using an event syntax. There is a spreadsheet interface for retrieving and analyzing data. Temporal filtering is done on selected data by using historical queries. A spreadsheet interface to specify distributed data is also proposed by Couch [20], but it lacks the static call graph based filtering mechanism presented in SIEVE.

Another approach in performance debugging borrows heavily from the event-based approach for correctness debugging. Perl and Weihl [75] proposed PSpec, an event-based modeling language for performance assertion checking. Users can identify an interval in terms of its start and end events. Complex predicates can be specified over the events by defining a metric of interest and specifying assertions about that metric. PSpec can be considered as a variant of EDL the modeling language of EBBA [9] that will be discussed in Section 2.5.3.

Performance and correctness debugging of parallel programs are similar in spirit: both need to collect "interesting" events, both have to deal with a large data set, both require behavioral modeling, and both need to address the issue of scalable feedback. Performance debugging, however, deals mainly with aggregate effects (the number of times a particular function is called or the total overhead of a particular function call) and thus can often ignore specific complex interactions between individual events. This reduces the overhead of event collection. The behavioral modeling phase of a performance debugger is straight forward due to the total order among events imposed by its assumption of a global clock. Providing scalable feedback in the context of performance debugging is also easier: subtle difference in performance metrics can be ignored without compromising the integrity of the feedback; for instance, processors having utilization of 95% and 97% can probably be grouped in the same category of a visual display without affecting the overall quality of the results. Correctness debugging allows no such flexibility: Massive parallelism demands categorization to ensure scalability, while the very nature of the task requires even little deviations be highlighted lest a bug go undetected.

2.4 Analysis of Parallel Programs

Theoretical analysis can be used to prove that a program obeys certain kinds of properties. These properties are broadly divided into two categories: *safety properties* saying nothing bad will ever happen, and *liveness properties* saying that good things will eventually happen. Safety properties include freedom from deadlock, and absence of access anomalies. Liveness properties include absence of starvation and fairness. Analysis techniques that attempt to prove that a program obeys certain properties based on the analysis of the source code alone, without the actual execution of the program, are known as *static* analysis techniques. *Dynamic* analysis techniques use a trace file of events collected during the execution, and can be *online* or *post-mortem.* The formal analysis techniques can be broadly grouped into three categories — state-based, proof-based, and data flow analysis techniques.

State based techniques attempt to compute all possible states of a parallel program by *control-flow* analysis, and construct a graph where nodes represent the state of every task in the system, and edges represent the execution of a single statement by a single task. Such a graph – called a *concurrency history graph* can then be analyzed for properties such as presence of deadlock or potential access anomalies [91]. A deadlock state can be found, for example, by searching the graph for a node that represents an active task without a successor. Potential race conditions can be found by looking for a node with more than one outgoing edges representing accesses to the same shared variable. Alternative state-based approaches construct a petri net [76] representation of the parallel program, where a place represents a concurrent state of the program, and a transition represents the execution of a single statement by a single task. These techniques rely on reachability analysis of the resulting petri net.

The primary problem with state based analysis is the explosion of state space: the number of states in a concurrency graph and the number of places in a petri-net is exponential in the number of tasks. Long and Clarke [57], and McDowell [60] have suggested ways to construct the graph from task interaction analysis rather than from control flow analysis to reduce the size of the graph. Under these techniques, the states between two synchronization operations are collapsed into one single state. Taylor [91] has also suggested a *parceling* technique that analyzes the graph in pieces.

Reduction techniques have also been proposed in the domain of petri nets. Valmari [93] and Godefroid and Wolper [32] suggest techniques that observe the fact that when two mutually independent concurrent actions are possible from a state, the order in which the actions are executed does not affect the invariance of the safety properties. The state space can thus be constructed by choosing one arbitrary ordering, and ignoring the rest, and the analysis techniques would still work. Valmari [93] proposes a way to choose the particular order by finding a special set of transitions for every state represented in the petri net, called the *stubborn set*. Godefroid and Wolper [32] give a separate implementation.

In contrast to the reachability analysis proposed by state-based techniques, proof-based techniques try to model the program under study using some axiomatic system. The idea is to derive certain properties of simple systems using the axiomatic framework, and then show equivalence between a known system and the program in hand. The algebraic frameworks developed for studying equivalences for parallel programs include Calculus of Communicating Systems (CCS) by Milner [64] based on *observational equivalence* of traces, Communicating Sequential Processes (CSP) by Hoare [38] based on *failure semantics*, and Asynchronous Concurrent Processes (ACP) by Baeten and Weijland [6] based on *bisimulation equivalence* of traces. ACP can model multicast as well as point-to-point communication where as CCS and CSP can only model point-to-point communication. All can be used to detect deadlocks. Several tools have been developed using these three frameworks: LOTOS [11], ECCS [14], PAL [95] and Concurrency Workbench [17]. None, however, can perform induction over a number of identical processes, and are thus not scalable.

The final technique – data flow analysis – has been used for finding access anomalies in parallel programs. The basic approach is to build a partial ordering graph either from source code analysis or execution tracing where each node represents execution of a statement, and edges represent temporal ordering between the nodes. The temporal ordering between statements executed on different processes can be estimated with the help of synchronization operations. Each node is also annotated with the shared variable set that it reads or writes. If two nodes that are unordered by the temporal ordering (i.e., there is no path connecting them) access any shared variable in a conflicting manner (i.e, one reads and the other writes, or both writes) then there is an access anomaly or data race.

Static data flow analysis techniques [24], using compile-time data dependence analysis, cannot capture all the temporal dependencies exactly, and have to be conservative in their estimation. Thus they can report an unacceptable number of *false positives*. Dynamic data flow analysis techniques capture the temporal ordering between nodes by instrumenting the program and collecting the execution trace. On line techniques keep track of all the accesses to shared variables — if any two accesses are in conflict, there is an access anomaly. Since keeping track of all accesses is potentially very expensive, one optimization technique is to keep track of only the latest read and write accesses to a shared variable. Although this will not detect all the races related to the variable, it guarantees that at least one race involving that variable will be detected. Such optimizations in dynamic data flow analysis for race detections have been proposed in [16, 67].

Program analysis techniques are not scalable, and they can account for only a limited types of errors. State-based techniques are prone to explosion in the state space and are thus unable to accommodate massively parallel programs. Proof-based techniques cannot use induction over a large number of processes and can only be used on small systems. Dataflow analysis is only applicable for detecting potential access anomalies, and current techniques are limited to a restricted set of synchronization operations. Though deadlock and potential access anomalies are two very important categories of bugs they are by no means exhaustive. Other types of bugs, for example subtle errors in communication pattern because of uninitialized variables or accessing memory space freed earlier in the program, cannot be detected by analysis techniques, and can only be accommodated in interactive debugging. Thus analysis techniques can be used as a complement to debugging and not currently as a replacement.

2.5 Analyzing, Monitoring and Debugging Parallel Programs

Empirical debugging approaches to parallel programs can be divided into the categories shown in Figure 2.1 according to their schemes for specification and verification of hypotheses regarding the bug.

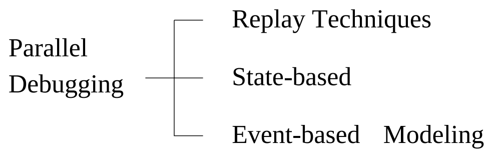


Figure 2.1. Taxonomy of Parallel Debugging Techniques.

Sections 2.5.1 through 2.5.3 describe parallel debugging systems using the empirical approach. As noted above, parallel programs often lack a common clock, have a very large state space, and may have embedded non-determinism that make programs irreproducible, and show perturbations due to asynchrony. Most of the techniques address only a subset of these issues. We group debugging systems into three broad categories — replay-oriented, state-oriented and event-oriented — on the basis of the dominant approach taken by the debugger. Replay techniques address the issue of reproducibility and thus can be incorporated in both state- and event-based debugging. State-oriented debuggers are often interactive, and they attempt to extend the sequential debugging techniques to the parallel domain. Event-based techniques are more innovative, as they circumvent the problems of consistent global states by modeling the execution in terms of events and the temporal relationships that exist between them.

2.5.1 Replay Techniques

Reproducibility of experiments is essential for the application of empirical approaches. Replay techniques guarantee reproducibility by logging the resolution of non-deterministic choices during the initial execution. The log is consulted during re-execution to ensure that the reexecution follows the same state transitions as in the initial execution. Replay systems for message passing programs (for example Radar[51], Belvedere [39], Agora [27], and Recap [71]) log all sends and receives of messages; those for shared memory programs (for example Instant Replay [52], Recap [71]) log the order in which shared variables are accessed. Systems like Recap [71], Igor [25] and Agora [27], also capture the system state periodically through *checkpointing*, thereby providing the user the capability of *random access playback* [1]. Recap [71] has an innovative checkpointing scheme that reduces logging overhead by making use of operating system resources. At periodic checkpoints, each process forks a child that inherits its data structures and state; the child is immediately halted after its creation. All data values read or written by a process from shared locations are stored in a log. During replay, any halted child can be selected, and executed to reproduce the program behavior with the help of the system log. Recap can be used for both shared-memory and message passing programs. Igor [25] optimizes checkpointing overhead in a virtual memory systems by writing out only *dirty* pages. Since it does not record events during execution, the replay may not reproduce the original program behavior. Local checkpoints taken independently (without global synchronization) cannot guarantee that a set of local checkpoints would constitute a consistent global state of the program. None

[1] This facility is mistakenly called *reverse execution* by many authors.

of the debuggers mentioned above can guarantee global consistency with their local checkpoints: they can only allow process-specific random-access playback. Netzer and Xu have addressed the issue of consistent global checkpoints in [69], by providing a scheme that can identify sets of local checkpoints making up consistent global states.

Replay techniques have been enhanced to be used directly for debugging purposes. Stone makes use of non-deterministic replay [89] to give the user the capacity to ask *what-if* questions. During execution, all messages are logged. The system then shows the potential concurrent local states of a program via *concurrency map* in which local states within a process between two communication events are coalesced into one state. During *speculative* replay, the user can force any of the potentially consistent global states to hold and reexecute the program with the message log. A reexecution that differs from the original execution indicates that additional synchronization constraints are needed for correct execution.

Miller and Choi extended Balzer's flowback analysis for shared memory parallel programs in their PPD debugger [62]. The debugger uses interprocedural data flow analysis to identify the variables that may be read at each block. Instrumentation is generated automatically to save all the variables at the beginning of a block that might be read before the next log point. At the end of each block, variables that have been written since the last checkpoint are saved. The debugger is thus able to give a flowback analysis of any variable by simply scanning the log or, if necessary, by reexecuting the program between checkpoints.

All the replay techniques described above introduce probe effect through the logging overhead. To overcome that, systems have been designed to achieve non-intrusive monitoring. Rubin, Rudolph and Zernik in their shared bus Makbilan architecture [80] make non-intrusive replay possible by having shadow processors monitoring processors and the main bus. The monitoring processors are connected by their own private bus. Message logging thus introduces no probe effect. Bacon and Goldstein [5] proposed a similar hardware based technique.

2.5.2 State-Based Debugging

The most common approach to parallel program debugging is to extend sequential debugging techniques to the domain of parallel programming. The basic facilities of location, visualization and modification of states are augmented to handle the aspects of interprocess communication. Commercial multiprocessor debuggers like Pdbx by Sequent Corporation [73], or udb provided by KSR [92] stop the execution on per-process basis. Breakpoints can be set by specifying the statement number: all threads are suspended when the statement is executed by any thread. The user is allowed to choose any one of the suspended threads and run it in isolation. Gait [29] proposed a debugger for concurrent processes running on a single processor. This method, however, cannot be extended to parallel programs running on multiprocessor machines. Parallel programs contain multiple processes interacting among themselves through inter-process communication. A global state of such a computation can be thought of as a set of local states. A simple enumeration of the individual loci of control does not, however, guarantee a meaningful global state. A local state of process A where the process stopped after it has received message M from process B, and one on process B where the process is stopped before sending message M to process A make sense in isolation, but a global state that contains both these local states is meaningless – it cannot exist is thus inconsistent. Simply gluing sequential debuggers where breakpoints are specified by selecting statements in the source code may result in such an inconsistent state.

The issue of consistent global states in the context of distributed computation has been addressed by many researchers. Distributed marker passing algorithms are proposed by Chandy and Lamport [15], Miller and Choi [61], Haban and Weigel [35], and Hurfin, Plouzeau and Raynal [43] using specially tagged messages to distribute information about local states across different processes and stop the whole computation in a consistent global state. In contrast, Manabe and Imase [58] present a centralized algorithm that ensures, for global predicates specifiable as conjunctions of local predicates, a debugger controlling the distributed system can halt all the processes at the earliest consistent global state. This requires running the program under the control of the debugger, and can induce time delay. They also prove, that for global predicates specified as a disjunction of local predicates, this can be accomplished only by replaying the trace twice.

Several state-based techniques also subsume the functionality provided by run-time monitoring systems. Jade [45] described in Section 2.3 and SPIDER [82] are examples of such systems. Jade allows the user to control the total ordering of the interprocess communication through a system-wide controller process. The SPIDER system by Smith [82] uses demons that continuously monitor the message traffic. The demons can atomically examine, interpret and modify interprocess communication. The user can write a boolean *trigger function* for each demon, and can assign priority classes to determine which demon should execute first in case of multiple firing. SPIDER also has a transcriber that logs all interprocess interactions for global replay.

State-based debuggers addressing the issue of consistent global states can stop the program execution at a global state meaningful from the perspective of the programmer. They lack, however, a powerful abstraction mechanism that can contend with the large state space of a parallel program and can make the specification of such meaningful states easy. State specifications in terms of local predicates are ad-hoc at best. How are the relevant predicates to be determined? Is there a methodology that enumerates the predicates in a hierarchical order that allows the user to explore different facets of the program behavior in a systematic way? The answer to both of these concerns lie in hierarchically modeling the program behavior – a facility that state-based debuggers do not provide. In the next section, we will discuss event-based approaches that provide such a scheme of behavioral modeling.

2.5.3 Event Based Modeling Approaches

Modeling approaches attempt to ameliorate the problems of large amounts of trace data, by allowing the user to describe parts of the programs at a higher level of abstraction which is then matched with the actual execution for possible bugs. Good modeling approaches reduce the complexity by allowing hierarchical abstraction mechanism that handle ordering between events in a consistent fashion by filtering out the effects of asynchronous executions. Often in a program, a group of processes engage in similar activities which can be thought of as a single logical action: The user can group actions performed by one or more processes, specifying program behavior in terms of these abstract actions. We have already seen such modeling approaches used in the state-based techniques as a mechanism to set a global breakpoint. Event-based approaches can be used with run-time monitoring or post-mortem modeling. They differ most significantly in their modeling languages: IDD [36], and TSL [37] use temporal logic based modeling language, YODA [54], HARD [23], and TQuel [83] use relational query language, whereas ECSP [8], Kraut [13], EBBA [9], and Meld [41] use variants of regular expressions as their modeling language. Independent of their modeling languages event-based approaches can be compared in terms of their internal representation of behavior: string representations impose a total order on event occurrences while partial order, graph representations based on Lamport's *happened before* relation do not. Those using the total order cannot model *true concurrency* and their matching algorithms are susceptible to a change in relative process speed. We will discuss these issues in more detail in Chapter 3.

IDD [36] by Harter, Heimbigner and King, and TSL [37] by Luckham are both event-based run-time monitors that impose a total order on event occurrences. In IDD, a programmer uses interval logic to specify assertions that should hold at intervals delimited by distinct events. In TSL, designed for debugging Ada programs, specifications are written as a declarative part of the task bodies, and are enabled any time a task enters the region. YODA [54], HARD [23], and TQuel [83] use relational query languages to verify properties of execution behavior. These methods can be used only with shared bus architectures, where the shared broadcast medium imposes a total order among the inter-process interactions. Even then, most of the event interleavings can be the artifacts of the physical medium (the shared bus) and may not correspond to the logical view of the programmer.

An important event-based modeling language based on the an internal string representation of program behavior is EDL developed by Bates [9] as part of his event-based behavioral abstraction (EBBA) toolkit. EBBA uses a constrained event formalism. Models are described in a regular expression based syntax used to specify temporal constraints. A filter associated with each model specifies the constraints on the attributes of the constituent events. This filtering based on event attributes makes EBBA's modeling language quite powerful. However, due to the total order imposed on event occurrences in its internal behavior representation, EBBA has to model parallelism with an interleaving semantics of the *shuffle* operator, and does not model *true concurrency.* Moreover EBBA's pattern matching approach is too unrestrained and, as a result, it is incapable of providing precise feedback on errors.

Debuggers that retain the partial order between event occurrences in their internal representation of program behavior include ECSP [8] by Baiardi, De Nicola, and Vaglini, Meld [41] by Hseush and Kaiser, Belvedere by Hough [39]. ECSP specifies the program behavior as a set of process-specific models. A specification acts as a trigger for assertion checking. Once the specification is matched certain user-defined assertions can be checked, or mode control can be transferred to the user. As with any run-time monitoring system without a replay facility, this approach suffers from the probe effect, and cannot guarantee reproducible executions.

Variants of path expressions are used to specify the *control flow* of a program by Garcia and Berman [31], by Bruegge in his KRAUT debugger [13], by Hseush and Kaiser in their Meld debugger[41]. Garcia and Berman designed a visualization system that renders the Petri-net equivalent of the path expression specified by the user. The Petri-net is animated to show the program execution, and can be used to set a breakpoint. Generalized path expressions extend path expressions with predicates on the components of the path expressions, and by allowing arbitrary block of statements to be a path function. They also allow assignment operations to local variables when a given path is executed. KRAUT uses generalized path expressions as a basis for its modeling. This introduces the ability to modify a program state based on the control flow. Hseush and Kaiser introduce an extension of generalized path expressions – *Data Path Expressions* – that allow data events to appear in the path expressions. Data Path Expressions provide hierarchical specifications of behavior, and can model true concurrency since they operate on the partial order graph of the execution trace. Data path expressions have some limitations, however: they require all of the relationships between events in the behavior model be known at compile time and their implementation can be exponential in the size of the regular expression. Moreover their extension of temporal relationships to abstract events is too strict, requiring a total order to be present among all the constituent events of the temporally ordered higher level events. This rigidity often hinders the specification of complex behavior where the user is interested only in the temporal order between parts of abstract events and does not really care about a total order.

Event-based modeling complements the state-based debugging approach by providing support for behavioral abstraction. Some of them [39, 41] also contain the correct notion of logical time to accommodate asynchronous execution. None, however, have tackled the issue of providing hierarchical error feedback in presence of massive parallelism. Moreover in the process of avoiding issue of state consistency event-based debuggers eliminate any considerations of local states. It is therefore extremely difficult for them to track an error back to the source code constructs.

2.6 Conclusions

In order to detect a bug in a parallel program, the user must navigate the huge state space produced as a result of the multiple loci of control. The vastness of such state space coupled with the presence of physical asynchrony renders the traditional state-based debugging techniques useless. Parallel program debugging requires a sophisticated abstraction mechanism, and the ability to investigate behavior from different view points in a manageable way. Mellor-Crummey, LeBlanc and Fowler in Moviola [53] and Lin and Leblanc in their Clouds [55] debugger argue for such abstraction facility. The use of sophisticated abstraction in itself, however, does not provide a comprehensive solution. It must be accompanied by a facility to track the error detected at a higher level of abstraction down to the source construct. To be effective, the user should be able to narrow his/her focus to the offending part of the program behavior, set a globally consistent breakpoint and examine the program state to an arbitrary level of detail. As we will see later in this dissertation, Ave/Ariadne introduces such a comprehensive solution.

Chapter 3

Ave/Ariadne: Introduction and Concepts

In this chapter we introduce the Ave/Ariadne[1] event-based debugger [22]. Ariadne has a simple modeling language; it can be parsed with collections of finite state machines that provide precise information on the syntactic reasons for a match failure. This simplicity comes at the expense of some expressivity and thus, Ariadne's models are often too coarse, matching more than the intended behaviors. To compensate, the debugger supports functional queries that return characteristics of matched and partially matched behaviors, allowing the user to thoroughly investigate a program's behavior. The Ariadne Visualization Engine (Ave) is a scalable graphical interface for visualizing abstract events that features hierarchical visualizations reflecting the structure of user-defined behavioral models, incremental, dynamic attribute calculation, and automatic partitioning of matched behaviors and attributes.

In Section 3.1 we give an overview of event-based modeling. Section 3.2 introduces our modeling language, its structural hierarchy, and its use of logical time. Section 3.3 discusses the scalable visual feedback scheme, Ave, and its spreadsheet based query interface. Section 3.4 introduces our matching strategy, and Section 3.5 compares our modeling language with others.

[1] In Greek mythology, Ariadne provided Theseus with the thread that enabled him to find his way through the Labyrinth to slay the Minotaur. Ave stands for Ariadne's Visualization Engine.

3.1 Overview of Event-based Behavioral Abstraction

A model of program behavior is an abstract representation of the expected execution. Since the actual execution is extremely complex, the model specifies only the salient features of interest. A model of program behavior specifies a set of events in terms of constraints on their attributes and extant temporal relationships. The basic set of events are known as primitive events, each corresponding to a particular point in the execution of a single process. A primitive event can have many attributes. For example, a primitive event `x_entry` that is generated every time a procedure `X` is called might have as attributes the arguments to the called procedure, the name of the callee, and a local timestamp. A set of primitive events grouped according to some relationship between their attributes is known as an abstract event. The user can form higher level abstract events by grouping abstract events in the same manner. Event-based debuggers typically provide abstraction facilities, and facilities to verify that the actual execution behavior matches the defined abstractions.

3.1.1 Primitive and Abstract Events

Consider the case of an Automatic Teller Machine where each deposit and withdrawal produces either a primitive event `deposit` or `withdrawal` with the attributes account number, name, amount, time, location, and check number. One can then form an abstract event corresponding to the deposits made by John Smith with an account number of 123456789 on teller machine located at `Nowhere Street` on the day of `Big Bang`

```
Deposit_John = deposit*
      where
      person(deposit) == John Smith &&
      account(deposit) == 123456789 &&
      location(deposit) == Nowhere &&
      time(deposit) == BigBang
```

Here the `Deposit_John` abstract event has been formed by grouping the set of `deposit` events that meet the criteria defined in the `where` clause. In this case, the criteria for selection of primitive events depend only on the local attributes of the events. Often the criteria may be based on a relation between two or more primitive events. For instance, had Mr. John Smith purchased a certified check deposited by Ms. Jane Doe; the bank would like to group those transactions together to check their integrity. The abstract event definition can be written as:

```
Certified_Check = withdrawal deposit
      where
      person(withdrawal) == John Smith &&
      person(deposit) == Jane Doe &&
      amount(deposit) == amount(withdrawal) &&
      checkid(deposit) == checkid(withdrawal)
```

Irrespective of the mechanism used to group primitive events into higher level abstract events, event-based modeling scheme makes it easy to contend with large and complex behaviors by viewing them through interactions between abstract events, filtering out low level details during the initial phases of debugging.

3.1.2 Basic Components of an Event-Based Debugger

An event-based debugger has to provide two facilities: an instrumentation facility to generate the relevant primitive events during program execution and a modeling language to use in specifying the model of program behavior.

There are a variety of ways to generate primitive events. Many commercial debuggers allow the user to attach a watch point to a variable that prints out information every time the variable is defined; most of these facilities are quite primitive. Events can also be generated by pre-instrumenting a library. As an example, the communication library in a parallel language extension (e.g., PVM or the Intel message passing library) can be instrumented to generate primitive events corresponding to send and receive of messages. To be effective, a debugger must have an extensible set of primitive events, an easy way to group such primitive events into abstract events, and a scalable feedback mechanism that retains the ability to provide precise feedback on match failures.

The effectiveness of a modeling language depends on how easy it is to describe complex behavior and how precise its feedback is on match failure. The two services are interdependent: the more expressive the modeling language, the less detailed the feedback. In Ariadne, we have made the modeling language simple in order to make its feedback detailed and precise. We will see later in this chapter that the coarseness of the models defined by our language can be refined with a visual query interface to complex a great variety of complex behaviors.

3.2 Ariadne: The Modeling Component

Ariadne models a parallel computation as a set of asynchronous processes that interact through explicit read and write operations. Each process has a unique identifier (pid) and each executes a sequence of atomic, primitive events. Events can be of type *read*, *write* or *user-defined*. An execution is a directed acyclic graph in which vertices denote events and there is a directed edge from vertex u to vertex v iff *(i)* u *and* v *are executed by the same process and the occurrence of u immediately precedes the occurrence of v in the program order, or*

(ii) u *is the transmission of a message and v is its receipt.*

This graph is called a *time diagram*, and an example is shown in Figure 3.1. The transitive closure of a time diagram represents Lamport's *happened before* relation [48], also known as the *causality relation* [59].

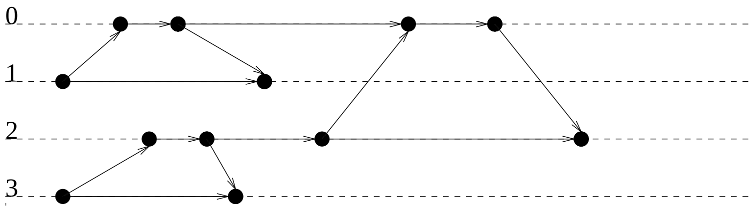


Figure 3.1. A sample time diagram with four processes

The matching semantics employed by Ariadne is that of parsing: each input event must be matched by the model, no skipping of events is allowed unless explicitly specified by the user. We use a simple user-directed filtering mechanism, and a sophisticated query interface to temper the rigidity of a pure parsing approach, without losing the ability to uniquely identify errors. These are discussed later in the chapter.

3.2.1 Modeling Language: Primitive Events

Ariadne has four pre-defined primitive events: `W` signifying the send of a message, `M` signifying the send of a multicast message, `R` signifying receive of a message, and `U` signifying any user-defined internal event. Each primitive event in Ariadne has five predefined attributes: *event_type* that corresponds to either of the four defined types; *pid* identifying the process where the event occurred; *oid* identifying any other process that may be relevant to the event; *utag* capturing the user-provided identifier for the event; and *ltime* storing the logical time. The user can extend the attribute list during instrumentation. For example, if process 1 sends a message to process 3 tagged as `query`, and the events are gathered as 5-tuples

$$< \mathit{pid},\ \mathit{oid},\ \mathit{event\_type},\ \mathit{utag},\ \mathit{ltime} >$$

then the message will generate two events

$$< \texttt{1, 3, W, "query", 1} >$$
$$< \texttt{3, 1, R, "query", 2} >$$

In a 4-process system, if process 0 sends a broadcast to all the other processes 1,2 and 3, it will produce 4 events

$$< \texttt{0, ALL, M, "query", 1} >$$
$$< \texttt{1, 0, R, "query", 2} >$$
$$< \texttt{2, 0, R, "query", 2} >$$
$$< \texttt{3, 0, R, "query", 2} >$$

Ariadne allows the user to extend the event-tuple dynamically. For example, the user can log the values of two local variables `x` and `y` every time a message is received, and add two new attributes x and y to each `R` event. This is a simple way to add state information to the event-based behavioral abstraction. In that case, the original message send will generate the following two events (assuming that the values of x and y are 7 and 10 respectively)

$$< \texttt{1, 3, W, "query", 1} >$$
$$< \texttt{3, 1, R, "query", 2, 7, 10} >$$

3.2.2 Modeling Language: Three-Tiered Structural Hierarchy

Ariadne's modeling language describes intended behaviors as subgraphs which are automatically "matched" against the recorded time diagram. To permit precise feedback on mismatches, we use a simple modeling language based on regular expressions. It has a three-tiered hierarchy of *chains*, *p-chains*, and *pt-chains*:

- *Chains* are patterns that represent "local views" of communication. They are described by extensions of regular expressions. When matched against a time-diagram, all events in the chain must occur exactly in the specified order. Ariadne provides a filtering mechanism to remove user-specified events. This compensates for the rigidity of the *parsing* approach, making pattern description somewhat more flexible while retaining feedback capabilities.

- *p-Chains* are patterns that represent the concurrent execution of a chain by one or more processes. They are described by binding a chain to a process set. When a p-chain is matched, the chain is matched on each element of the associated process set. The chains matching on different processes can interleave but they cannot share events among themselves.

- *pt-Chains* are patterns that represent the logical, temporal composition of p-chains and they are matched in two steps: first the individual p-chains are matched and then the asserted temporal relations between those events are verified. At the initial stages of debugging errors are manifested in predominantly two ways: through extraneous or missing events and through incorrect causal relationships between events. The decomposition of the matching algorithm allows us to distinguish between these classes.

The three definitional levels appear to form a natural mechanism for describing parallel systems, as evidenced by their use in other contexts such as the XYZ levels of Phase Abstractions [84] and the LaRCS specification language [56]. The matching algorithm for our language is straightforward: a pt-chain is recognized by a finite state machine that invokes copies of other finite state machines to recognize chains on specific processes. This matching can be done efficiently — in most cases in linear time, although the verification of the logical time relationships between abstract events may take polynomial time — avoiding the costliness of pattern matching approaches [9, 37] and the expensive implementations (number of states can potentially be exponential in the length of the parallel expression to be recognized) of previous languages [41]. At the same time, our matching algorithm can provide precise information on the reasons for a match failure.

3.2.2.1 Example: Binary Image Compression

To illustrate our modeling language, we consider a parallel version of an image compression algorithm [72]. Because the same example is used throughout the thesis, we digress slightly here to discuss it in some detail.

The input to the algorithm is an image stored as a *bintree* as shown in Figure 3.2. The compression algorithm works up from the leaves, comparing siblings. Identical, uniformly colored siblings are removed and their single color is represented at the new leaf (their former parent) as shown in Figure 3.3.

The parallel version of the algorithm runs on an n processor, distributed memory computer. Entire subtrees of the initial bintree are distributed to the processes, each of which does a local compression and then participates in a global reduction to form the complete, compressed image.

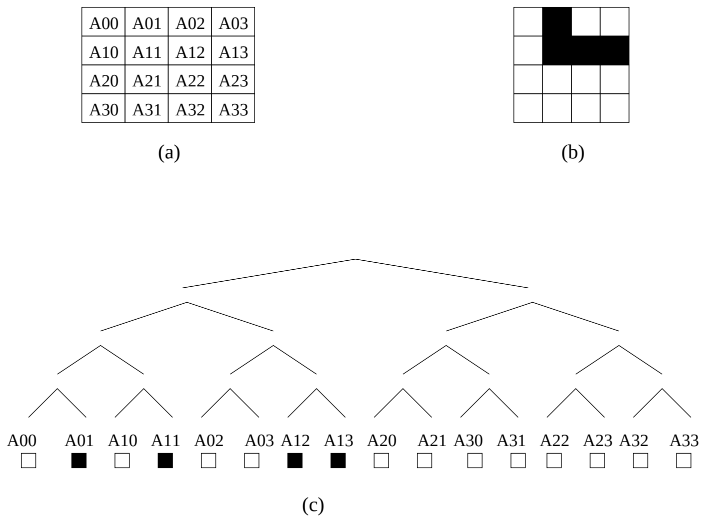


Figure 3.2. Binary Image Compression: Image Representation. Labeled 4×4 pixel array (a); corresponding image pixel array (b); and complete *bintree* representation obtained by alternatively splitting the image along y and x axes (c).

To model this behavior in Ariadne, we added user-defined attributes for the traced events[2]. Specifically, we included *message tags* to distinguish application-specific functions of the read and write operations. Generic reads and writes are denoted `R` and `W`. In our trace, they were each tagged with one of three possibilities: `query` indicating the initiation of a communication; `fail` indicating an unsuccessful compression of two siblings; and `succ` indicating a successful compression of two siblings.

Using the tags, we modeled the two possible outcomes of a sibling communication with

```
ch compr_fail = W_query @ R_query W_fail @ R_fail;
ch compr_succ = W_query @ R_query W_succ @ R_succ;
```

[2]At the moment, we are using hand instrumentation to provide user-defined events and/or attributes.

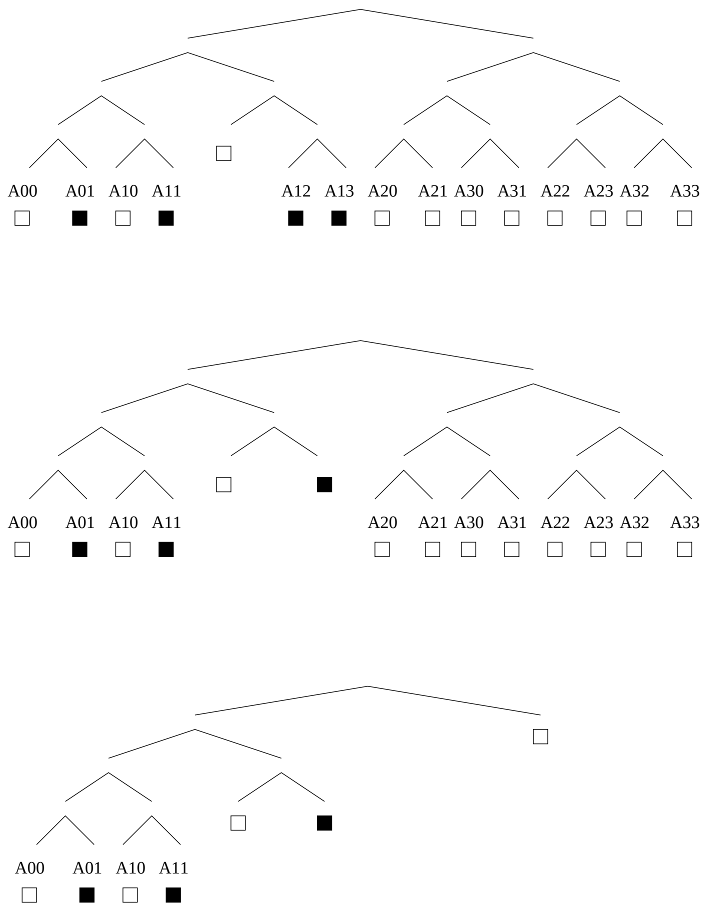
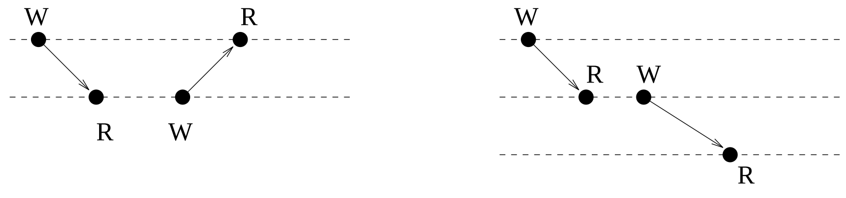


Figure 3.3. Binary Image Compression: Algorithm in action.

where **ch** is a keyword that begins the definition of a chain. Here, each chain contains a query message followed by a response message. Messages are modeled as a send (denoted by `W`) connected by an edge (denoted by `@`) to the corresponding receive (denoted by `R`). In both chains, the specified pattern of messages can match the intended communication pattern as shown in Figure 3.4 (a) but also the unintended behavior shown in Figure 3.4 (b). This lack of precision is consequence of the simplicity of our modeling language: a regular expression-based language permits inexpensive matching with accurate feedback but it does so at the expense of a loss in expressivity. Ariadne models are often too coarse. To compensate, we provide a query facility that enables the user to explore matched subgraphs as explained in Section 3.3. Here we ignore, for the moment, the possibility of unintended matches and proceed with developing the full model.



(a) (b)

Figure 3.4. Coarseness of chain level description. Intended behavioral match (a); unintended behavioral match (b).

From the perspective of an initiating process, a comparison can either fail or succeed and can be specified as

```
ch compr = compr_fail or compr_succ
```

On each iteration of the reduction, comparisons are performed concurrently on a number of processes, which we modeled as

```
set procset = {0..3};
pch compr = ( compr_fail or compr_succ ) onsome procset;
```

where **set**, **pch**, and **onsome** are again keywords. **set** indicates a set definition (in this case, the relevant set of pids), **pch** indicates a p-chain definition, and **onsome** indicates that the match need occur on some, but not necessarily all, of the specified processes. As the reduction proceeds, each step involves fewer processes; **onsome** allows us to use the same model across all steps.

The behavior of the algorithm as a whole is modeled as

```
match compression = compr* ;
```

where **match** is a keyword, indicating that 0 or more of the `compr` patterns are to be matched in the time diagram and the resulting subgraph is to be assigned to the variable `compression`. Figure 3.5 shows a match of this model to the time diagram of Figure 3.1.

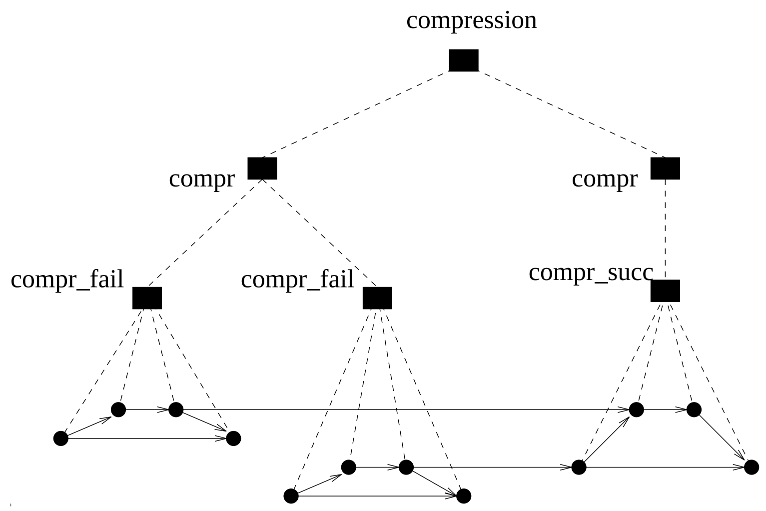


Figure 3.5. Match tree representation. The match of a user model the time diagram of Figure 3.1; components of chains, p-chains and pt-chains are connected by dashed edges.

The matching imposes a tree structure, called a *match tree*, on the diagram that corresponds to the model's three-tiered structure: internal nodes represent chains, p-chains, and pt-chains while the leaves represent primitive events.

This tree is called the *match tree*, and should not be confused with the tree representation of the image that we discussed in the Binary Image Compression example. The levels of the match tree correspond to the hierarchical levels of the model definition and the root corresponds to the outermost level of pt-chains.

All nodes inherit the name provided by the user in the model definition (here *compression*, *compr*, *compr_succ*, and *compr_fail*). By selecting any internal node of the match tree, the user can narrow his/her attention to a portion of the execution history matched, and submit queries regarding the selected portion of the execution history. Selecting the root selects the whole matched behavior.

3.2.3 Modeling Language: Logical Time Manipulation

Programmers are often confused by the results of asynchronous executions because they do not foresee all possible interleavings of events. In fact, most of these interleavings are arbitrary and irrelevant: program correctness depends only on the event orderings imposed by data dependencies. Thus, we are interested in the ordering of events not in physical time but in *logical time*. At the primitive event level, this ordering is captured by Lamport's *happened before* relation. Ariadne imports Belvedere's extension of Lamport's relation to sets of primitive events; defining three relations: **precedes**, **parallels**, and **overlaps** [40] for use in the construction of pt-chains. Informally, if A and B are sets of primitive events, then

A **precedes** B (denoted $A \rightarrow B$) iff there is some dependency from an event in A to an event in B but no dependency from an event in B to an event in A,

A **parallels** B (denoted $A||B$) iff there is no dependency from any event in A to an event in B and no dependency from any event in B to an event in A, and

A **overlaps** B (denoted $A \leftrightarrow B$) iff there is both a dependency from an event in A to an event in B and a dependency from an event in B to an event in A.

Similar extensions have previously been proposed [26, 35, 41] but we have found our definitions to be more appropriate for debugging. In particular, the **precedes** relation captures the notion that just some part of a complex event *happens before* some part of another event; other definitions require a total ordering which is rarely found among the higher-level abstract events in the programs we have examined.

In the Binary Image Compression example, each communication step strictly precedes the next. This behavior, for a 32 process system, can be described as

```
set procset = {0..31};
ch compr_fail = W_query @ R_query W_fail @ R_fail;
ch compr_succ = W_query @ R_query W_succ @ R_succ;
pch compr = (compr_fail or compr_succ) onsome procset;
match compression = < compr* >;
```

Figure 3.6. Refined Ariadne model of binary image compression.

where the angular brackets – < > – around `compr*` in the last statement indicate that *precedes* relation holds between successive iterations of the communication phase.

The model now describes the complete behavior of the program but only for four processes. To increase that number, we need only change the definition of `procset`. Even a modest increase, however, brings us to the next issue for parallel debuggers: scalable output. Part of the match tree produced by a 32 process system for this pattern is shown in Figure 3.7. Clearly this is not a scalable representation.

Our introduction has highlighted the following features of Ariadne's modeling language.

- **Flexible hierarchical abstraction.** The three definitional levels of Ariadne allow the user to specify parallel program behavior hierarchically. The flexibility of the language is evident in at least three mechanisms: the **onsome** clause at the *p-chain* level that allows a subset of the specified processes to

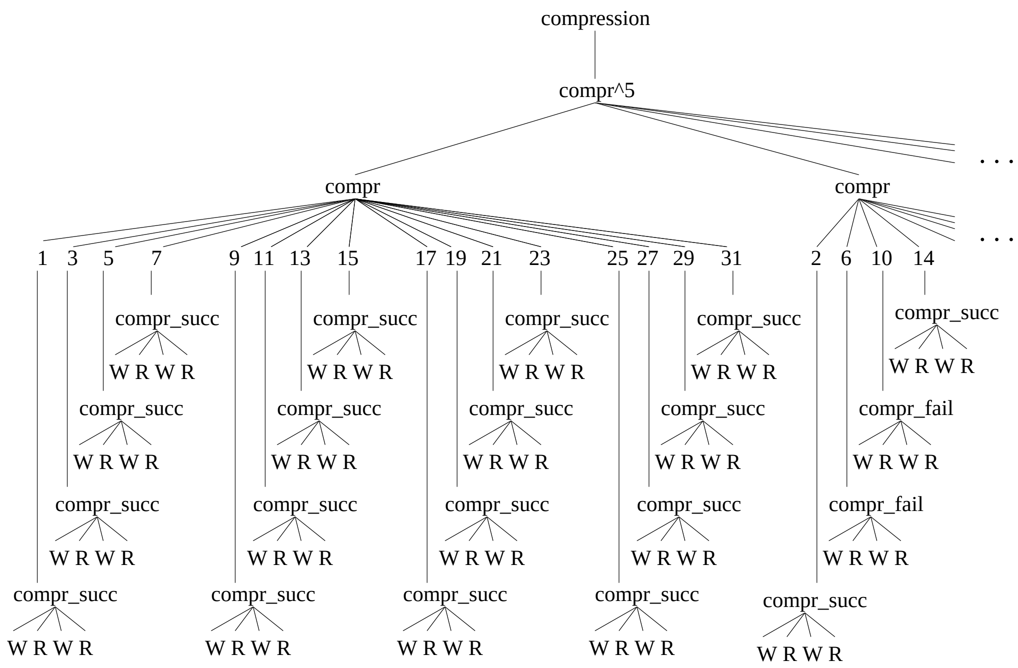


Figure 3.7. Non-scalable visualization of match tree. (Note we have dropped tags on W and R events for display purposes.)

take part in a behavior; the interleaving semantics between individual *chains* matching on different processes as part of a single *p-chain*; and the `precedes` relationship at the *pt-chain* level that allows only a part of an abstract event to precede another.

- **Scalable modeling.** By separating an abstract behavior and the manifestation of such behavior on a set of processes, Ariadne provides the facility to specify scalable models of program behavior. If the number of processes in the computation changes, only the process set definitions at the p-chain level need to change.

- **Precise feedback on error.** The parsing interpretation allows Ariadne to detect the first occurrence of a mismatch between the user specified model and the program behavior. The two-phased implementation of the matching algorithm (events are matched before the temporal relationships are verified) allows Ariadne to differentiate between errors manifested through extraneous or missing events, and those through incorrect causal relationships between events.

Ariadne's modeling language, however, does not solve all of the problems in event-based modeling of massively parallel computations. In particular, we have already encountered two problems in our simple examples:

i. Ariadne models often provide only a coarse description of program behavior that may match an execution in unintended ways, and

ii. Ariadne's output is not sufficiently scalable.

both of these are addressed in the design of the Ave[47] user interface.

3.3 Ave: Scalable, Visual Feedback Mechanism

Figure 3.8 shows Ave's depiction[3] of the 32 process match tree from Figure 3.7.

The tree has been compressed both horizontally and vertically: horizontally by eliding sibling nodes and vertically by pruning levels. The horizontal compression is straightforward: suppressed siblings are represented by a single node with button that can be used for scrolling; an integer label gives the number of hidden constituents. In our example, the match tree included 5 instances of the `compr` p-chain but only three are shown with those remaining represented by the node labeled 2.

[3]All debugger screen outputs in this document are redrawn with xfig for clarity and to accommodate the reproduction limitations. The actual debugger screen dumps are similar and use colors to highlight different conditions.

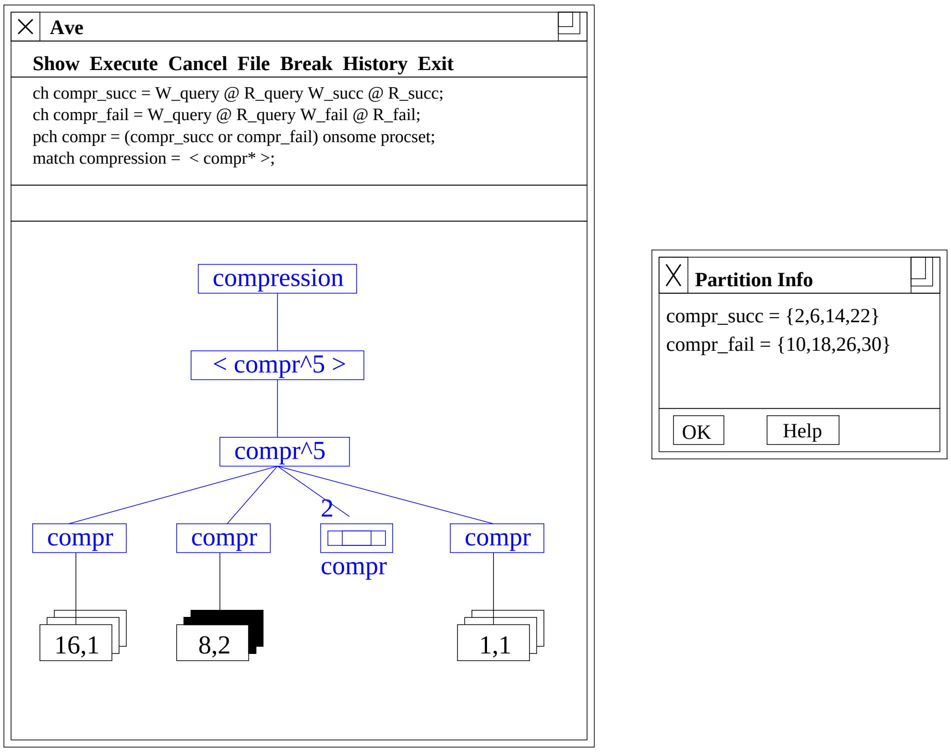


Figure 3.8. Ave feedback showing abstracted match tree for binary image compression example.

Vertically, the tree is compressed by pruning subtrees below each p-chain. The pruned subtrees are partitioned into equivalence classes and represented by an icon of stacked rectangles labeled with a pair of integers that summarizes the partitioning: the first element is the total number of subtrees represented and the second is the number of equivalence classes. In our example, the leftmost dashed box is labeled with **16,1** which indicates that 16 instances of the `compr` chain were found and that all of them were identical (that is, they matched the same p-chain pattern). This correctly describes the first step of our reduction in which all 32 processes happened to be paired in successful comparisons. The icon with summarizing label **8,2** represents the second step in which eight pairs of processes participated and they had two distinct behaviors, some matching the pattern `compr_succ` and others matching **compr_fail**. To get further information, we clicked on the **8,2** icon to get the explanation shown in the small side window: processes 2, 6, 14, and 22 executed **compr_succ** while processes 10, 18, 26 and 30 executed **compr_fail**.

Figure 3.8, shows Ariadne's feedback on a correct version of the binary compression algorithm. Unfortunately, our first attempt at the program was not correct. It produced the output shown in Figure 3.9 with the detected error highlighted with dashed boxes (in the actual debugger errors are highlighted with red rectangles). Note that extra events are shown to make the error visible despite scaling.

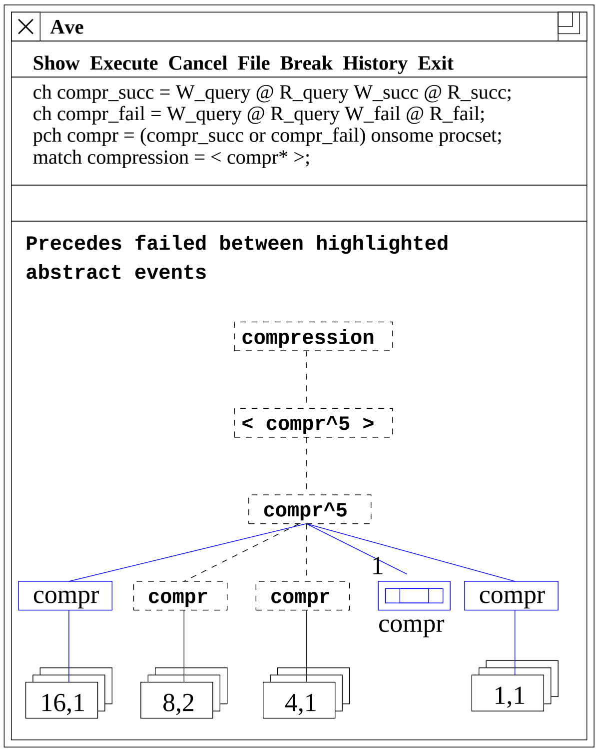


Figure 3.9. Ave feedback showing error in binary image compression.

This figure shows that the p-chain **compr** occurred 5 times but that the **precedes** relation did not hold between the second and third instances. It does not explain why the relation did not hold: Were the processes correctly paired?

Were the two instances executed on disjoint sets of processes (and thus in parallel) or were there overlapping dependencies? Was there proper synchronization between iterations?

These uncertainties are results of the initial coarse grained modeling. For instance, the model in Figure 3.6 does not specify explicitly the number of processors taking part in each iteration, and the chain description can match two behaviors: the expected one as shown in Figure 3.10(a), and the unintended one shown in Figure 3.10(b). This, and other assertions, can be verified by using Ave's query facilities.

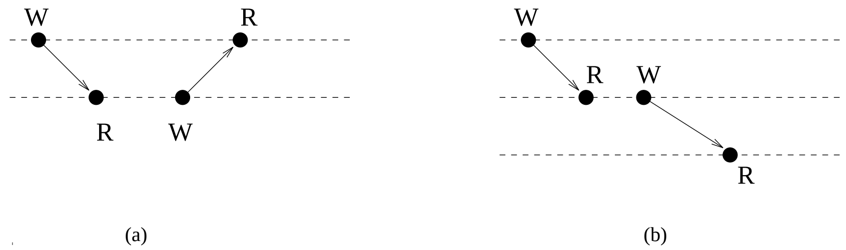


Figure 3.10. Depicting coarseness of Ariadne's model of exchange.

3.3.1 Spreadsheet Based Query/Assertion Facilities

Debugging with Ariadne typically proceeds in a top down manner. Initial coarse models are further refined until the error is located. The investigated models themselves are simple, reducing the possibility of a modeling error, and allowing for better characterization of mismatches. It is very unlikely that all bugs would be so accommodating as to show up while matching the initial coarse grained model. Even the detection of error may not really illuminate the manner in which the program failed. For instance, from Figure 3.9, we only know that the second `compr` abstract event did not precede the third `compr` abstract event. However, we do not know how the relationship failed: did the two abstract events encompass disjoint sets of processes, or were they overlapping on some processes and not others? The first resolution will suggest some sort of communication error where processes are not computing their partners in the exchanges correctly, whereas the second resolution would suggest a synchronization error. To accommodate additional investigation Ariadne provides a spreadsheet based query facility. For this example, we need to ask the following four questions to eliminate the possibility of an erroneous communication pattern.

1. Did all exchange take place between paired processors?
2. Were the sets of processors taking part in successive iterations disjoint?
3. Were the sets of processors initiating an exchange in different iterations disjoint?
4. Did the number of processors initiating an exchange get halved at each iteration?

For a correct execution, the answer to the first question should be "Yes" since exchanges between paired processors is a pre-requisite for the integrity of the compressed image. The answer to the second question should be "No", since all the processes take part in the first communication phase, and the number gets halved in each subsequent iteration. The third question should return "Yes", since processes that initiate an exchange do not take part in any subsequent communication round. The fourth question is a verification of stricter condition, that indeed the communication phase follows a binary search pattern and should also return "Yes".

To facilitate such verification of assertions, Ave treats its match tree visualization as a spreadsheet interface. A flat two-dimensional spreadsheet contains active cells with data values, and it allows a new value to be computed for a destination cell by applying a transformation on the data of the selected cells. Cells are selected by by specifying their relative location by a coordinate transformation from the anchor cell. Ave presents a tree-structured spreadsheet, where each node corresponds to an abstract event, each containing a complex data structure. The data structure at each node can be augmented by new data values. The elements of the complex data structure are called *attributes* of the abstract event. As in the case of flat spreadsheets, in order to compute a new attribute based on attributes of other nodes, the user needs to select the relevant source nodes. These nodes are selected by giving their relationship (*parent, child, sibling, descendant*, and *ancestor*) with the anchor node. The more general relationship *descendant* and *ancestor* can be used to identify a single or a set of nodes uniquely.

A spreadsheet interface for performance debugging has been proposed and implemented by Sarukkai and Gannon [81]. However, instead of process interaction, they use the static control flow structure to achieve coarse grained filtering. Events are selected by walking down the static control flow graph, and then issuing temporal queries on a historical database. Ave uses the user-defined abstraction to handle the multi-dimensional temporal aspect, and the resulting match tree to select relevant events. Ave thus provides a closer coupling between the dynamic program behavior and the user defined abstractions.

Example: Binary Image Compression (Continued) In an attempt to characterize the error highlighted by Ave display, we verified each of the four assertions above. Ariadne provides an intrinsic name *proc* to the tree nodes that correspond to the processes where a chain description has been matched. All these nodes can be referenced as a group with the identifier *proc*, and individual nodes can be referenced with their process identifiers. In order to verify that exchanges took place among paired processors, we need to check whether the process-identifier of the second `R` event in each chain is the same as the first `W` event. This can be done with

```
with compression foreach proc
  compute diff = eq(pid(first(leaf(proc))), pid(last(leaf(proc))));
with compression show diff(proc) wrt compr;
```

All of Ariadne's pre-defined functions take lists as arguments. `first` and `last` return the first and the last member respectively of any list, `pid` returns the process-identifier of an event, and `eq` takes two lists as its arguments and returns 1 if the lists are equal. The attribute `diff` can take only two values: 0 and 1. For display, Ave partitions the abstract events into equivalence classes based on the attribute value and maps the equivalence classes to different colors (Note that Ave's initial visualization of the match tree also exhibits similar partitioning and color mapping: matched nodes are colored blue, nodes that indicate collapsing a p-chain behavior over a process set are colored green, and unmatched events are shown in red). Here, the computed attribute all had the same value 1, so only one color(black) was shown. We'd asked for the display `wrt compr` so the information is grouped into five components, one for each `compr` event. The plot shows 16 `diff` values of 1 for the first `compr` event, 8 for the second `compr` event, 4 for the third, 2 for the fourth, and one for the fifth.

For a larger version of the system, this display does not scale well. Even for 128 processors, it would not be very comprehensible. We can, however, combine computed the attributes further by calculating a single value for each `compr` event

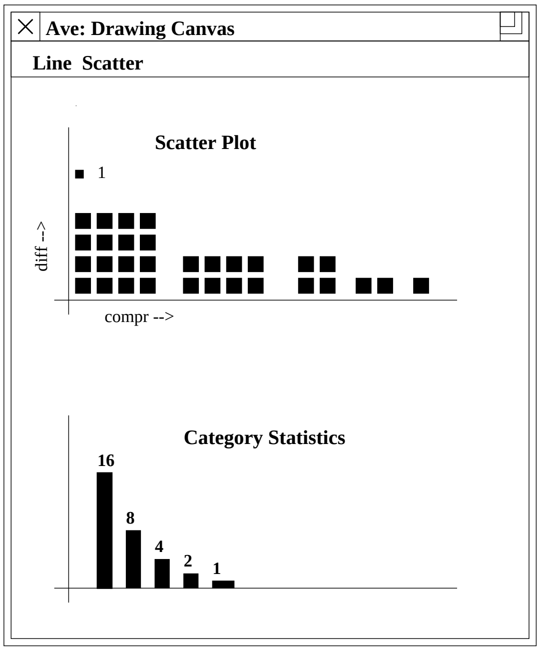


Figure 3.11. Scatter Plot showing exchanges occurred correctly between partners in Binary Image Compression.

as a whole. This was done with the additional query

```
with compression foreach compr compute idiff = eq(diff(proc), 0);
with compression show idiff(compr) wrt compression;
```

which produced the display shown in Figure 3.12. This display occupies much less space than that of Figure 3.11 but it is at the cost of some loss in precision. In case of error, it can only identify the *iteration* that contains the erroneous exchange – the user needs to investigate further to get a more detailed feedback. Typically there is a trade off between scale and precision [19]. Ave allows the user to control the balance so that debugging can begin with the most scalable display and then progress, as the focus of attention narrows, through increasing levels of detail.

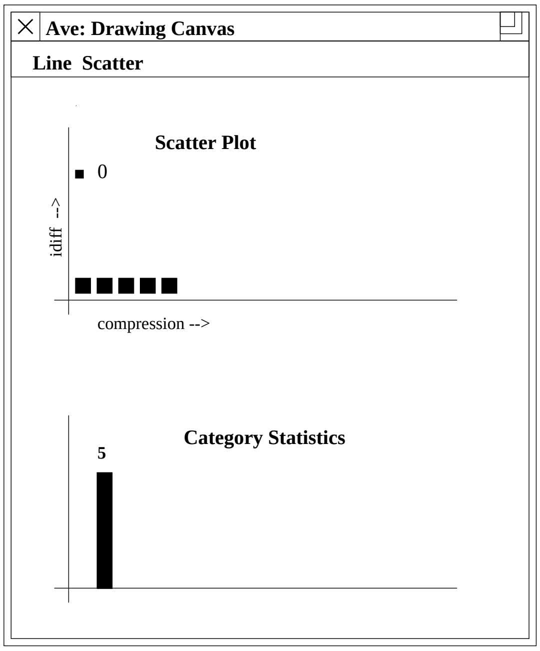


Figure 3.12. Scatter Plot showing summary of exchanges in Binary Image Compression.

The second assertion that the processors taking part in successive iterations are not disjoint can be verified by using the query

```
with compression foreach compr compute pset = pid(leaf(compr));
with compression foreach compr compute
  is_pset_disj = disjoint(pset(compr), pset(right(compr)));
with compression show is_pset_disj(compr) wrt compression;
```

where `disjoint` is a predicate indicating whether or not a list of sets are disjoint and `left` and `right` refer to neighboring siblings. The query resulted in the output shown in Figure 3.13 as expected. It shows that for each the first four `compr` events, the sets of processes active in that event overlapped overlapped with the set of processes active in the next event and thus `disjoint` was false as expected. In the last case, the figure shows `disjoint` as true but this is because there is no "next" `compr` event so that set is by default disjoint from the current one.

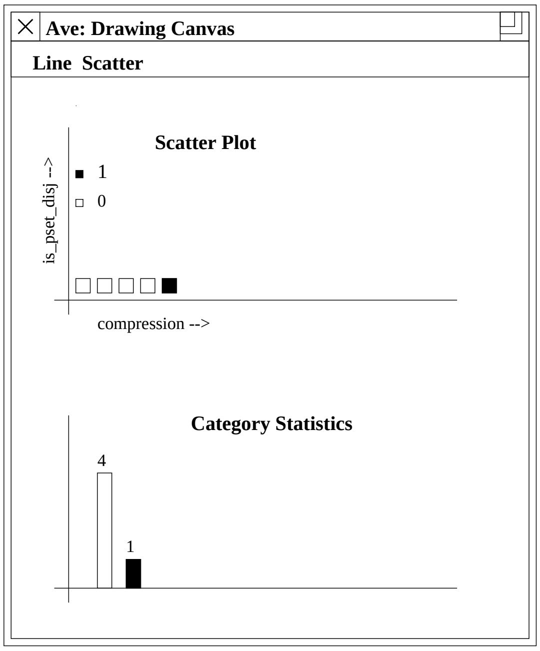


Figure 3.13. Scatter Plot comparing sets of active processes in the reduction steps of the Binary Image Compression.

The third assertion that disjoint sets of processors initiate exchanges in different iterations can be checked similarly, but this time only considering those abstract events that are named `proc`.

```
with compression foreach proc
  compute iset = pid(first(leaf(proc)));
with compression foreach compr compute iset = iset(proc);
with compression foreach compr
 compute is_iset_disj = disjoint(iset(compr), iset(right(compr)));
with compression show is_iset_disj(compr) wrt compression;
```

which resulted in the expected output shown in Figure 3.14 showing that sets of processors initiating exchanges in different iterations are indeed disjoint.

Finally, we checked the assertion that the number of processors initiating exchanges was halved at each iteration. Actually, this was unnecessary since we could have gotten that information from Figure 3.11. We use it here to demonstrate

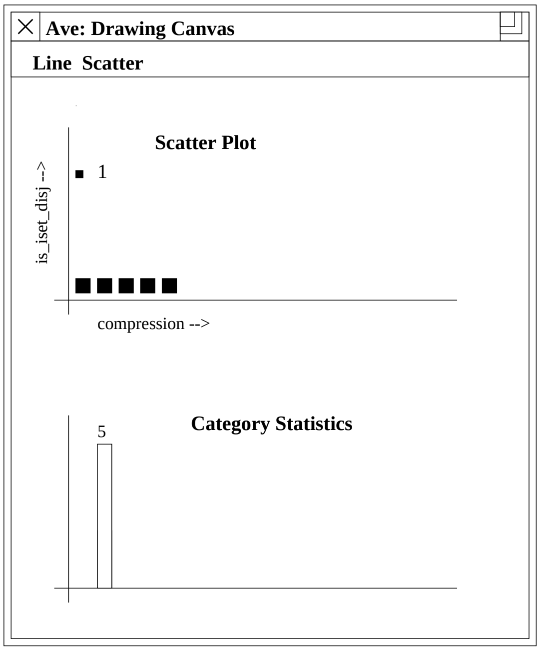


Figure 3.14. Scatter Plot showing sets of processors initiating an exchange in different iterations are disjoint.

another available plotting style. The query was

```
with compression foreach compr
    compute size = sizeof(iset(compr));
with compression show size(compr) wrt compr;
```

and the output is shown in Figure 3.15.

Having used these queries to investigate our match, we were convinced that the original coarse model was indeed correctly matched against the system's behavior. The detected **precedes** failure, then, was not an artifact of our modeling but a real error. At this point, we had considerably narrowed our focus of attention: we needed only to consider only two instances of the `compr` events and, since their communication patterns seemed correct, we should look for a synchronization error. Further use of event-based debugging, however, did not help to locate the error.

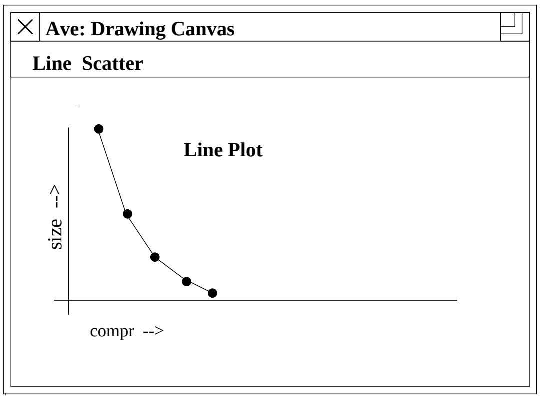


Figure 3.15. Ave line plot showing number of processors halved at each subsequent iteration.

This is typical: event-based debugging is useful in the initial stages of debugging but it does not help in the final stages of relating the observed errors back to source-code constructs. For that, we need to use state-based debugging, using the results of our analysis thus far to establish consistent, global breakpoints as described in Chapter 5.

Ave's capabilities can be summarized with the three services that it provides in conjunction with Ariadne.

- **Scalable hierarchical visual feedback.** Ave visualizes Ariadne's match trees to provide hierarchical feedback regarding the match between the model and the program behavior. It also allows the user to control the scale of the feedback by using the query language.
- **Automatic partitioning of behavior.** Ave partitions the processes at the p-chain level into equivalence classes based on the chain level behavior, thus illuminating the abstract behavior of a set of processes.

- **Dynamic attribute computation.** Ave's treatment of the match tree as a spreadsheet facilitates examination of the different facets of the match in terms of derived attributes of abstract events. Ave partitions the abstract events into different categories with respect to different attributes, and maps distinct categories to distinct colors[4].

In the following sections, we describe the matching semantics of Ariadne's modeling language, and compare Ariadne's modeling language with others.

3.4 Ariadne's Matching Algorithm

Ariadne separates the issues of the matching of an abstract event and the verification of its temporal relationships with other abstract events. Matching is done with one automaton for each chain level description; it is never necessary to glue automata together by taking their product as in the case of DPEs. Ariadne's models explicitly enumerate the set of processes where matching can start and thus avoid the need for product automata; the same simple automaton is used for each member of the process set. Our space complexity is thus additive in the size of the regular expression at the pt-chain level and the number of processes instead of the multiplicative complexity introduced by DPE.

At the conceptual level, chain expressions are variants of regular expressions with some extensions. Each chain expression is translated to a non-deterministic finite state automaton as shown in Figure 3.16. Matching a p-chain definition ensures that a copy of the chain is matched in each process identified in the p-chain expression.

[4]The color space is finite. So colors are recycled if the number of categories exceed the number of colors available. The user can select the number of colors used by Ave, and the particular colors.

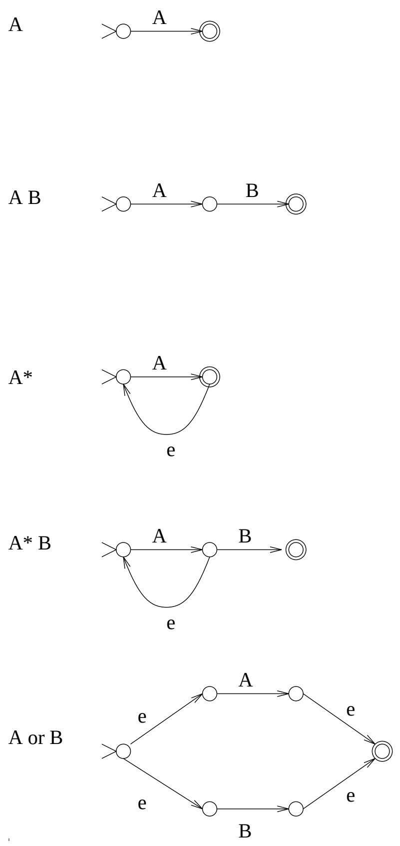


Figure 3.16. Converting Ariadne's chain expressions to non-deterministic finite automata.

Pattern matching conceptually proceeds in two steps: first the specified events are found in the event trace and then the specified temporal relations between those events are verified. Thus, for example, to describe a pattern in which two p-chains `a` and `b` are logically parallel, we use $|$[`a b`]$|$ which is matched as

find(a); **find**(b); **assert**(a$||$b)

This means that expressions on this level actually denote two kinds of "orderings". The first specifies the search order and the second specifies the required temporal ordering relations. The search order is linear and thus can easily be expressed using standard regular expressions. For example, we might have an expression like

$$(\texttt{a b c})* \ \texttt{d}$$

which matches as many occurrences of `a b c` as it can and then matches a single occurrence of `d`. The specification of the required logical orderings is "overlaid" on these regular expressions.

To express the required logical orderings, we need abstract events that are the collection of things found in a particular match. Our use of regular expression to describe behavior associates a "search order" with each set; thus, an abstract event is a list of events in search order; the empty list will be denoted by `nil`. (This will greatly simplify our expressions and provide for uniform treatment of `precedes` and `parallels`.) Does this extraneous ordering cause any problem? No, since abstract events that are related by the *precedes* relation need to be matched in some order anyway, and in case of *parallels* or *interleaving* the order is irrelevant.

We use different forms of left and right brackets to specify assertions of logical relations. The set braces used above — { and } — denote an abstract event without an asserted order (note: this does not mean unordered, it means "don't care"). Composite brackets — $\|[$ and $]\|$ — denote `parallels` and angular brackets — $<$ and $>$ — denote `precedes`. Thus

$$< \ \|[\texttt{a b}]\| \ \texttt{c} \ >$$

means

$$\texttt{find(a); find(b); find(c); assert}(\texttt{a} \parallel \texttt{b}, \ \{\texttt{a b}\} \rightarrow \ \texttt{c})$$

where the symbols used in the `assert` refers to the matched instance of the pt-chain. We have already seen the use of the `precedes` relationship. Further examples are shown in Figure 3.17.

Expression	Interpretation
$\{$a b$\}$	find(a); find(b); assert()
$<$ a b $>$	find(a); find(b); assert(a $\rightarrow$ b)
$<$ a b c d $>$	find(a); find(b); find(c); find(d); assert(a $\rightarrow$ b, b $\rightarrow$ c, c $\rightarrow$ d)
$<$ $<$ a b $>$, c $>$	find(a); find(b); find(c); assert(a $\rightarrow$ b, $\{$a, b$\}$ $\rightarrow$ c)
$\|$[$\{$a b c$\}$ d]$\|$	find(a); find(b); find(c); find(d); assert($\{$a, b, c$\}\|\|$d)
$\|$[a b c]$\|$	find(a); find(b); find(c); assert(a$\|\|$b, b$\|\|$c, a$\|\|$c)
$<$ $\|$[a b]$\|$ c $>$	find(a); find(b); find(c); assert(a$\|\|$b, $\{$a, b$\}$ $\rightarrow$ c)

Figure 3.17. Examples showing interpretations of logical orderings.

The expression:

$$y \;=\; < \;\; \mathtt{a}\;\mathtt{b}\; || \;\{\mathtt{c},\mathtt{d}\},\; <\mathtt{e}\;\mathtt{f}\;\mathtt{g}> \;\; || \;\; >$$

defines an abstract event y composed of abstract events abstract events {a}, {b}, {c}, {d}, {e}, {f}, {g}, {c, d}, {e, f, g}, {c, d, e, f, g}, and {a, b, c, d, e, f, g}, meeting the following requirements:

$$\begin{array}{l} \mathtt{e} \rightarrow \mathtt{f} \\ \mathtt{f} \rightarrow \mathtt{g} \\ \{\mathtt{c},\mathtt{d}\}||\{\mathtt{e},\mathtt{f},\mathtt{g}\} \\ \mathtt{a} \rightarrow \mathtt{b} \\ \mathtt{b} \rightarrow \{\mathtt{c},\mathtt{d},\mathtt{e},\mathtt{f},\mathtt{g}\} \end{array}$$

For more complicated patterns, we have to consider the possibility of `nil` results. Thus, we can now write patterns like $<$ a b$*$ c $>$ which should succeed in the event that the a is matched once, b not at all and c once. The b$*$ thus "successfully" matches the `nil` abstract event. How should this match be treated during the assertion of `precedes`? Allowing `nil` events to occur in the assertion produces an unappealing extension to our precedes relationship that dictates, for all events x,

$$nil \;\rightarrow\; x \;\text{ and }\; x \;\rightarrow\; nil$$

We therefore strip `nil` matches from the sequence before asserting temporal relationships. Thus in the event that `b` matches 0 times, $<$ `a b`$*$ `c` $>$ initially finds three events but the middle `nil` event is stripped and thus ignored by the precedes assertion. What if only `nil` is left? The event is stripped automatically, and the relation holds true trivially.

In the next section we compare Ariadne's modeling language with three other languages — EDL [9], DPE [41], and PEDL [39] — to highlight the differences.

3.5 Evaluation of Ariadne's Modeling Language

Hough in his thesis [39] presents a detailed critique of Event Definition Language (EDL) [9] and Data Path Expressions (DPE) [41] and introduces a new modeling language called Parallel Event Definition Language (PEDL). Since these three languages are most similar to ours among the event-based modeling languages, we will evaluate them and present the rationale behind Ariadne's modeling language. Our critique of EDL and DPE will borrow heavily from Hough's comprehensive treatment, as will be evident from the Figures 3.18 and 3.19 which are very similar to those used in his thesis.

EDL uses a string representation of program behavior which imposes an arbitrary total order on events that are unordered. For example, the behavior shown in Figure 3.18(a) and Figure 3.18(b) have different temporal relationships between events W and R; both, however, have the same string based representation.

Modeling languages that use strings to represent program behavior internally make their matching sensitive to changes in relative process speeds. The model "W followed by R" in Figure 3.19 matches different events depending on the relative speed of processes 1 and 2.

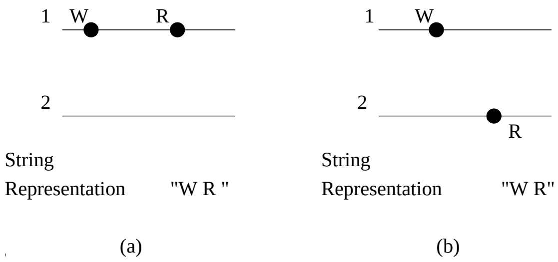


Figure 3.18. Lack of precision of temporal relationships in string representation of program behavior.

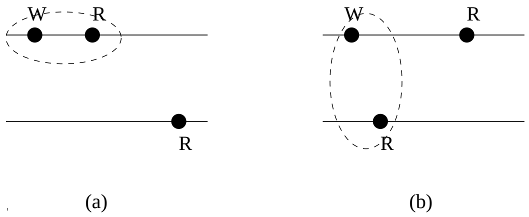


Figure 3.19. Speed sensitive matching algorithm. Matching of a model "W followed by R" with a program behavior of two different executions of the same program (a) and (b). The match is shown with a dashed oval.

The string based representation imposes a third limitation on EDL: It forces EDL to use an interleaving semantics to describe concurrency. EDL's *shuffle* operator (denoted by) matches concurrent events a and b by accepting the events in either order. Such a concurrency semantics is extremely weak. Figure 3.20 shows two possible matching of the model "W parallel with R": in (a) the events `W` and `R` are unrelated, while in (b) they are the send and receive of the same message. The matching in Figure 3.20(b) is wrong, the events can never occur concurrently; but a notion of concurrency based on the interleaving semantics can do no better.

To ensure that EDL models are insensitive to process speeds, one needs to introduce constraints based on process identifiers. For instance, EDL can ensure that it always matches Figure 3.19(a) by specifying that the W and the R events

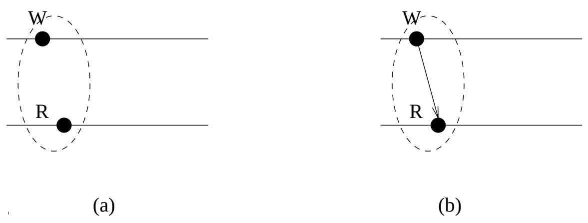


Figure 3.20. Speed sensitive matching of *shuffle* operator due to its interleaving semantics. Matching of a model "W parallel with W" with a program behavior of two different executions of the same program (a) and (b). The match is shown with a dashed oval.

occur on the same process. EDL's weakness related to the interleaving semantics of its *shuffle* operator is, however, a more difficult task. Modeling true concurrency among n events in EDL requires specification of $\frac{n(n-1)}{2}$ constraints on process identifiers. Constraints using explicit process identifiers make the modeling non-scalable: Models need to be changed at all levels whenever the number of processes in the computation changes. It also impedes abstractio because concurrency between abstract events cannot be specified without referring to the process identifiers of their constituents.

EDL's second shortcoming is its use of pattern extraction as the semantics of matching. A model can be matched to a behavior in different ways. Baiardi *et. al.* [8] and Zave [96] require the model to match *all* input events: no skipping of events is allowed. This can be thought of as *parsing* approach. Another alternative [37, 9] is to use a *pattern extraction* approach where events are skipped as needed. A parsing approach can detect and characterize an error uniquely; a pattern extraction approach, in general, lacks this ability. The difference is illustrated in Figure 3.21.

The two approaches reflect extreme points of a spectrum. With parsing, an error can be uniquely identified but the user has to account for all low level events in the model description; the modeling interface is thus not flexible. With pattern

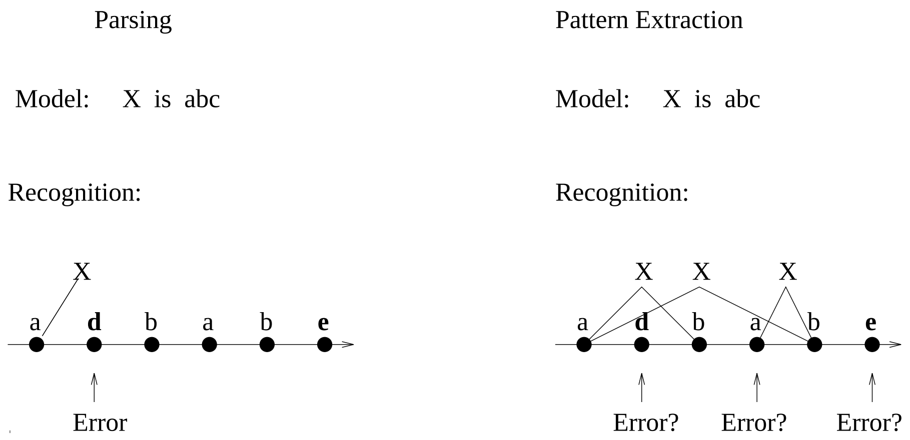


Figure 3.21. Parsing vs. Pattern Extraction

extraction, the user has a very flexible modeling interface, but the unrestrained filtering during matching restricts its feedback to a match/mismatch vocabulary. In Figure 3.21, the parsing approach can flag an error as soon as it encounters the first *d* event. No such unique error event exists with the pattern extraction interpretation. The lack of unique error state is because pattern matching approaches allow multiple finite state machines to be active simultaneously, each of which may have consumed different number of inputs, as shown in Figure 3.21.

DPE and PEDL circumvent the problems with asynchrony by using Lamport's *happened before* relationship to represent behavior, and employing a parsing approach toward matching. Modeling based on *happened before* relationship is less sensitive to relative speed of processes (asynchronous execution). For example, "W followed by R" will always match according to Figure 3.19(a), and "W parallel to R" will only match Figure 3.20(a).

DPE's implementation scheme is, however, not very scalable. Data Path Expressions are implemented in terms of finite state machines called *predecessor automata*. Small automata that recognize subexpressions are glued together to form a single automaton that recognize the behavioral model. This insistence on product automaton makes DPE unsuitable for modeling massively parallel programs because the implementation of the parallel operator uses a product of machines: to model n abstract events that are parallel, we need to find the product of the n automata that recognize the individual abstract events. The number of states in such an automaton will be the product of the number of states of the constituent machines, and the implementation is susceptible to state explosion.

We have already identified DPE's rigidity in defining the **precedes** relationship between abstract events. Two abstract events in DPE are sequential if and only if all primitive events making up the first precede all primitive events in the second. This total ordering among events are rarely visible in MIMD computation, and it makes the modeling language inflexible.

Hough's PEDL overcomes both the limitations of DPE by not composing product automaton and by defining a flexible **precedes** relationship between abstract events that allows only part of one abstract event to precede part of the other. This is the same definition that we have inherited in our modeling language.

Although both DPE and PEDL use use a time diagram similar to us capturing the *happened before* relationships between events, they both miss a subtle characteristic of the time diagram. A time diagram of a n process system is not a general graph: all its vertices have a common attribute called the process identifiers, and can be partitioned into n equivalence classes according to this process identifier attribute. The process-identifier also classifies the happened before relationship that exists between events: *happened before* relationships between events occurring on the same process is imposed by the program order, while that occurring on different processes are due to inter-process communication. Neither DPE nor PEDL recognizes this difference. Thus the only way they could differentiate between the behaviors shown in Figure 3.22 is through constraints on process identifiers. This, however, compromises abstraction and scalability of a modeling language. We differentiate between the two relationships – a *happened before* relationship imposed by program order is represented by simple concatenation in Ariadne, while one imposed by inter-process communication is represented by the symbol `@`.

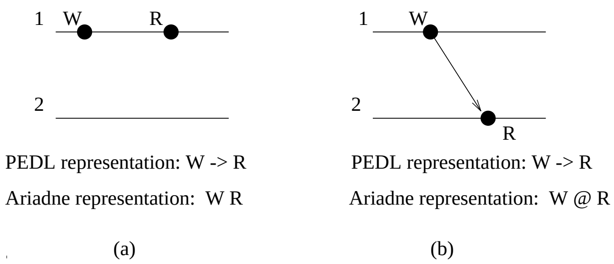


Figure 3.22. Lack of precision of PEDL modeling. The behaviors shown in (a) and (b) cannot be distinguished in PEDL without the use of constraints on process identifiers.

A second limitation of PEDL is the semantics of its parallel closure operator, denoted by $||*$. It is designed to mimic the largest possible parallelism present in the computation, and Hough envisioned using it to model concurrent behavior on a set of processes. The parallel closure operator is not very robust in presence of an error. In Figure 3.23, the user expected all the *a* events to be parallel: but the one in process 3 and 4 have a *happened before* relationship between them. PEDL can match either of the grouping shown in Figure 3.23.

The parallels closure operator is ill-defined in the presence of an error because of PEDL's use of temporal relationships to guide its matching process. In the above example, PEDL will spawn *4* threads to match individual `W` events in each process; each thread will check with other threads to ensure the event consumed by it obeys

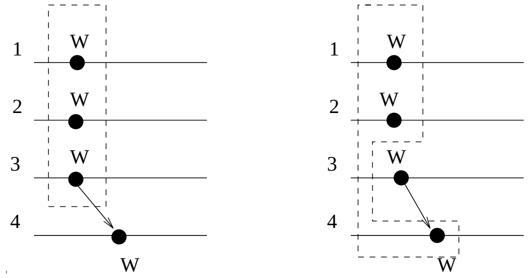


Figure 3.23. Parallel closure (||∗) is ill defined in PEDL. PEDL may match either (a) or (b) depending on which events it examines first. The events matched are shown within the dashed outline.

the temporal relationship specified by the user. The `W` events on processes 1 and 2 are pairwise parallel with other `W` events. But which of process 3 and 4's `W` event should be consumed? PEDL cannot provide a meaningful semantics in presence of such an error.

Ariadne's semantics is unambiguous. It recognizes the fact that a behavior is quite independent of the processes that take part in it. The processes under consideration are explicitly enumerated at the p-chain level. If the user had tried to match the behavior shown in Figure 3.23 and directed Ariadne to examine events on all 4 processes, Ariadne will return an error saying that the events on processes 3 and 4 are not concurrent with each other. If the user wants either of the two matching shown in Figure 3.23, he/she has to explicitly exclude process 4 or process 3 from the matching context in the p-chain specification.

The most important feature of Ariadne is its capability to provide precise feedback on errors. A model of a program behavior specifies the abstract events that make up the behavior and the existing temporal relationships between them. Ariadne's two-phase matching enables it to characterize an error precisely: it can detect whether an abstract event itself is missing, or if the abstract event is present in the behavior but but the required temporal relationships with other abstract events do not hold. Other languages lack this capability.

3.6 Conclusions

In this chapter, we introduced Ariadne's simple modeling language for specification of coarse grained models of parallel program behavior, and Ave's scalable visual feedback scheme to verify assertions on matched behavior. The whole design of Ave/Ariadne is geared to handle massive parallelism: the behavioral models expressed in Ariadne's modeling language are scalable (since abstract behavior is separate from processes exhibiting the behavior); Ave's visual feedback scheme is scalable (since abstract events are categorized into equivalence classes in both the match tree visualization, and in the scatter plots depicting derived attributes); and Ariadne's implementation of its matching algorithm is scalable (since recognition of concurrent behaviors does not require building of product automaton). In the next chapter, we will present debugging of several programs to illustrate the versatility of Ave/Ariadne.

Chapter 4

Ave/Ariadne: Examples and Conclusions

In this chapter, we will present six examples to illustrate the use of Ave/Ariadne debugger in the initial stages of debugging. In each case, we write coarse grained behavioral models with Ariadne's modeling language, refine them with Ave's query language, and use Ave's scalable visual feedback to detect errors. The first two debugging sessions deal with shared memory programs written for the Sequent Symmetry, while the last four consider explicit message passing programs written for the Intel Paragon.

4.1 Sample Debugging Sessions with Ave/Ariadne

4.1.1 Dynamic Task Allocation

This program and the detected bug was taken directly from the *Sequent Programming Guide* [74]. It was written for shared memory and uses a dynamic task allocation scheme to search a list of cities for the one closest to an origin. Each process repeatedly gets a sublist of cities and determines whether any of them is closer than the closest city currently known. The number of cities allocated to a task at one time is stored in `BITE`. With `BITE` set to 1, the program got the correct answer but when we examined its behavior to get a better understanding of it's load balancing properties, we discovered an anomaly that indicated a performance bug.

We ran the code on a four processor system with 10 cities using instrumentation that generated an `R_list` event whenever a process accessed a city from the list and an `R_var` or `W_var` whenever it read or wrote the shared variable containing the currently known closest city. The initial model was

```
ch accs = R_list R_var (W_var)*;
pch srch = accs* onall {0,1,2,3};
match find = srch;
```

The pattern matched and resulted in the Ave feedback shown in the background window of Figure 4.1.

To look at the load balancing properties of the program, we computed the following attributes giving the percentage of the load (that is, the percentage of the cities) handled by each process

```
with find foreach accs compute flag = 1;
with find foreach proc compute load = (sum(flag(accs))/10)*100;
with find show load(proc) wrt proc;
```

The `show` command drew the category statistics graph given in the foreground window of Figure 4.1 (it also drew the scatter plot but since we are interested only in the actual values here, we did not reprint it.) Looking at the figure, it is clear that something is wrong, as the sum of the load is 190%[1]!

To understand this behavior better, we reinstrumented the program to trace attributes of the `R_list` event. When a process acquired a sublist, it was given the base location for the start of that sublist and it then accessed the next `BITE` elements using a loop with variable `index`. To trace these loops, we tagged each

[1]The % signs on the top of the bar graphs in the category statistics is added by the user. Ave allows the use to specify the units in which bar graphs should be displayed.

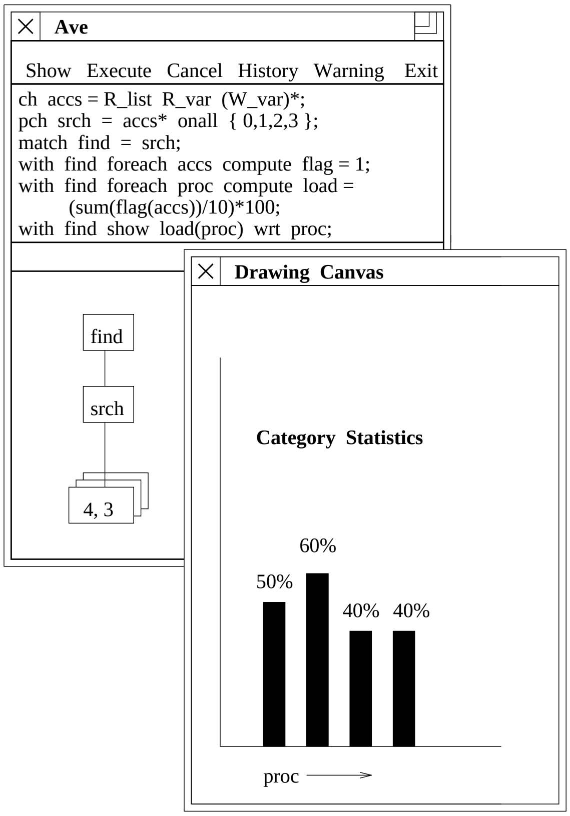


Figure 4.1. Ave feedback showing that the load is more than 100%.

`R_list` event with `base` and `index` attributes. The new trace was then processed using the same behavioral description and the commands

```
with find foreach accs compute offset = index(R_list)-base(R_list);
with find show offset(accs) wrt proc;
```

Note that in this particular case, because `BITE` was 1, the loop should be executed only once for a given value of `base`, and `offset` should always be 0. The scatter plot of this attribute should have just one color. Ave, however, produced the scatter plot in Figure 4.2 with alternating colors.

Each process was accessing the list at two locations per iteration instead of one. A look at the upper limit on the loop revealed that it was incorrectly set

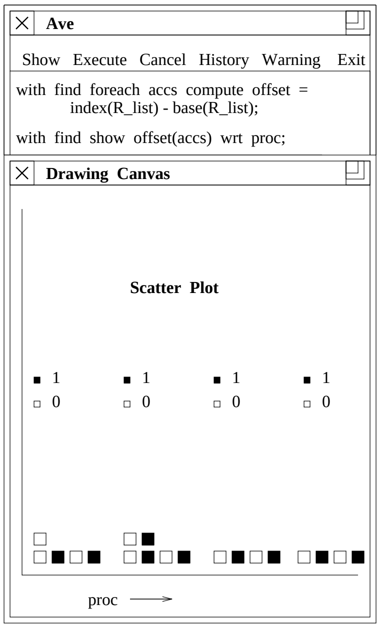


Figure 4.2. Ave feedback showing that variable **base** took two values in each iteration.

as `upper = base + BITE` instead of `upper = base + BITE - 1`, resulting in each element except the ninth being examined twice.

4.1.2 Gaussian Elimination

This code is a shared memory Gaussian elimination with static data partitioning; we ignore the back substitution step and focus on the elimination. The program executes $n-1$ iterations for an $n \times n$ system of equations. In each iteration, the "parent" process performs any necessary *pivoting* and then all processes manipulate the individual rows for which they are responsible. The program produced correct results on some test runs but incorrect results on others; on one 10×10 matrix, the program failed when run with nine processes but worked correctly with fewer than nine.

The program was instrumented to record read and write accesses to rows of the matrix as single events. The accesses during the pivot and elimination steps were recorded as `R_pivot`, `W_pivot`, `R_elim`, and `W_elim` respectively. These events were then used as the basis for modeling the system's behavior.

For pivoting, the parent process swaps two rows to get a non-zero element in the pivot position, which is modeled at the chain level as two read accesses followed by two write accesses:

```
ch pivot = R_pivot R_pivot W_pivot W_pivot;
```

Similarly, each process in the elimination step reads the pivot row and then reads and writes each its rows. This is modeled as

```
ch elim = R_elim (R_elim W_elim)+ ;
```

and the behavior of a process during a single iteration of the program is

$$\texttt{pivot} * \ \texttt{elim}$$

where the Kleene star on **pivot** is to model the fact that pivoting may or may not occur in any given iteration. This behavior is bound to the process set by the p-chain definition

```
set procset = { 0..8 };
pch iter = pivot* elim onsome procset;
```

The entire behavior of the program – a sequence of iterations – is defined by

```
set procset = { 0..8 };
ch pivot = R_pivot R_pivot W_pivot W_pivot ;
ch elim = R_elim (R_elim W_elim)+ ;
pch iter = pivot* elim onsome procset;
match gauss = <iter^9 >;
```

where the **match** statement here indicates that 9 instances of the p-chain **iter** are to be matched in the execution trace and assigned to the variable **gauss**; the angular brackets indicate the logical relation of **precedes** must hold between successive instances. Figure 4.3 shows the result of the match.

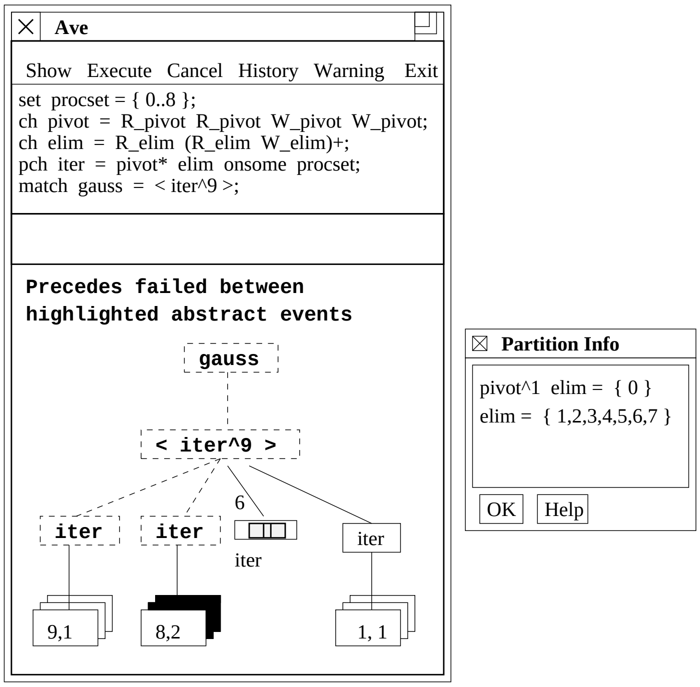


Figure 4.3. Ave feedback showing **precedes** failure in Gaussian Elimination.

The match tree in the figure shows an unsuccessful model match as indicated by the error message

Precedes failed between highlighted abstract events

This feedback led us to the bug. The failure of `precedes` is likely the result of a race: some process must not have finished its first elimination step when the parent began its second pivot step causing contention for some row of the matrix. An examination of the code revealed that this error had indeed occurred as the result of a missing barrier synchronization between iterations. (Note: the program worked when number of processes was less than nine because the parent process was responsible for more than one row and thus did not get to its next pivot as quickly.)

4.1.3 FFT on a Mesh

This program is an MIMD, message-passing version of the FFT on a mesh with static data partitioning. It was written for 128 nodes, thus it executed seven iterations with processes communicating at distances of 64, 32, 16, 8, 4, 2 and 1 unit apart on successive iterations. The program produced incorrect results. Initially, we modeled its intended behavior as

```
set procset = {1..128};
ch exch = W @ R ;
pch comm = exch onall procset;
match ffts = comm^7;
```

where `exch` is the pattern for a single message. The p-chain `comm` binds `exch` to the set of processes and the keyword `onall` indicates that a successful match must occur on every process. Note that although the user has requested that 7 instances of `comm` be matched, nothing is said about the logical relationship between successive iterations. In this example, the match was successful and it resulted in the assignment shown in Figure 4.4.

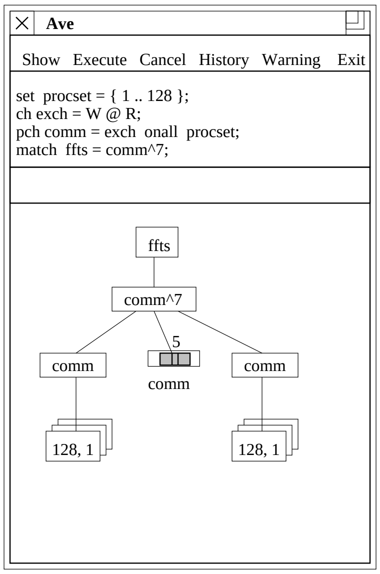


Figure 4.4. Ave feedback in the FFT example.

At this point in our debugging, we knew that the **exch** and **comm** patterns were successfully matched but they are both coarse patterns that could match many behaviors. To explore this execution history graph further, we used the command

```
with ffts foreach exch compute dist = abs(pid(W)-pid(R));
```

which computes a new attribute, named **dist**, for each event **exch**. The value of the attribute is the distance (based on PIDs) between the sending process (**pid(W)**) and the receiving process (**pid(R)**).

```
with ffts show dist(exch) wrt comm;
```

The display resulted in a scatter plot[2] `comm` phase as shown in the window *Drawing Canvas* of Figure 4.5.

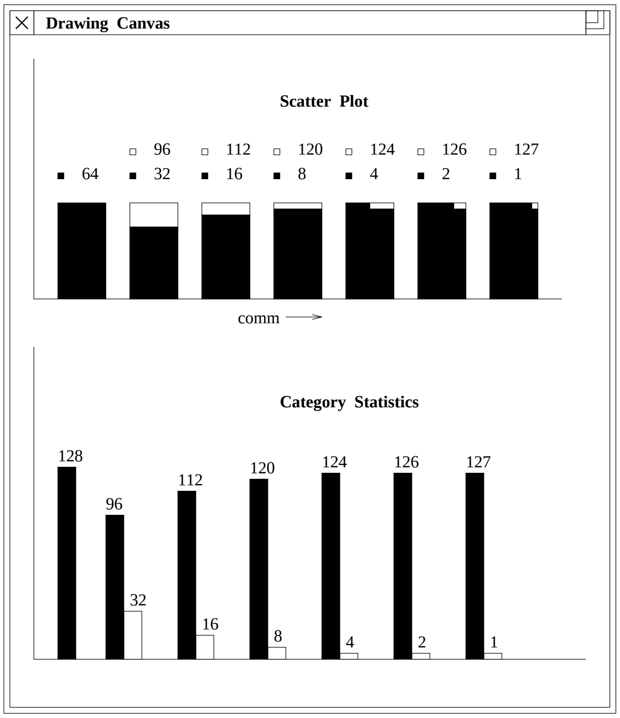


Figure 4.5. Ave feedback in the FFT example showing wrong partner computations.

The plot shows a partitioning of attribute values with each partition mapped to a color; note that automatic color binning has been used with the color *black* representing the range of values {*1, 2, 4, 8, 16, 32, 64*}, while the color *white* is used to represent the values {*96, 112, 120, 124, 126, 127*} [19]. We intentionally set the number of colors to 2 to show that color binning can be used successfully even with limited number of colors.

[2]Note the individual rectangles in the scatter plot can not be distinguisehed. Ave allows the user to select the height and width of individual rectangles of the scatter plot, and the horizontal and vertical separations between individual rectangles. In this case, we have set the horizontal and vertical separtaions to be zero.

The scatter plot led us to the bug. During a phase, all processes should communicate across the same distance and thus, the scatter plot for each phase should consist of rectangles of a single color. For this program, however, only the first phase was uniformly colored; the others all had at least some nodes of a second color. This observation led us to an "off by one" bug in the code which computes the destination of data exchanges. Note that the numbers in the *Category Statistics* does not signify the value of the attribute *dist*, but the frequency distribution of the attribute[3].

4.1.4 Jacobi-like Singular Value Decomposition

The algorithm consists of a number of iterations. At any given iteration each processor takes part in a number of *compute* phases followed by a single *divide* phase. Between every two *compute* phases, each processor exchanges data with one of its neighbor: the identity of the neighbor is determined by the iteration count and the *compute* phase count. The *divide* phase too, consists of processors exchanging data with partners, where the identity of the partners are determined only by the iteration count. Thus, although the whole computation has a fixed communication pattern, it is hard to express statically.

Messages exchanged in different phases were tagged differently, `cmp` for the *compute-phase* exchanges, and `dvd` for *divide-phase* exchanges. The program produced incorrect result. Clearly, something has gone wrong with the exchange of columns. Ariadne model for the behavior was written as:

---

[3]The fact that there are 96 occurrences of the value 32, and 32 occurrences of the value 96 in the second iteration (and similar correlation in each subsequent iteration) is purely coincidental. A cursory glance of the *Category Statistics* may falsely suggest some subtle error in the debugger's way of using color.

```
ch msg_comp = W_comp @ R_comp;
pch compu = msg_comp onall procset;
ch msg_dvd = W_dvd @ R_dvd;
pch divide = msg_dvd onall procset;
match svd = < (compu+ divide)^4 >;
```

The match succeeded, and we proceeded to check other assertions about the communication pattern. Since the *compute* phase had more intricate communication pattern, we decided to investigate that first.

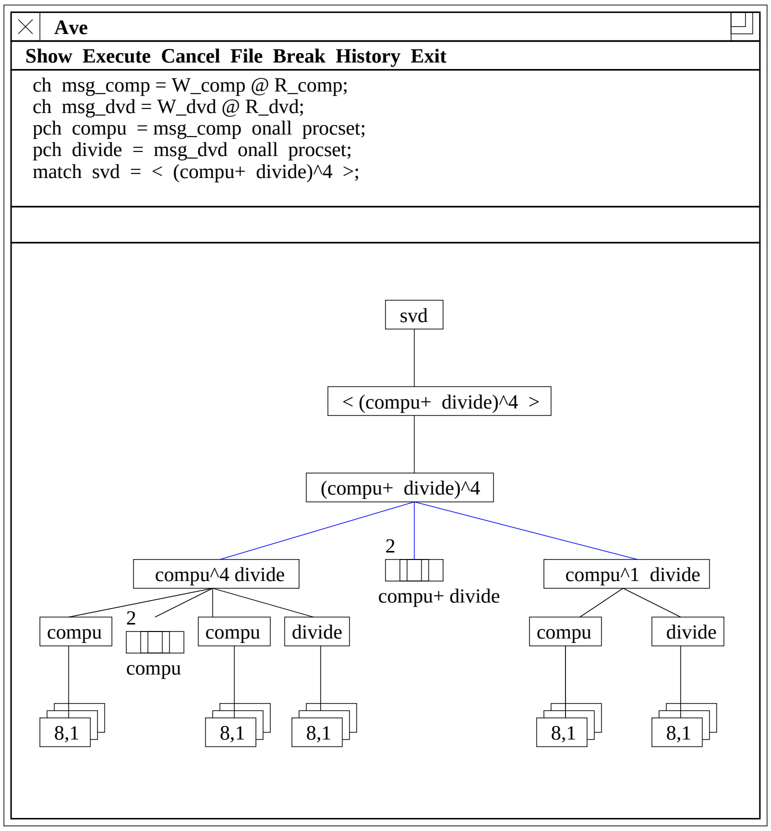


Figure 4.6. Ave feedback showing pattern describing the singular value decomposition matched the observed behavior.

As in the case of FFT example, the exchanges taking place during each **compu** abstract event must take place between paired processes. The chain level description does not enforce this constraint. So we followed the same approach as we did in the FFT example. We first computed the distance over which each exchange took place, and plotted them against the `compu` abstract event.

Note that the neighbor computation is not straight-forward, it is hard to express an assertion over a series of exchanges, without writing down the actual function used in the source code to compute them. But certain observations can be made about the exchanges:

1. Every compute phase starts with an exchange over a distance 2^0
2. If we number the first exchange as exchange #1, then every odd exchange in a compute phase would be over a distance of 1
3. Two consecutive even exchanges would always be over different distances
4. If we plot the distance over which the exchanges take place, we will see a varying amplitude triangular wave, whose maximum will occur at the middle.

With color-binning, it is hard to tell whether the assertions hold, so the user had to use the *line-plot* facility to gain a better understanding of the behavior.

```
with svd foreach msg_comp compute dist = abs(pid(W)-pid(R));
with svd foreach exch show dist(msg_comp) wrt comp;
```

This produced the Ave feedback shown in Figure 4.7 where all the assertions seem to hold.

This is one important aspect of our basic philosophy: Use coarse grain query, and deduce from the feedback whether fine grain properties hold. Making queries coarse grained makes it less likely that user would inadvertently make a mistake. We do not allow the user to specify complicated functions, asking instead to look at

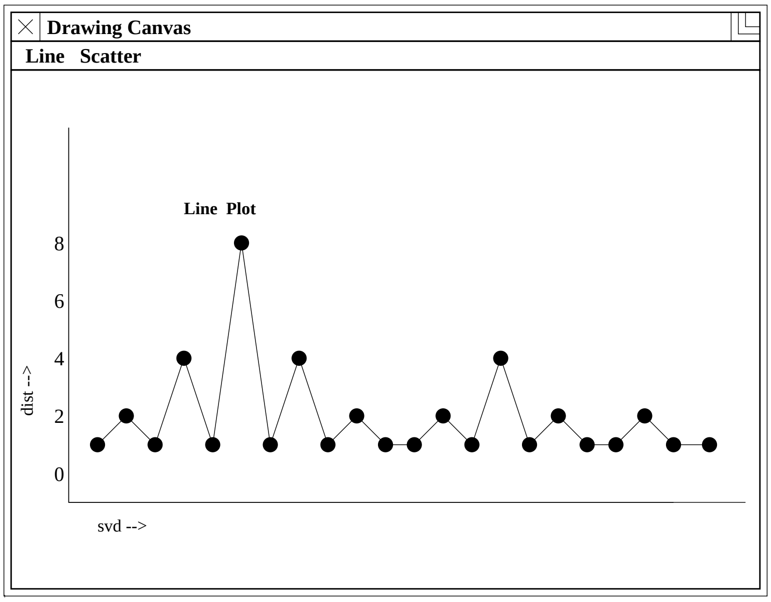


Figure 4.7. Ave line plot showing the exchange distances for different *compute* phases.

graphs that illuminate functions' characteristics. Most likely that the user has used the same function in the original source code to implement the communication; and rewriting would generally involve copying the same code over. Not only is this a redundant exercise, there is no way one can find error that may be in the implementation of the function itself.

The next assertion was on the *divide* phase.

```
with svd foreach msg_dvd compute dist =
      abs(pid(W_dvd)-pid(R_dvd));
with svd show distance(msg_dvd) wrt divide;
```

which produced the Ave display shown in Figure 4.8.

The *divide* phase has a more straight forward behavior. The *i-th divide* phase consist of exchanging data between processors that are 2^{d-i} apart where d is the dimension of the logical hypercube, and the first iteration is numbered 1. Since we have a 3 dimensional hypercube, and there are 4 iterations, the last iteration would

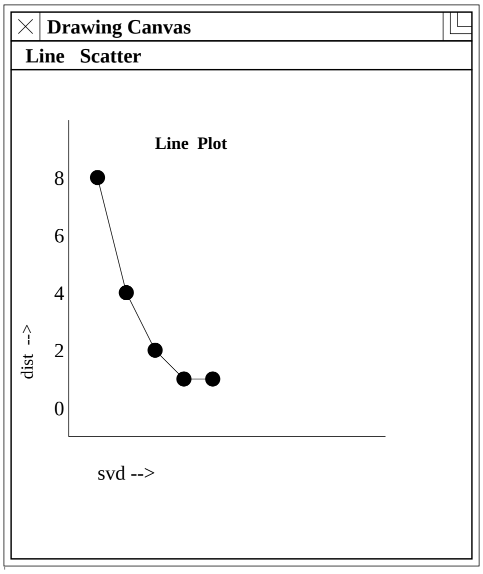


Figure 4.8. Ave line plot showing the exchange distances for different *divide* phases.

require data to be moved over a distance of 2^{-1}. The programmer had tried to take care of the situation by mapping distances less than 1 to 0, and thus at the last iteration each processor sends and receives data from itself. Thus the line-plot should show a monotonically decreasing function that will saturate at 0. But the line-plot showed that the last two *divide* phase has processors distance 1 apart exchanging data.

But at the last divide phase, the function **myexp2()** that computes the distance over which communication is to take place is supposed to get -1 as its input and return 0 as output. A closer look at the function revealed that the programmer has no check for negative arguments, thus arguments less than or equal to zero all gets the value 1. The code was reused from another program that performs Poisson method of solving partial differential equations, which does not need to handle negative *indices*.

Notice without the event-based modeling, the user would have no way of knowing where to look for errors. Users always expect complex codes to fail, and have enormous faith in the simpler sections of the code. Without the availability of event-based modeling facility, a likely suspect would have been another function *h()* that computes the distance of the partner at every *compute* phase. With Ariadne's modeling and Ave's attribute computation, we could eliminate the possibility of that part of the code being responsible for the mistake with a couple of queries.

An alternative would have been to test each function in the program separately with all possible test inputs to verify that they are correct. With large programs consisting of many modules, however, this is hardly feasible.

4.2 Other Features of Ave/Ariadne

Ave/Ariadne features two other powerful mechanisms – a sophisticated filtering scheme and the capability of imposing multiple views on a behavior. We showcase both features through another sampling debugging session.

The filtering mechanism [46] allows for conditional attribute expressions to be used to guide the matching process as in EDL [9]. Unlike EDL, however, Ariadne's filtering scheme is restrained.

Use of multiple views of program behavior had been advocated strongly in [53]. Ariadne provides it in two ways: it allows the user to select a portion of the execution history and model the behavior of the selected events in multiple ways, and it provides the capability of partitioning the abstract events into equivalence classes based on their derived/primitive attributes. Once the user has partitioned an abstract event into different equivalence classes, Ave can be used to visualize the match tree as it is reflected through the new structural transformation due to the partition.

4.2.1 α-β Search: The case for Multiple Views

The related program performs a bi-directional α-β search on a 8-puzzle game tree on a message passing MIMD computer. Such searches involve finding a path from an initial configuration of the puzzle to a user-defined goal state. To make the search efficient, a heuristic is used to prune the game tree. In a bidirectional implementation, two searches are performed simultaneously, one going from the initial state to the goal state, and the other from the goal state to the initial state. A solution is found when two searches meet. A typical search scenario is presented in Figure 4.9.

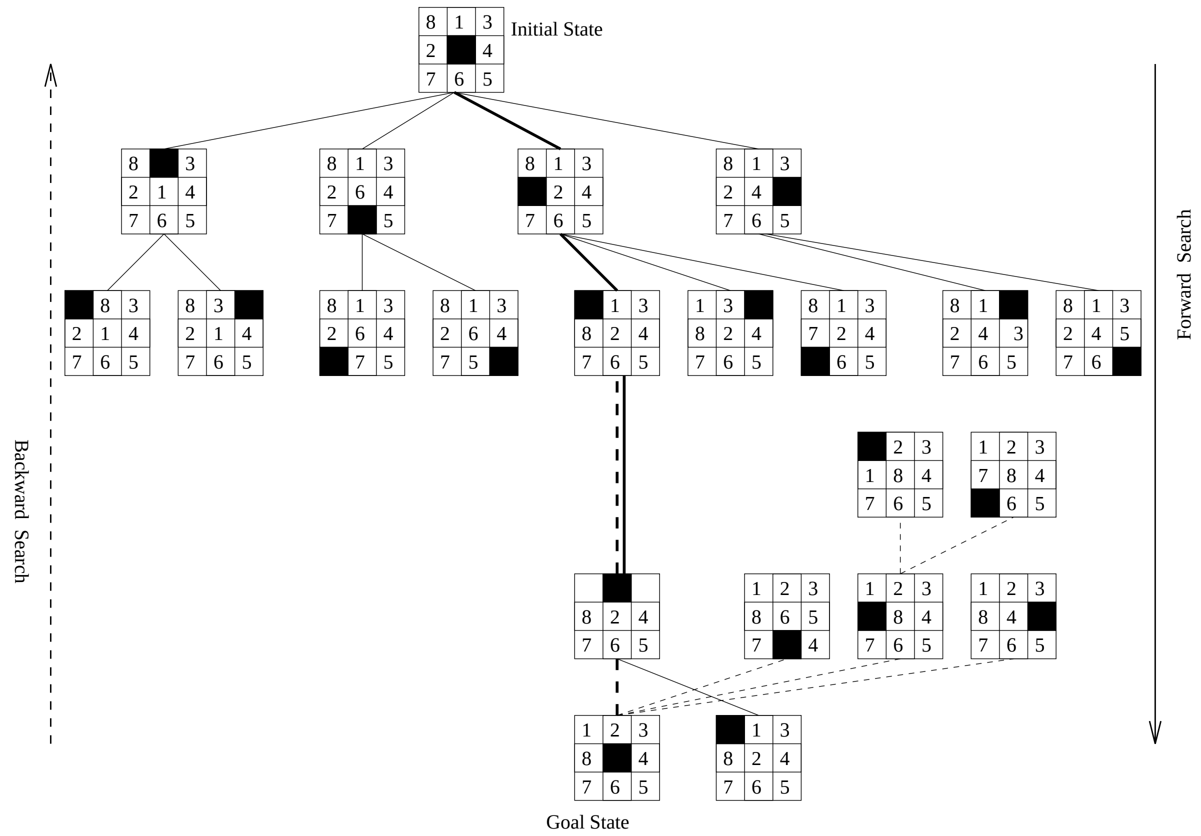


Figure 4.9. A bidirectional $\alpha - \beta$ search. Thick line indicates the solution path.

The parallelization effort assigns two managers to coordinate the searches in each direction. The manager sorts the unexpanded tree nodes (nodes whose children have not yet been generated) according to the heuristic function, and assigns each worker (including itself) a single node to work on. To achieve the pruning and termination detection, each worker broadcasts its successor list to all the processors. Upon receiving such a broadcast, a worker checks to find out whether it had ever generated any of the successor nodes, and replies accordingly. Each processor finds out whether another processor has generated the same tree node as itself — if so then if the other processor is searching in the same direction the tree node can be pruned to avoid duplication of work [4], and if the other processor belongs to the pool doing the search in the other direction, a solution has been found and the search should end. Once a worker receives replies to its broadcast, it sends the existing nodes (those not pruned) to its manager for resorting and expanding. The cycle repeats until the search termination is detected.

A worker node receives a node from the manager, expands it, and broadcasts the generated nodes to every processor to check for termination or potential pruning opportunity. It then receives and answers broadcasts from other workers. Finally it receives the answers to its own broadcasts, prunes the duplicate node(s) if any, and send the non-pruned nodes back to the manager, to be sorted according to their heuristic value. So the behavior of the worker after it receives an assignment from the manager can be written as:

[4]We need to assign some sort of priority. If two processors searching in the same direction generate the same node, the processor with the lower identifier prunes it from its list.

```
ch worker = M_check @ R_check W_answer @ R_answer W_reply @ R_reply;
```

The manager sorts the nodes according to their heuristic function, sends distinct nodes to each worker in its pool, mimics the worker, and then receives the non-pruned nodes from each worker before resuming the cycle. The assignment of work from the manager to a single worker can be written as:

```
ch assign_work = W_assign @ R_assign;
```

The reply contains a special field that indicates whether the termination has been detected or not. If termination has been detected, then the manager sends out a message tagged as `stop`. This can be modeled as:

```
ch stop_work = W_stop @ R_stop;
```

The whole computation can be specified, when seen from the perspective of a manager processor as

```
set procset = {0..7};
ch assign_work = M_assign @ R_assign;
ch worker = M_check @ R_check W_answer @ R_answer W_reply @ R_reply;
ch stop_work = M_stop @ R_stop;;
pch work_cycle = (assign_work worker)+ onsome procset;
pch work_terminate = stop_work onsome procset;
match search = < work_cycle+ > work_terminate;
```

The Ave output in Figure 4.10 shows that the `precedes` relationship does not hold between successive iterations. Since the manager process initiates every iteration, this was a surprise. Note that the model does not impose the program

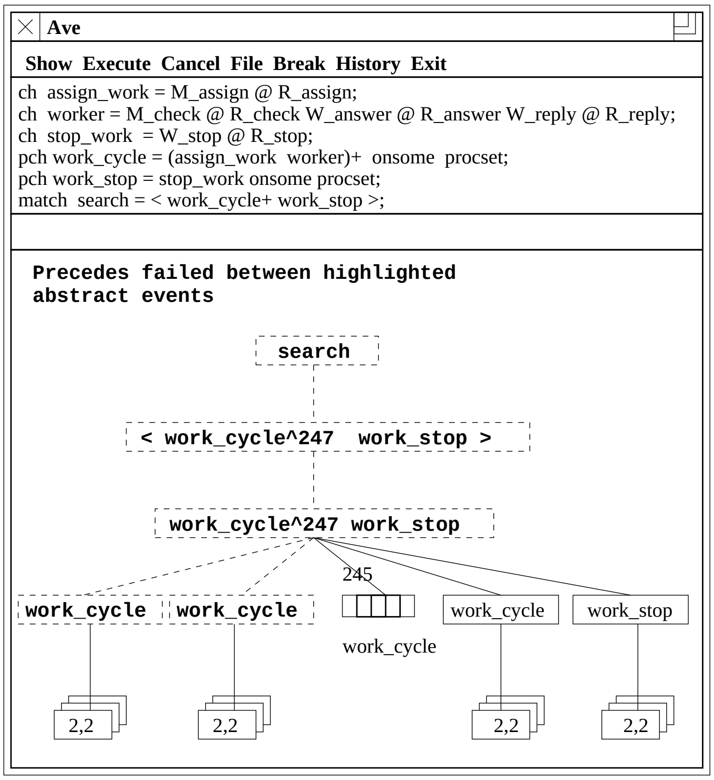


Figure 4.10. Ave feedback on $\alpha - \beta$ search example.

order embedded in the manager process. Since there are 4 processes in each pool, each manager process sends 3 **assign** messages (`W_assign^4`), it then broadcasts a **check** message containing the new successor nodes it produces to all the other processes (`M_check`, reads and answers to 7 **check** messages (`(R_check W_answer)^7`) from the other 7 processes, reads 7 answers it receives from the other processes (`(R_answer)^7`), and then receives 3 replies from the other processes in its own pool (`R_reply^3`). To verify whether that program order is what we expected, we imposed the following view on our match. We modeled the program order of the manager process as:

```
ch mgr = W_assign^4 M_check (R_check W_answer)^7 R_answer^7 R_reply^3;
pch work_mgr = mgr onsome procset;
```

```
match search_mgr = work_mgr+ work_terminate with search;
```

In Ariadne's matching scheme once an abstract event is matched all its constituent events are removed[5] from the trace. Thus the primitive events forming the abstract events **search** need to be restored in the trace, and the new matching should be confined only to these restored primitive events. The **with** clause at the end of the match statement accomplishes three things:

1. it restores the primitive events returned by **search** before the matching is started;
2. it forces the match to be performed only against the returned events;
3. once the matching is performed it again removes all the returned events from the trace.

The outcome of the new match, with the error path highlighted with dashed lines, is shown in Figure 4.11. The figure illuminates the error in more detail. The manager was only receiving 1 reply instead of 3! Ave does not identify the exact cause of such behavior, but the next reasonable step would be to examine the section of code related to the receipt of answers in the manager process. One will then presumably set breakpoints at different parts of the source code to explore the behavior incrementally hoping to uncover the bug. In the next section, we present our approach to setting consistent global breakpoints that takes the guesswork out of the exercise, and stops the computation in a meaningful state.

---

[5] Events are removed in a logical sense. A bit is set for each event, marking it disabled so that it is no longer considered for subsequent matching. Restoring the events can then be accomplished by simply resetting the bit.

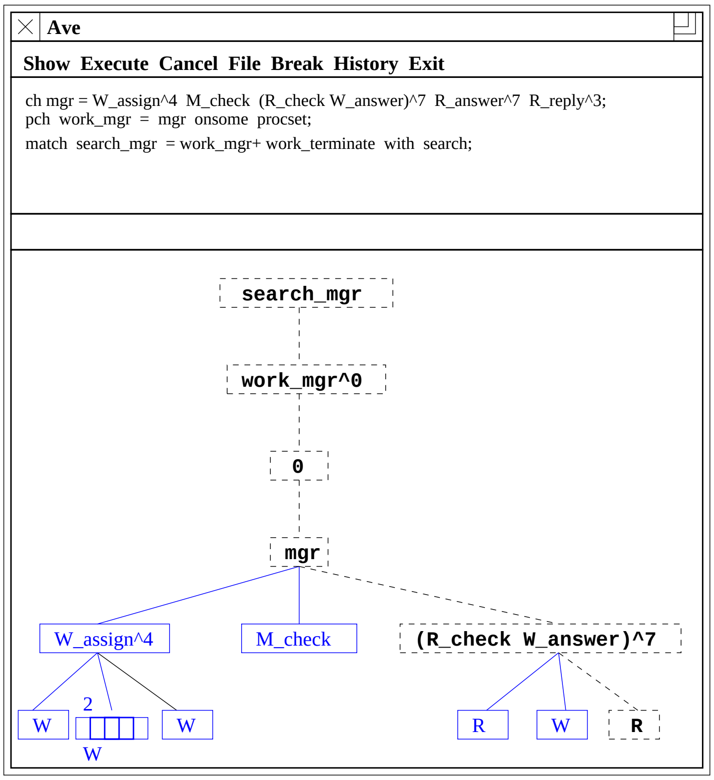


Figure 4.11. Ave feedback on a different viewpoint of $\alpha - \beta$ search example.

4.2.2 Example: Finding Biconnected Components on a Network of Processors

We use this example to illustrate Ariadne's filtering mechanism and its support for multiple views. The distributed algorithm finds biconnected components on asynchronous networks. The algorithm proceeds in three phases: (1) A spanning tree of the network rooted at the initiator is constructed, (2) the articulation points in the tree are identified, and (3) individual biconnected components are found by initiating multiple election algorithms in link-disjoint portions of the network.

The initial communication pattern of the first phase is a spreading activation that reaches each node in the network (note the initiating node later becomes the root of the spanning tree). All of the spreading activation messages are tagged as `sprd`. Upon receiving the first such message, each node floods all of its other links: the other `sprd` messages received are ignored. The sender of the first `sprd` message received is identified by the receiving node to be its parent in the spanning tree: At the end of the spreading activation, each node in the network will have identified its parent. The nodes of the network that receive a `sprd` message from all their neighbors identify themselves as the leaves of the spanning tree.

Figure 4.12 shows a possible resolution of a spreading activation in a network of nodes. Except for the initiating node, each node sends messages to all but one of its neighbors. Note the `sprd` messages are allowed to race. In Figure 4.12(e) both nodes 2 and 8 send `sprd` messages to node 7, but the message from node 8 reaches it first and as a result 8 gets identified as the parent of node 7. With different executions the program may produce different spanning tree. Another possible outcome of the spreading activation is shown in Figure 4.13. Since, the number of neighbors of a node may vary we model only the coarse grained behavior that node may send multiple `sprd` messages.

Consider node 2 in Figure 4.12. It receives its first `sprd` message from node 1, and sends `sprd` messages to nodes 3, 4, 5 and 7. The portion of the time diagram is shown in Figure 4.14.

It would seem we can model the multiple messages sent by a process with the chain `(W_sprd @ R_sprd)*` by matching the send and the receive of individual messages. With Ariadne's matching algorithm, however, as soon as the symbol `@` is encountered the matching context shifts to the receiving process: in our case with the first `@` it will shift from process 2 to process 3, and Ariadne will attempt to match the next message send in the context of process 3. Somehow we need to

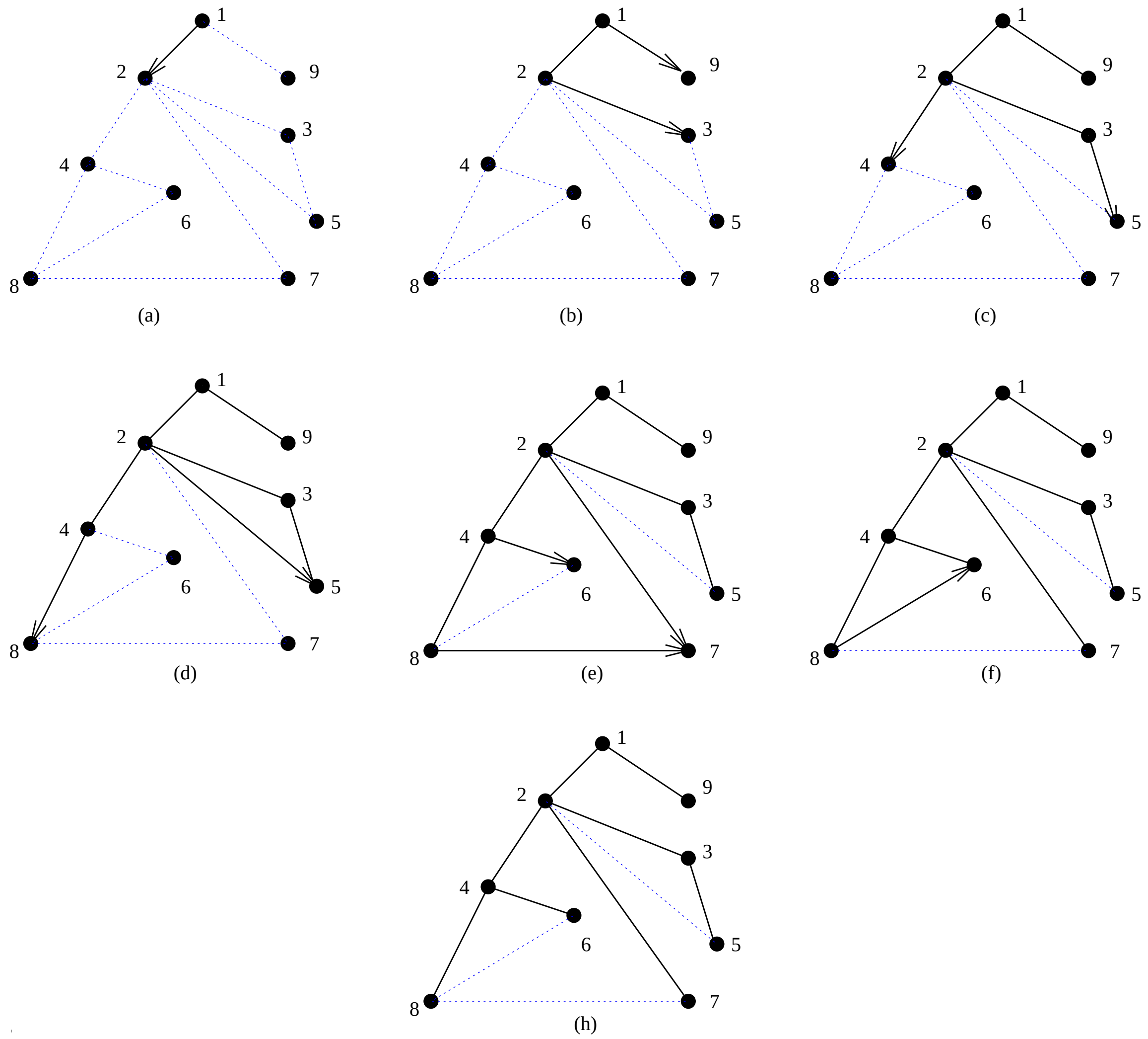


Figure 4.12. Progress of the spreading activation of **sprd** messages in Biconnected Components algorithm. Dotted lines signify original network connection. Node 1 initiates the algorithm (a); the spreading activation (b) through (g), the solid lines identify the parent edges identified by **sprd** messages encountered earlier; the final spanning tree with the tree-edges identified by solid lines, and the non-tree edges identified with dotted line (h).

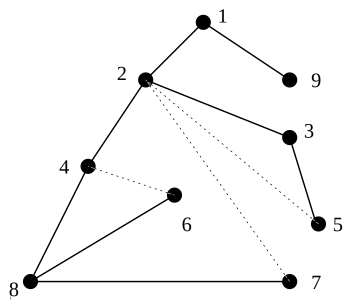


Figure 4.13. A second possible spanning tree resolution for the network. The tree-edges are solid, and the non-tree edges are represented with dashed lines.

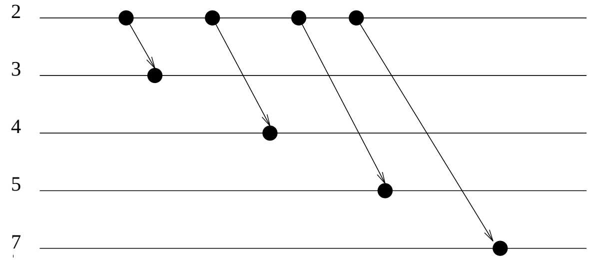


Figure 4.14. Behavior of process 2 during the spreading activation phase.

remember that the matching started in the context of process 2 and return to our original context. Ariadne accomplishes that with a chain level directive `<:` that saves the current context of the match. A second directive `:>` restores the most recently saved context. With these two directives, we can now write the coarse grained specification as:

```
ch sprd_scatter = ( W_sprd <:  @ R_sprd :>)*
pch span_sctr = sprd_scatter onsome procset;
ptch span_begin = span_sctr+;
```

The matching of the chain **sprd_scatter** on process 2 occurs as follows: Ariadne matches the **W_sprd** on process 2, encounters the `<:` directive and saves process 2's context in a stack, follows the **@** to process 3, matches **R_sprd** on process

3, and upon encountering the directive `:>` restores the context of process 2. Since there is a Kleene star in the chain definition, the matching starts again, this time matching the `W_sprd` event that corresponds to the message sent to process 4. The overall specification, however, is coarse: it does not specify the exact number of messages sent by each node, it does not even require that all nodes receive a **sprd** message.

This communication pattern is followed by a reduction where messages travel upward from each node to its parent (through the links that form the spanning tree). The reduction phase starts with the leaves of the spanning tree sending a message tagged as **gthr** to their parents with a content of 1. Each internal node upon receiving **gthr** messages from all its children sends a message to its parent: the message contains the number of nodes that make up the subtree rooted at the sending node (this can be done by adding the contents of the messages received from its children and adding one to the sum). Figure 4.15 shows for our example network and a given spanning tree resolution, how **gthr** messages are sent.

Again the behavior can be expressed coarsely in Ariadne as:

```
ch gather = W_gthr @ R_gthr;
pch span_gthr = gather onsome procset;
ptch span_end = span_gthr+;
```

The two sub-phases can be combined to specify the first phase that builds the spanning tree, as

```
ptch Span = span_begin span_end;
```

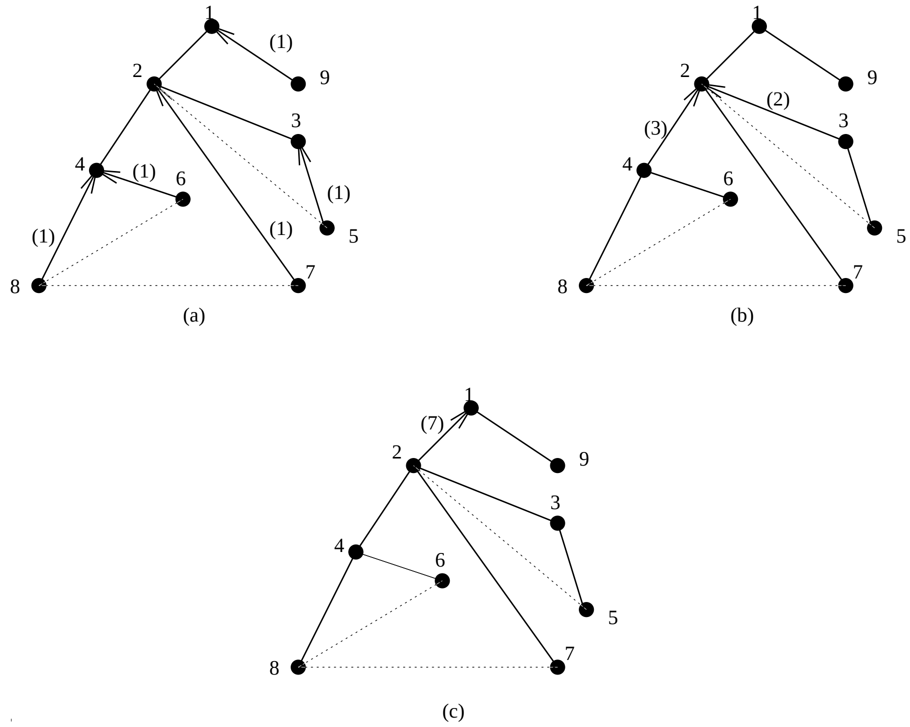


Figure 4.15. Messages in the **gthr** phase. Solid arrows indicate **gthr** messages. The number in parentheses beside a message indicate the message content. Dashed lines indicate non-tree edges.

The second phase identifies articulation points of the spanning tree by ordering the nodes in the network by a depth-first traversal of the spanning tree. Each node, upon receiving a **pre** message from its parent, sends a **pre** message to all its non-parent neighbors: first to all its children and then to all its non-child neighbors. The messages sent to the children are sent in sequence: the node waits till it receive a reply from a child (in form of **gthr** messages) before it sends a **pre** message to its next child. **pre** messages sent to a child contains the child's sequence number as computed by the parent. Once a node has received replies from all its children, it sends a **gthr** message back to its parent, containing the maximum sequence number used by the sub-tree rooted at the node. **pre** messages sent to non-child nodes contain the sequence number of the sender, the receiver node updates its internal table with the information but does not respond to the message. Thus the behavior that modeling the messages sent to child nodes resemble a depth-first traversal of the tree.

For the network and the spanning tree shown in Figure 4.13, the messages sent to child nodes are strictly ordered, and can be enumerated as follows, where $t_{i,j}$ refers to a message of type t is sent from process i to process j.

$pre_{1,2} \rightarrow pre_{2,3} \rightarrow pre_{3,5} \rightarrow gthr_{5,3} \rightarrow gthr_{3,2} \rightarrow pre_{2,4} \rightarrow pre_{4,6} \rightarrow gthr_{6,4} \rightarrow pre_{4,8}$
$\rightarrow gthr_{8,4} \rightarrow gthr_{4,2} \rightarrow pre_{2,7} \rightarrow gthr_{7,2} \rightarrow gthr_{2,1} \rightarrow pre_{1,9} \rightarrow gthr_{9,1}$

Many nodes also send **pre** messages to non-child nodes, once they have received messages from all their children, but these messages do not figure in the total order. For instance, node 5 sends a **pre** message to node 2 immediately before it sends a **gthr** message to node 3 ($gthr_{5,3}$). Similarly, messages $pre_{6,8}$, $pre_{8,6}$, $pre_{7,8}$, and $pre_{2,5}$ are respectively sent before the relevant processes (6,8,7 and 2) sent the messages $gthr_{6,4}$, $gthr_{8.4}$, $gthr_{7,2}$ and $gthr_{2,1}$. Only the sends of these **pre** messages figure in the total order specified above, but nothing can be specified regarding their receives. In the scenario shown in Figure 4.16, for example, process 2 receives $pre_{5,2}$ before it receives $gthr_{3,2}$ but program flow would have allowed it to receive the messages in the opposite order. Ariadne's coarse grain model descriptions are flexible to accommodate such non-determinism.

We need to use the directives related to saving and restoring match contexts to model such a depth-first traversal. Since the nodes receiving **pre** messages through the non-tree edges do not respond to them, the coarse grain chain level description can be written as

```
ch pre_sctr = ((W_pre <:  R_pre)* (W_gthr @ R_gthr)* :>)*
pch cut = pre_scatter onsome procset;
```

Figure 4.16 shows how such a model can match a possible behavior: magenta boxes shows that the outer Kleene Star matched 13 instances of the behavior, many of the instances are very different from each other.

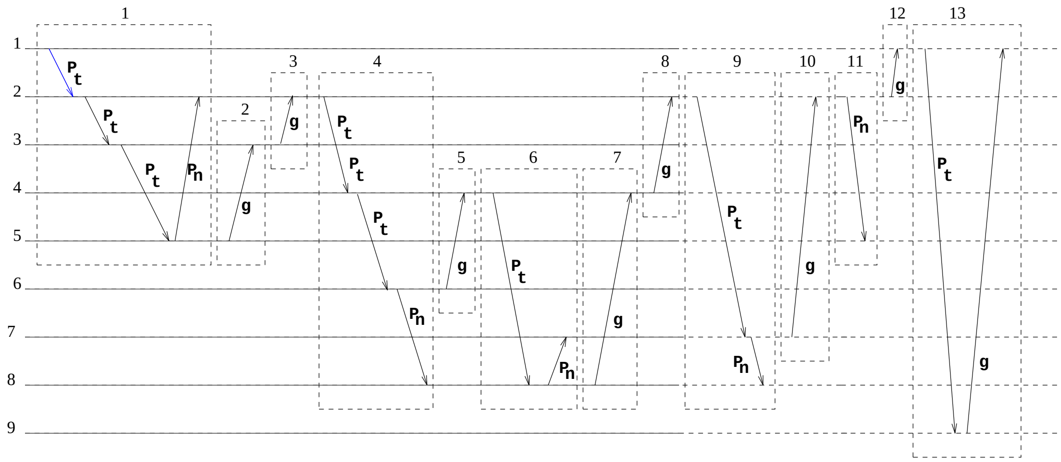


Figure 4.16. The time diagram showing the depth-first traversal of the spanning tree. The 13 instances of the pattern that make up the chain definition **pre_scatter** are identified by dashed boxes. Arrows marked p_t represent **pre** messages sent through tree edges, arrows marked p_n represent **pre** messages sent through non-tree edges, and arrows marked g represent **gthr** messages.

The third phase starts with each articulation point sending an **init** message to each of its links that are potentially in different biconnected components. These *candidate* links are identified at the end of the second phase.

```
ch color_init = W_init @ R_init;
```

The nodes receiving the **init** message start an election algorithm where each of them attempts to *color* the links in its biconnected component with its depth-first sequence number computed during the second phase. The pattern etched by the **color** messages belonging to each candidate resembles a depth-first traversal similar to that seen in phase 2, of the links forming the biconnected component. Multiple nodes can initiate the *coloring* computation within a biconnected component but only one (with the highest *pre-ordering* number) succeeds. For instance, with the spanning tree shown in Figure 4.13, both nodes 3 and 5 will receive an `init` message from node 2, and both would start the election algorithm. In a correct execution, node 5 would win the election. The Ariadne model is:

```
ch color_attempt = ((W_clr <:  @ R_clr)* gather :>)*;
pch color_phase = color_init color_attempt onsome procset;
ptch Color = color_phase+;
```

We can model the depth-first traversal as we did in phase 2. But the pt-chain level behavior is a bit different, since we can have multiple candidates trying to win the election, and each candidate may send multiple `color` messages.

The whole behavior – a sequence of three phases – can be specified as

```
ch scatter = (<:  W_gthr @ R_gthr :>)*
pch span_sctr = scatter onsome procset;
ptch span_begin = span_sctr+;
ch gather = W_gthr @ R_gthr
pch span_gthr = gather onsome procset;
ptch span_end = span_gthr+;
ptch Span = span_begin span_end;
ch cut_sctr = ((W_pre <:  @ R_pre)* gather* :>)*;
pch Cut = cut_sctr onsome procset;
ch color_init = W_init @ R_init;
ch color_attempt = ((W_clr <:  @ R_clr)* gather :>)*;
 pch color_phase = color_init color_attempt onsome procset;
 ptch Color = color_phase+;
 match Bcc = Span Cut Color;
```

The Ave output is shown in Figure 4.17.

The Ave output highlights the scalability problem. Even with aggressive compression, the output is not very scalable. There is no way to avoid the problem of scalability without user intervention. The programmer is the only one who knows

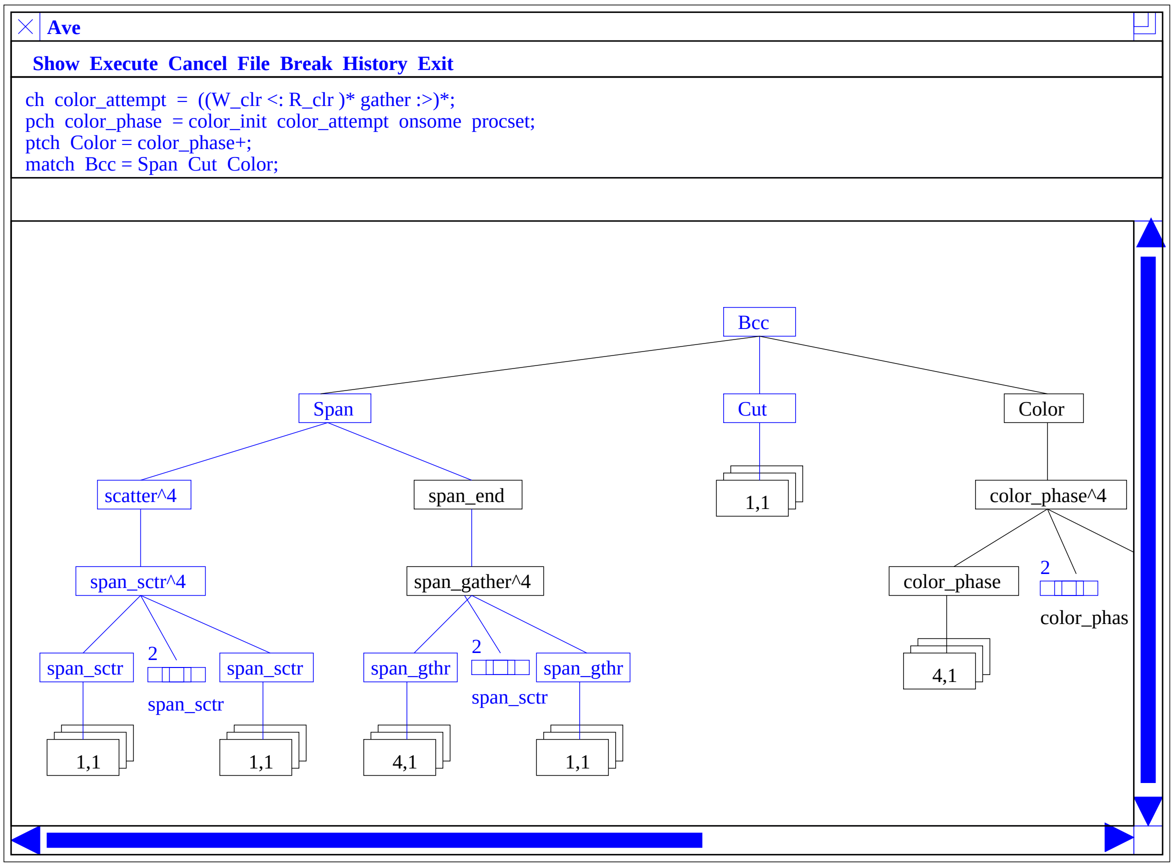


Figure 4.17. Ave output on biconnected component algorithm.

what detail can be omitted without a loss of vital understanding. Ave allows the user to control the depth of the feedback tree. For instance, the user can always direct Ave to limit the feedback to a tree of depth 3, and selectively expand parts of the tree. Figure 4.18 shows an output, where the user has initially limited the feedback to a depth of 3, and then selectively expanded the **span_begin** up to the p-chain level.

The initial behavioral description is coarse. It does not capture many of the constraints that should be placed on the behavior to verify its correctness. So as before, we used the spreadsheet interface of Ave to verify certain assertions about the particular execution of the program.

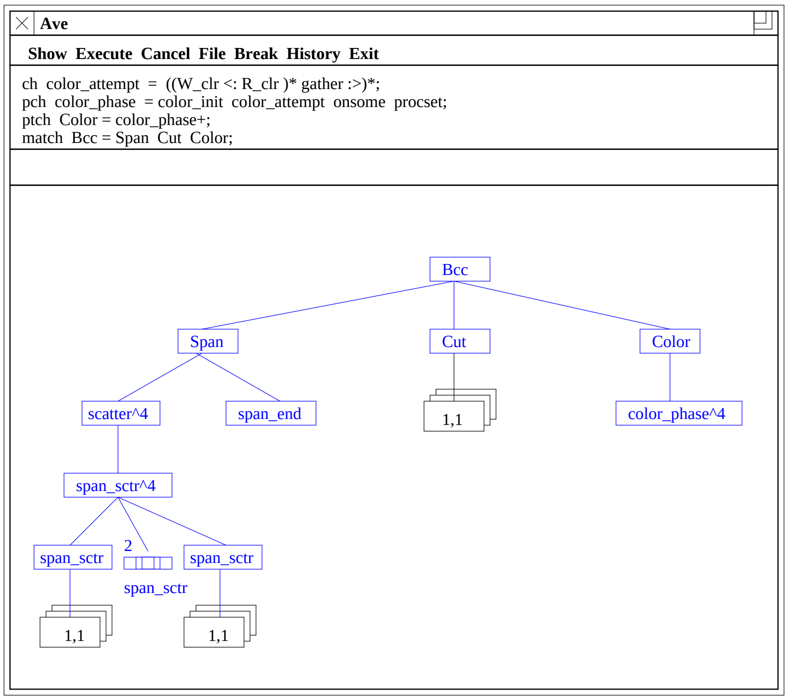


Figure 4.18. Ave output on biconnected component algorithm (with user directed scaling).

1. Except for the root, each node in the network receive at least one **sprd** message.

2. Except for the root, each vertex in the spanning tree has only one parent edge.

3. No node sends a **sprd** message to its parent in the first phase.

4. Each **pre** message sent by a node to its children in the spanning tree has different contents, while the **pre** messages sent along the non-tree edges have the same content.

5. Successful coloring computations take place along disjoint components.

To verify that each node in the network except the root received at least one **sprd** message, we issued the following query.

```
with Span foreach Span compute set node = pid(R_sprd);
with Span foreach Span compute left_out = procset difference node(Span);
with Span print left_out(Span) wrt Span;
```

where **difference** is a set operator that gives the difference of two sets. Since **procset** contains all the process identifiers, **left_out** will contain identifiers of the processes that have not received any **sprd** messages. This produced the output shown below, as expected, identifying process 1 to be the only process not to have received a **sprd** message in the first phase.

Abstract Event	left_out
Span	1

Since *parent-edges* are identified in the first phase, by the sending of **gthr** messages, we can verify the second assertion with the following

```
with Span foreach proc compute parent_id = pid(R_gthr);
with Span foreach proc compute parent_count = sizeof(parent_id);
with Span foreach show parent_count(proc) wrt proc;
```

The output of the query is shown in Figure 4.19(a), showing that each vertex (except the root) has indeed only one parent edge.

Since nodes disregard all but the first **sprd** messages, sending of such messages does not affect the correctness of the algorithm, but may, in a network application present an undue load on the network. The third assertion – that there are no extraneous **sprd** messages – is thus related to the performance of the program and can be verified with the following:

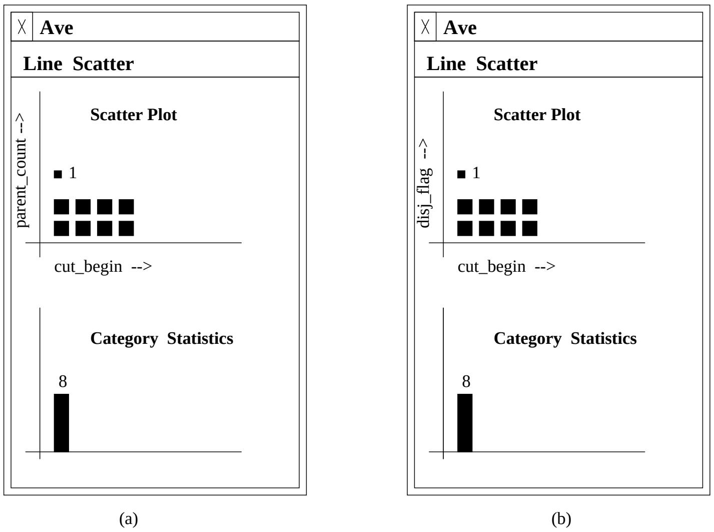


Figure 4.19. Ave Scatter Plot in the Biconneceted Components algorithm. It shows: the spanning tree was built correctly in the first phase (a); no extraneous messages were sent in the first phase (b).

```
with Span foreach proc compute set p_rcvr = pid(W_sprd);
with Span foreach proc compute disj_flag = disjoint(parent_id, p_rcvr);
with Span foreach show disj_flag(proc) wrt proc;
```

The result of the query is shown in Figure 4.19(b), again showing that the assertion is satisfied in this particular execution of the program.

Each parent, in the second phase, computes the pre-order number for its children, and thus messages sent to different children should have distinct content. In addition, each node sends its own *pre-order* number to all the nodes connected to it by non-tree edges. In our modeling of the `Cut` phase, however, we did not differentiate between the messages sent to child-nodes and non-child nodes. In order to verify the fourth assertion we need to somehow relate the `pre` messages sent in the second phase, with the *parent-* and the *child-* edges identified in the first phase. To facilitate exploring a behavior from different viewpoints, Ariadne allows matching of a different behavior with respect to a subset of events constituting an already matched abstract event. Both Ariadne's filtering mechanism and its facility of imposing multiple viewpoints are required to verify the two assertions.

Initially we were interested in modeling the depth-first traversal behavior of the `pre` messages, but for this fourth assertion we need to group all the messages sent by a node in the `Cut` phase together. We had logged the contents of the messages during the initial execution, so we could use Ariadne's filtering mechanism to accomplish the grouping. In the `span_end` phase each node receives `gthr` messages from its children and sends a `gthr` message to its parent. In the `Cut` phase, each node receives a `pre` message from its parent and sends `pre` message to its children and other non-child neighbors: the children respond with `gthr` messages but the non-child neighbors do not. For the assertion at hand, the interactions with the parent are irrelevant. The following chain level coarse behavior description illustrates how a simple mechanism can filter out the irrelevant events.

```
ch pre_model = R_gthr* W_pre* wrt [R_gthr, W_pre];
```

where the `wrt` clause specifies that while matching the chain definition only the `R_gthr` and `W_pre` events are to be considered, which filters out all interactions with a node's parent. We now have to ensure that this chain be matched only against those events belonging to either `span_end` or `Cut` abstract events. We do it using the `with` clause introduced in the last example. The following description accomplishes the task.

```
ch all_msg = R_gthr* W_pre* wrt [R_gthr, W_pre];
pch pre_msg = all_msg onsome procset;
match pre_order = pre_msg with (Bcc wrt [span_end, cut]);
```

`Bcc wrt [span_end, Cut]` returns only those primitive events in `bcc` of the abstract events `span_end` and the abstract event `Cut`. The `with` clause restores the returned events to the trace, restricts the matching to the returned events, and once the matching is done removes the returned events from the trace.

This illustrates Ariadne's filtering capability at two different levels of the structural hierarchy. The `wrt` clause at the chain level depicts a simple filtering scheme where the user selects a subset of the primitive events for modeling; while matching in presence of the `with` clause can be thought of as imposition of a different view on an abstract event, since we match a different model on a part of an already matched behavior. The Ave feedback is shown in Figure 4.20.

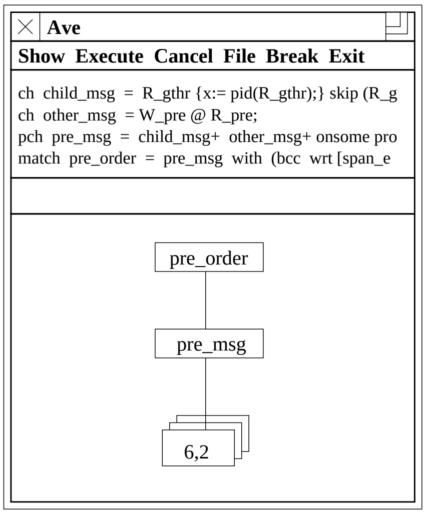


Figure 4.20. New match tree in the biconnected component algorithm.

Differentiating between the messages sent to child and non-child nodes is now easy. We can now identify each `W_pre` event that corresponds to a message sent to a child node by first computing the set of children for each process, and then comparing the oid attribute of the W_pre event to verify whether it belong to the set of children.

```
with pre_order foreach proc compute child_set = oid(R_gthr);
with pre_order::proc foreach W_pre compute is_child
    = disjoint(child_set(proc), oid(W_pre));
```

`::` is the scope resolution operator. We cannot write **with proc**, since there may be multiple match variables active in a session, each containing abstract events named proc. pre_order::proc returns a list of proc nodes that are part of the match variable named pre_order, and the rest of the statement is evaluated in the context of each element of that list. Now it is simple to identify the messages sent to children nodes (those with is_child = 1) as child_msgs, and the messages sent to non-child nodes (is_child = 0) as other_msgs with the following commands:

```
with pre_order foreach W_pre { eq(is_child(W_pre),1); }
      add marker child_msg;
with pre_order foreach W_pre { eq(is_child(W_pre),0); }
      add marker other_msg;
```

The above commands introduce the constraint based event selection mechanism of the query language. { eq(is_child(W_pre), 0) } represents the constraint that determines which W_pre events are selected to be marked as child_msgs. The markers child_msg and other_msg can now act as aliases for the selected events. Needless to say add and marker are keywords in the modeling language.

The new modeling makes the verification of the assertions straight forward.

```
with pre_order foreach child_msg
      compute value = value(W_pre);
with pre_order foreach proc
      compute value_flag = distinct(value(child_msg));
with pre_order show value_flag(proc) wrt pre_order;
```

This produces the output shown in Figure 4.21(a). As can be seen, each message sent to a *child edge* had different contents.

The assertion about the messages sent through the non-tree edges can be verified similarly

```
with Pre foreach other_pre compute value = value(W_pre);
with pre_order foreach proc compute
    other_flag = equal(value(other_msg));
with pre_order show other_flag(proc) wrt pre_order;
```

The output is shown in Figure 4.21(b). This time too the scatter plot has the value 1, saying all message contents are equal.

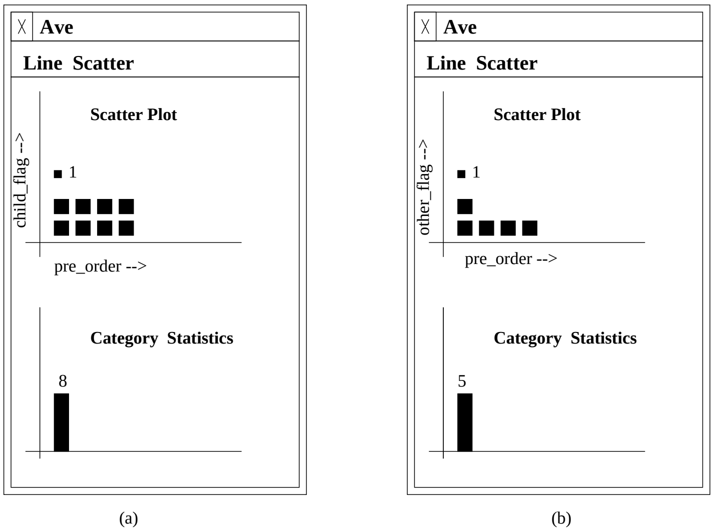


Figure 4.21. Ave scatter plot in the Biconnected Components algorithm. It shows: the pre_ordering computed for nodes are distinct (a); all non-child neighbors are informed consistently.

The behavioral description of the Color phase does not group events that happen in the context of a single biconnected component. In order to verify the final assertion, we need to differentiate the successful color computations from the unsuccessful ones. Successful `color_attempts` have all their `gthr` messages carrying a value 1, while unsuccessful attempts consist of at least one **gthr** message with a value 0. This is very hard to specify in the initial pattern description, but can be done easily with the query language by using the feature of partitioning a behavior into equivalence classes.

```
with Bcc foreach proc compute succ = value(W_gthr);
with Bcc foreach proc compute succ = eq(succ(response),1);
with Bcc foreach color_phase create partition succ wrt succ(proc);
show Bcc through succ;
```

which produces the output shown in Figure 4.22.

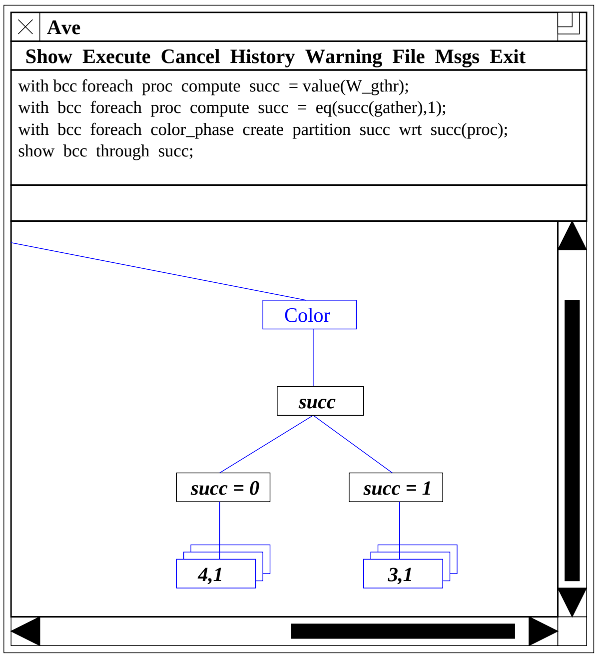


Figure 4.22. Visualizing the match tree through user-defined partition.

It can be seen from the Ave feedback, that our algorithm found 3 biconnected components, as the node **succ = 1** has three children. We can now compute the list of links spanned by each of the child and see whether the lists are disjoint or not.

Selecting the `succ=1` partition, we issued the following commands.

```
with select foreach W_color
      compute link_tuple = (pid(W_clr), oid(W_clr));
with select foreach proc
      compute link_set = link_tuple(W_clr);
with select assert disjoint(link_set(proc));
```

which printed the following:

```
Assertion failed.
link_set[2] and link_set[3] have the following elements in common:
     (1,3), (3,1)
```

To see what color computations are in conflict, we issued the following command.

```
with select print color(proc), pid(proc) wrt proc;
```

which printed the following:

Abstract Event	color	pid
	9	2
succ = 1	7	3
	6	3

So biconnected components colored with label 6 and 7 actually had a common link (1,3). The `pid` field values tell us that the successful color computations were initiated by the processes 2 and 3. In the next chapter, we will revisit this example to show how Ave/Ariadne can be used to examine processes 1 and 3 in a very selective manner to track down this particular bug.

4.3 Conclusions

Modeling languages for event-based debugging have always erred on the side of expressivity at the expense of blurring the detail of error feedback. This has made modeling languages unnecessarily complex in their implementation and in their semantics. Also most of the modeling languages did not incorporate the right notion of logical time. We have followed the other road, our modeling language is simple and concise. But the loss of expressivity has been compensated by a sophisticated notion of temporal relationships and filtering based on logical time, and a scalable visual feedback facility. In this chapter, we presented sample debugging sessions in the context of seven real-world parallel applications where the behaviors can be modeled successfully using our modeling language and explored using its companion query interface. We have shown, how with the aid of the query language, we can provide the user with the control over the scalability/precision ratio of the feedback. Ave/Ariadne allows for initial coarse grain behavioral specification of a complex program, and can focus the user attention to the offending part of the behavior. Often errors are detected in terms of anomalous relationship between abstract events, which precludes us from pointing out the source code construct responsible for the error. How are we to shift our attention from the event domain to the domain of source code? In the next chapter we will discuss how the duality of events and states can be exploited to allow the user to set breakpoints in terms of abstract events and halt the computation in the earliest consistent state that is meaningful in the context of the detected error.

Chapter 5

Integration of Event- and State-Based Debugging

To effectively use break/examine techniques on massively parallel programs, it is necessary to stop those programs in meaningful states, that is, states that make sense within the context of the current debugging session. Standard mechanisms that set breakpoints in terms of program locations or combinations of local state predicates are problematic: How are the breakpoints to be specified across multiple processes? Must they be globally consistent or are local breakpoints sufficient? If local, how and when should the remaining processes be stopped in order to provide a meaningful state? One possibility is to allow breakpoints only at barrier synchronizations [66] but this is overly constraining in multithreaded language environments. Alternatively, local breakpoints can be extended dynamically to global breakpoints [35, 15, 61]. This can be done explicitly by notifying the remaining processes – either through communication with a central monitor [61] or over normal communication channels [35, 15] – that a breakpoint has been reached or it can be done implicitly by allowing the remaining processes to run until they block waiting for input. Unfortunately, in either case, it is possible for some processes to execute well beyond the established breakpoint, potentially obliterating information relevant to the source of the error. We choose instead to take advantage of the replay mechanism: we use the results of *post mortem* modeling to insert directives that will, during replay, halt each process at the last point where it could have influenced the process(es) triggering the breakpoint.

Section 5.1 gives an informal overview of the abstract-event based breakpointing scheme, the actual algorithm and its proof of correctness, and presents a sample debugging session to show how the integrated methodology can be used to track a bug. The binary image compression example is continued in Section 5.2. Section 5.3 gives another complete example, where the integrated debugging methodology is used to detect and locate a bug which would be extremely difficult to do with either method alone. Section 5.4 presents three more debugging examples that showcase the integrated debugging environment.

5.1 Setting of Consistent Global Breakpoints Based on Abstract Events

Manabe and Imase [58] also use a replay-based approach, allowing breakpoints to be to specified as conjunctions and disjunctions of predicates on local states. It may, however, be difficult to determine the appropriate predicates: errors observable only as incorrect output are often hard to relate back to individual, local predicates in specific processes. Instead, our Ariadne/*ipd* debugger allows the user to specify breakpoints in terms of the same abstract events that he/she is already using for behavioral modeling.

During event-based behavioral modeling, user-specified models of intended program behavior are matched against the actual program behavior as captured in event traces. Within Ariadne, abstract events are sets of primitive events grouped by logical, temporal relationships. Breakpoints can be set immediately before or after a matched or partially matched abstract event. Once set, the debugger annotates the trace and then reexecutes the program to the latest *consistent* state preceding those annotations. Often, this allows the programmer to easily stop his/her program at states that would be difficult to specify using local state predicates. For example, the program could be stopped at the first point in an execution where two processes see a sequence of broadcast operations in a different order, or at a point where all of the processes reached by a specific spreading activation have sent replies, or at the beginning of the first phase in which some processes communicate over different distances.

Setting a local breakpoint based on an event is straightforward – we can set the breakpoint immediately before the local event occurrence or we can set it immediately after the local event occurrence. In the first case, the breakpoint state is one terminated by the specific event, and in the latter the breakpoint state is the one initiated by the specific event. To set a breakpoint based on an abstract event, we partition the processes of a computation into three equivalence classes: the *core set* containing those that executed parts of the abstract event; the *influence set* containing those that could have causally affected a component of the abstract event; and the *other set* containing those that remain. We define the *core*, *influence* and *other* sets of an abstract event A formally as:

1. The *core set* of an abstract event A is defined as:

   (a) $\forall\ c \in \mathit{core(A)},\ \exists\ a \in A$, such that $c = \mathit{pid(a)}$

   (b) $\forall\ a \in A,\ \mathit{pid(a)} \in \mathit{core(A)}$

2. The *influence set* of an abstract event A is defined as:

   (a) $\mathit{influence(A)} \cap \mathit{core(A)} = \emptyset$, and,

   (b) $\forall\ i \in \mathit{influence(A)},\ \exists$ an event e such that $\mathit{pid(e)} = i$, and $e \rightarrow A$.

(c) $\forall$ *event* e, such that $pid(e) \notin core(A)$, and $e \rightarrow A$, $pid(e) \in influence(A)$.

3. The *other set* of an abstract event A is defined as:

   (a) $other(A) \cap core(A) = \emptyset$

   (b) $other(A) \cap influence(A) = \emptyset$

In order to stop a computation, the state most relevant to the abstract event is restored as follows:

1. processes in the *other* set are stopped at their earliest state,

2. processes in the *influence* set are stopped immediately after the latest event that could causally influence the abstract event, and

3. processes in the *core* set are stopped immediately before (after) the earliest (latest) local event that is a part of the abstract event, depending on the user request.

In Figure 5.1, the abstract event is formed by the events within the thick dashed line. Processes 4,5,6 form the *core* set; 2,3 and 7 form the *influence* set, and 0 and 1 form the *other* set.

In some cases, this procedure results in a state that is not *consistent*, that is, a state that could not have occurred in an actual execution. We define consistency as in [33], letting $e_{i,k}$ be the *k-th* event on process i, which gives rise to $s_{i,k}$ the *k-th* state on process i, and signals the termination of $s_{i,k-1}$ the *(k-1)-th* state on process i. Alternatively, we will refer to these states as $s_s(e_{i,k})$ and $s_t(e_{i,k})$ respectively, when we can identify the occurrence of a particular event, but not the number event occurrences on a process. Denoting Lamport's *happened before* relation on events by

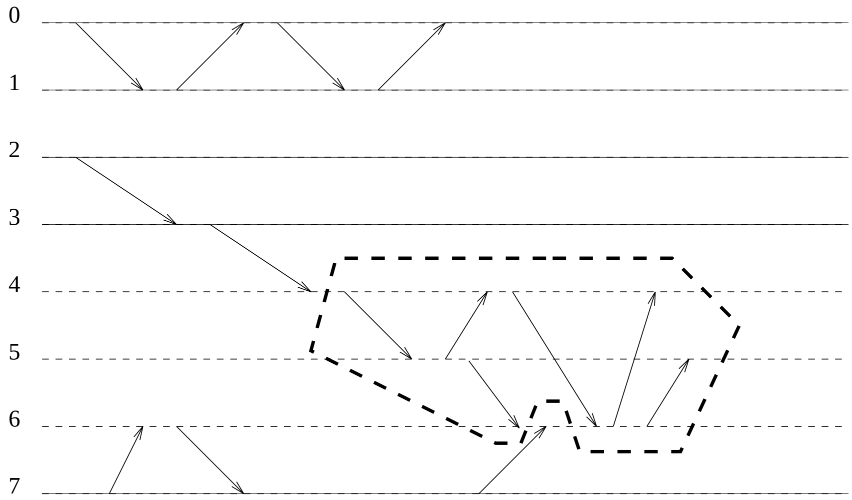


Figure 5.1. Partitioning process set from the perspective of an abstract event (shown with thick dashed line).

$\rightarrow$, we define the analogous partial ordering of process states as follows: $s_{i,k} \rightarrow s_{j,l}$ iff

$$e_{i,k+1} \rightarrow e_{j,l} \quad \text{or} \quad e_{i,k+1} = e_{j,l}$$

Thus state a precedes a state b, if the terminating event of a *happens before* or is the same as the starting event of b. Local states that are incomparable under this relation are said to be concurrent (denoted by $||$).

Definitions.

1. A global state for an n process system, denoted by a n-tuple $< s_{1,k_1}, s_{2,k_2}, \ldots s_{n,k_n} >$ of local states, is **consistent** if its elements are pairwise concurrent.

2. A global state for an n process system, denoted by a n-tuple $< s_{1,k_1}, s_{2,k_2}, \ldots s_{n,k_n} >$ of local states, is **inconsistent** if not all of its elements are pairwise concurrent.

3. A *projection* of a global state G on a set of processes p denoted by *proj(G,p)* and is defined as:

   $proj(G,p) = \{s_{i,k_i} \textit{ such that } i \in p\ \}$, where $G = < s_{1,k_1}, s_{2,k_2}, ...s_{n,k_n} >$.

   The concept of *projection* will be used to select parts of global states to reason about state consistency later in the chapter.

Whenever a user's breakpointing request would result in an inconsistent state, our debugger automatically constructs an earlier, consistent state using a greedy algorithm. Consider, for example, the behavior depicted in Figure 3.9. Having been informed that the expected **precedes** relation did not hold between the second and third iterations, the user might insert a breakpoint after the second one, intending to stop the processes at the locations indicated by the rectangles in Figure 5.2 but this state is meaningless. It could not have happened in any execution of the system because it shows the receipt of a message by process **0** that has not yet been sent by process **8**. This state is *inconsistent.* Our greedy algorithm adjusts this inconsistent state to construct an earlier consistent state by including earlier local breakpoints (denoted by the oval markers in Figure 5.2). This is the *closest* consistent state that preceded the inconsistent state.

The user can set multiple breakpoints by selecting multiple abstract events as long as the breakpoints do not conflict. Two breakpoints conflict if they are reached (in the trace) by different processes in different orders. Conflicting breakpoints would deadlock execution replay and so must be avoided.

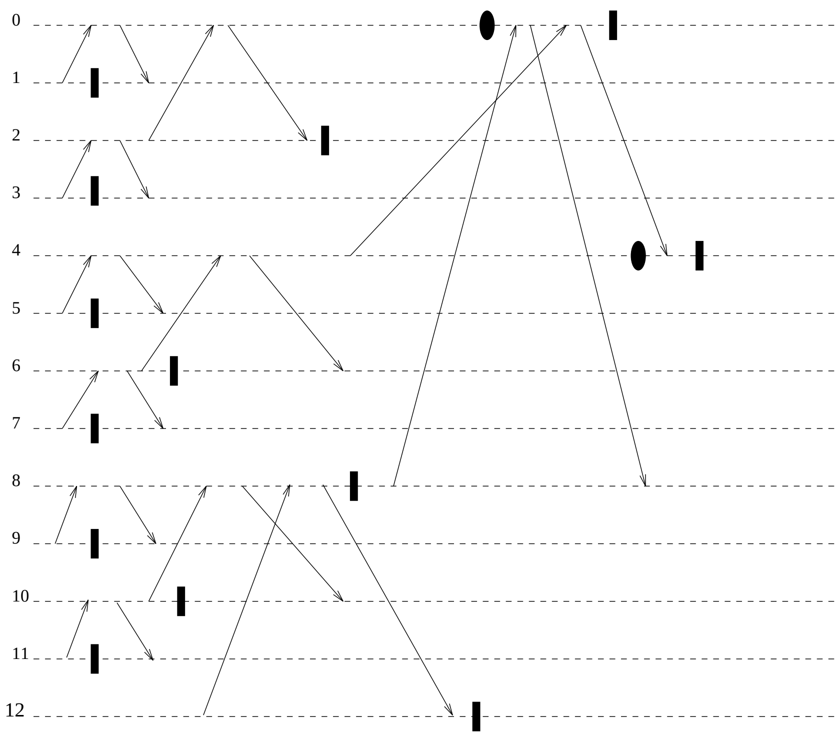


Figure 5.2. Depiction of the execution history graph of the Binary Image Compression example, illustrating Ave/Ariadne's handling of global breakpoint state. The rectangles indicate the inconsistent, global breakpoint specified by the user and the ovals indicate the automatic readjustment needed to establish a consistent state.

Definitions. Given two global states A and B,

1. *follow_set(A,B)* is the set of A's local states that happen after the corresponding local states of B.

   Observations:

   (a) $follow\_set(B, A) \neq follow\_set(A, B)$.

   (b) $follow\_set(A, B) \subset A$

   (c) $follow\_set(B, A) \subset B$

2. *identical_set(A,B)* is the set of A's local states that are identical to the

corresponding local states of B.

Observations:

(a) $identical\_set(B, A) = identical\_set(A, B)$.

(b) $identical\_set(A, B) \subset A$

(c) $identical\_set(A, B) \subset B$

We extend the notion of *precedes* between two local states to consistent global states. There is no notion of *concurrency* between two completely enumerated consistent global states, but we capture the *overlaps* relationship between consistent global states that is relevant in the context of setting global breakpoints.

1. A **precedes** B iff there is at least one local states a_{i,k_i} in A and b_{i,l_i} in B such that $a_{i,k_i} \rightarrow b_{i,l_i}$ and there is no state a_{j,k_j} in A and b_{j,l_j} in B such that $b_{j,l_j} \rightarrow a_{j,k_j}$, $i \neq j$.

   In other words, A **precedes** B, iff *follow_set(A,B)* $= \emptyset$, and *follow_set(B,A)* $\neq \emptyset$.

   Figure 5.3 shows an example of one global state preceding another.

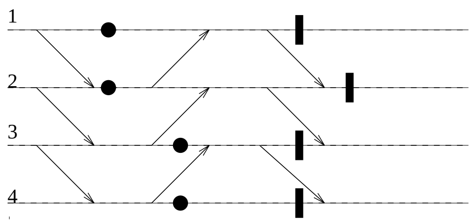


Figure 5.3. Illustrating *precedes* relationship between global states. The global state enumerated by the filled circles precedes that enumerated by the filled rectangles.

2. A and B **conflict** iff there are local states a_{i,k_i}, a_{j,k_j} in A or b_{i,l_i}, b_{j,l_j} in B such that $a_{i,k_i} \rightarrow b_{i,l_i}$ and $b_{j,l_j} \rightarrow a_{j,k_j}$, $i \neq j$

In other words, A and B **conflict** iff *follow_set(A,B)* $\neq \emptyset$ and *follow_set(B,A)* $\neq \emptyset$.

Figure 5.4 shows an example of two consistent global states that conflict.

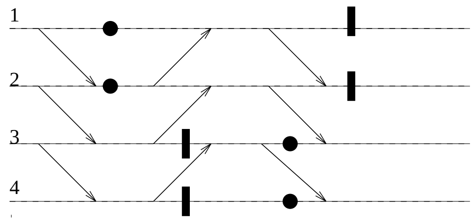


Figure 5.4. Illustrating *conflict* relationship between global states. The global state enumerated by the filled circles and that enumerated by the filled rectangles *conflict.*

Lemma 1. If there exists states a_{i,k_m} and a_{i,k_n} such that $a_{i,k_m} \rightarrow a_{i,k_n}$. Then the following two conditions hold for another state b_{j,l_p}

1. $a_{i,k_m} \mid\mid b_{j,l_p}$ implies $a_{i,k_n} \not\rightarrow b_{j,l_p}$

2. $a_{i,k_n} \mid\mid b_{j,l_p}$ implies $b_{j,l_p} \not\rightarrow a_{i,k_m}$

In other words, for a pair of concurrent states a_{i,k_m} on process i and b_{j,l_p} on process j, no state on process i that happens after a_{i,k_m} can precede b_{j,l_p}, and b_{j,l_p} cannot precede any state on process i that precedes a_{i,k_m}.

Proof:

1. $a_{i,k_m} \rightarrow a_{i,k_n}$, and $a_{i,k_m} \mid\mid b_{j,l_p}$.

   If $a_{i,k_n} \rightarrow b_{j,l_p}$, then by transitivity, $a_{i,k_m} \rightarrow b_{j,l_p}$.

   Hence, $a_{i,k_n} \not\rightarrow b_{j,l_p}$.

2. $a_{i,k_m} \rightarrow a_{i,k_n}$, and $a_{i,k_n} \mid\mid b_{j,l_p}$.

   If $b_{j,l_p} \rightarrow a_{i,k_m}$, then by transitivity, $b_{j,l_p} \rightarrow a_{i,k_n}$.

   Hence, $b_{j,l_p} \not\rightarrow a_{i,k_m}$.

Lemma 2: Given two distinct consistent states, the consistent state formed by considering the pairwise latest state of the local states is also consistent, where the *latest* of two states (s_{i,k_m}, s_{i,k_n}) is defined to be s_{i,k_n} iff $s_{i,k_m} \rightarrow s_{i,k_n}$ or $m = n$.

Proof: Let,

$$\text{A} = < a_{1,k_1}, a_{2,k_2}, ...a_{n,k_n} > \text{ and } \text{B} = < b_{1,l_1}, b_{2,l_2}, ...b_{n,l_n} >,$$

be two consistent global states. Either A precedes B, or, A and B conflict.

The state C defined by taking the latest of the pairwise local states of the two states can be written as

$$\text{C} = \{ \textit{follow\_set(A,B), follow\_set(B,A), identical\_set(A,B)} \}$$

Case 1. A precedes B:

Claim: $\forall(f, g) \in C, f \,||\, g$

Since, follow_set(A,B) = $\emptyset$, *follow_set(B,A)* $\subset$ *B*, *identical_set(B,A)* $\subset$ *B*, and B is a consistent global state.

Case 2. A and B conflict:

1. *Claim:* $\forall(f, g)$ *such that* $f \in$ *follow_set(A,B)*, $g \in$ *identical(A,B)*, $f \,||\, g$.

   Because,

   follow_set(A,B) $\subset$ *A*, *identical_set(B,A)* $\subset$ *A*, and A is a consistent global state.

2. *Claim:* $\forall(f, g)$ *such that* $f \in$ *follow_set(B,A)*, $g \in$ *identical_set(A,B)*, $f \,||\, g$.

   Likewise.

3. *Claim:* $\forall(f,g)$ *such that* $f \in$ *follow_set(B,A)*, $g \in$ *follow_set(B,A)*, $f \,||\, g$.

   Let $a_{i,k_i} \in$ *follow_set(A,B)*, and $b_{j,l_j} \in$ *follow_set(B,A)*. By definition, there must exist events a_{j,k_j} in A and b_{i,l_i} in B such that $b_{i,l_i} \rightarrow a_{i,k_i}$ and $a_{j,k_j} \rightarrow b_{j,l_j}$.

   (a) $b_{i,l_i} \rightarrow a_{i,k_i}$, and b_{i,l_i} and b_{j,l_j} are pairwise concurrent.

   Hence from Lemma 1,

   $a_{i,k_i} \not\rightarrow b_{j,l_j}$

   (b) $a_{j,k_j} \rightarrow b_{j,l_j}$, and a_{i,k_i} and a_{j,k_j} are pairwise concurrent.

   Hence from Lemma 1,

   $b_{j,l_j} \not\rightarrow a_{i,k_i}$

   $a_{i,k_i} \,||\, b_{j,l_j}$.

   Since both the events are chosen arbitrarily from the sets *follow_set(A,B)* and *follow_set(B,A)*, the claim is true for all such pairs of events.

So, $\forall(f,g) \in C$, $f||g$. C is consistent. (QED).

The most significant contribution of Ariadne's breakpoint scheme is its automatic readjustment of inconsistent breakpoints to the *closest* consistent breakpoint that precedes the original inconsistent global state. In order to formalize the concept of *closeness*, we need to introduce the concept of *event_distance* between states.

Definition. The *event_distance* between two local states on the same process, is the number of events that occur to take the process from one state to the other.

$$\text{event\_distance}(s_{i,k_m},\, s_{i,k_n}) = |m - n|$$

Definition. Given two global states A and B, such that A and B do not conflict, the *event distance* between A and B is the sum of the *event distances* between the pairwise local states on the same processes.

If,

A = $< a_{1,k_1}, a_{2,k_2}, ..., a_{n,k_n} >$, and, B = $< b_{1,l_1}, b_{2,l_2}, ..., b_{n,l_n} >$

then

$$\mathit{event\_distance(A,B)} = \textstyle\sum_{i=1}^{n} \mathit{event\_distance}(a_{i,k_i}, b_{i,l_i})$$

Lemma 3: Given three global states A, B and C, such that $A \rightarrow B \rightarrow C$,

$$\mathit{event\_distance(A,C)} = \mathit{event\_distance(A,B)} + \mathit{event\_distance(B,C)} \quad .$$

Proof:

Assume

A = $< a_{1,k_1}, a_{2,k_2}, ..., a_{n,k_n} >$, B = $< b_{1,l_1}, b_{2,l_2}, ..., b_{n,l_n} >$, and

C = $< c_{1,m_1}, c_{2,m_2}, ..., c_{n,m_n} >$

$$\begin{aligned}
\mathit{event_distance(A,C)} &= \textstyle\sum_{i=1}^{n} \text{event_distance}(a_{i,k_i}, c_{i,m_i}) \\
&= (m_1 - k_1) + (m_2 - k_2) + \dots + (m_n - k_n) \\
&= (m_1 - l_1) + (l_1 - k_1) + (m_2 - l_2) + (l_2 - k_2) + \dots \\
&\quad + (m_n - l_n) + (l_n - k_n) \\
&= (m_1 - l_1) + (m_2 - l_2) + \dots + (m_n - l_n) + \\
&\quad (l_1 - k_1) + (l_2 - k_2) + \dots + (l_n - k_n) \\
&= \textstyle\sum_{i=1}^{n} \text{event_distance}(c_{i,m_i}, b_{i,l_i}) + \\
&\quad \textstyle\sum_{i=1}^{n} \text{event_distance}(b_{i,l_i}, a_{i,k_i}) \\
&= \text{event_distance(B,C)} + \text{event_distance(A,B)}
\end{aligned}$$

5.1.1 Breakpoint Scheme and Formal Definitions

Ariadne allows the user to set a breakpoint immediately before or immediately after an abstract event has occurred, and stops each process such that the global state of the computation is consistent. We introduce the concepts of *early* and *late* sets to identify the initiation and the termination of an abstract event, and the concept of *sunrise* to identify the latest events on processes that could have causally influenced an abstract event.

Definitions.

1. The *early* set of an abstract event A identifies the earliest events belonging to A for each process identifiers in *core(A)* and can be defined as:

   $early(A) = \{e \mid e \in A, \text{ and } \forall\ a \in A,\ a \neq e \Rightarrow a \not\rightarrow e\ \}$

2. The *late* set of an abstract event A identifies the latest events belonging to A for each process identifiers in *core(A)*, and can be defined as:

   $late(A) = \{l \mid l \in A, \text{ and } \forall\ a \in A,\ a \neq l \Rightarrow l \not\rightarrow a\ \}$

3. The **sunrise** of an abstract event A on a set of processes P, *sunrise(A,P)* is the latest set of events on process set P that precedes the abstract event A, and can be defined as:

   (a) $core(sunrise(A,P)) = P$

   (b) $\forall\ e,\ f \in sunrise(A,P)$,

   if e $\neq$ f then, $pid(e) \neq pid(f)$

   (c) $sunrise(A,P) \cap A = \emptyset$,

   (d) $sunrise(A, other(A)) = undefined$

   (e) $sunrise(A,P)$ *precedes* A if $P \cap other(A) = \emptyset$

   The sunrise of a primitive event can be defined as a special case of the above definition.

Figure 5.5 shows the *early* and *late* sets and the *sunrise* of an abstract event (shown in thick dashed line).

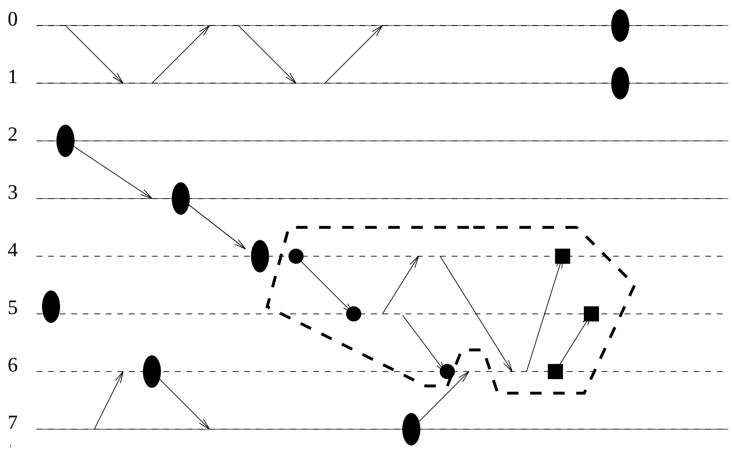


Figure 5.5. Depiction of early, late, and sunrise sets for an abstract event. The early set (shown with filled circles), late set (shown with filled rectangles), and the sunrise (shown with filled ovals) are shown for an abstract event (shown with thick dashed line).

Breakpoint Scheme: Let $s_s(e_{i,k})$ be the state initiated by the occurrence of $e_{i,k}$, the *k-th* event on process i, and $s_t(e_{i,k})$ be the state terminated by it. We can define the state S_B where the computation will be halted when the user stops the computation before or after an abstract event A, in terms of a "first approximation" $S_{tentative}$:

1. If the breakpoint is set before the occurrence of A, we stop the computation for the processes in the core set immediately before the occurrence of A which is captured by the events in *early(A)*. Processes in the *influence* set are stopped immediately after the occurrence of the latest event that could have preceded $early(A)$ (Stopping the computation before that may result in an inconsistent global state, stopping it after that may lose relevant state information). Since by definition, events on processes *other(A)* cannot influence A, it does not matter where those processes are stopped, but to keep the response time low they are stopped at their initial states. The user request is translated to a global state $S_{tentative}$. $S_{tentative}$ is simply "our first

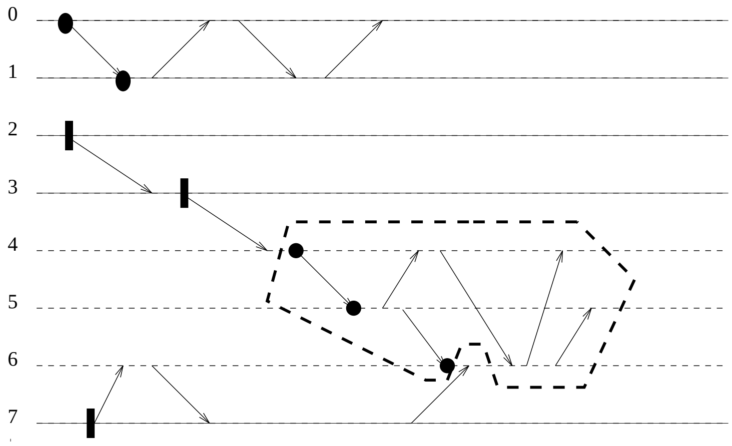


Figure 5.6. Global breakpoint scheme for setting a breakpoint before an abstract event. Local breakpoints for user request of setting a breakpoint immediately before the occurrence of an abstract event (shown with thick dashed line). The processes in the *influence* set are stopped immediately after the occurrence of the events shown with filled rectangles, and the processes in the *other* set are stopped immediately before the occurrence of the events shown with filled ellipses.

approximation" to the global breakpoint. If $S_{tentative}$ is consistent we have our global breakpoint; otherwise we have to find the closest consistent global state preceding $S_{tentative}$ which will become the readjusted global breakpoint.

proj($S_{tentative}$, *core*(*A*)) = states terminated by *early*(*A*)

proj($S_{tentative}$, *influence*(*A*)) = states initiated by *sunrise*(*early*(*A*), *influence*(*A*))

proj($S_{tentative}$, *other*(*A*)) = Initial states of the processes.

Figure 5.6 shows where local breakpoints would be set by our algorithm if the user requests a global breakpoint to be set before the abstract event delineated by dashed line.

2. If the breakpoint is set after the occurrence of A, we stop the computation for the processes in the core set immediately after the occurrence of A which is captured by the events in *late(A)*. Processes in the *influence* set are stopped immediately after the occurrence of the latest event that could have preceded $late(A)$. The processes in the *other* set are again stopped at their initial states. We define $S_{tentative}$ as

   proj($S_{tentative}$*, core(A))* = states initiated by *late(A)*

   proj($S_{tentative}$*, influence(A))* = states initiated by *sunrise(late(A), influence(A))*

   proj($S_{tentative}$*, other(A))* = Initial states of processes

   Figure 5.7 shows where local breakpoints would be set by our algorithm if the user requests a global breakpoint to be set after the abstract event delineated by dashed line.

If $S_{tentative}$ is consistent, then $S_B = S_{tentative}$, otherwise, S_B is the consistent state preceding $S_{tentative}$ that with the minimum event distance between it and $S_{tentative}$.

1. *Claim: There always exists a consistent state that precedes an inconsistent* $S_{tentative}$.

   The initial global state of the computation (before the occurrence of any event) is always consistent and it precedes any other global state of the computation.

2. *Claim: There is a unique consistent state* S_B *that precedes and has the minimum* event distance *from* $S_{tentative}$.

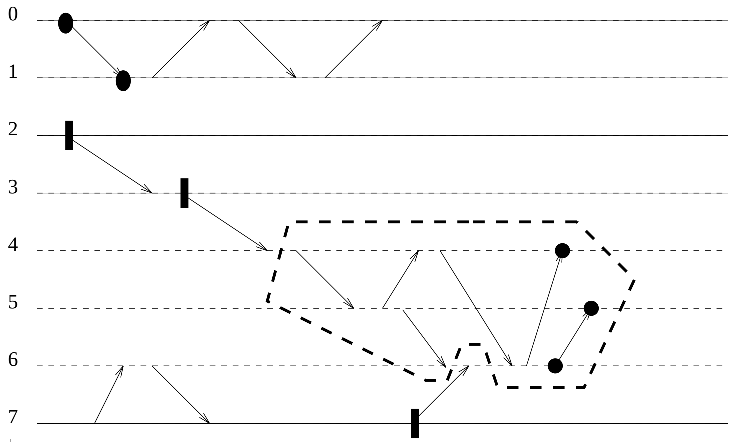


Figure 5.7. Global breakpoint scheme for setting a breakpoint before an abstract event. Local breakpoints for user request of setting a breakpoint immediately after the occurrence of an abstract event (shown with thick dashed line). The processes in the *influence* set are stopped immediately after the occurrence of the events shown with filled rectangles, and the processes in the *other* set are stopped immediately before the occurrence of the events shown with filled ellipses.

Assume, this is wrong. So there must exist another consistent state $S'_B \neq S_B$ such that

$S'_B \rightarrow S_{tentative}$, and $event\_distance(S_B, S_{tentative}) = event\_distance(S'_B, S_{tentative})$

By our own definition, $S_B \not\rightarrow S'_B$.

Because, from Lemma 3, this would mean

$event\_distance(S_B,\ S_{tentative}) = event\_distance(S_B,\ S'_B) + event\_distance(S'_B,\ S_{tentative})$.

This would contradict our assumption that S_B and S'_B are equidistant from $S_{tentative}$.

Likewise, $S'_B \not\rightarrow S_B$.

Since the states are distinct, S_B and S'_B must conflict.

According to Lemma 2, $S_g = \text{latest}(S_B,\ S'_B)$ is consistent.

$S_g = \{\ \textit{follow\_set}(S_B,\ S'_B),\ \textit{follow\_set}(S'_B, S_B),\ \textit{identical\_set}(S_B,\ S'_B)\ \}$

$\textit{follow\_set}(S_B,\ S'_B) \subset S_B \Longrightarrow \textit{follow\_set}(S_B,\ S'_B) \rightarrow S_{tentative}$.

$\textit{follow\_set}(S'_B,\ S_B) \subset S'_B \Longrightarrow \textit{follow\_set}(S'_B,\ S_B) \rightarrow S_{tentative}$.

$\textit{identical\_set}(S_B,\ S'_B) \subset S_B \Longrightarrow \textit{identical\_set}(S_B,\ S'_B) \rightarrow S_{tentative}$.

Hence, $S_g \rightarrow S_{tentative}$.

Since, S_B, $S'_B \rightarrow S_g \rightarrow S_{tentative}$, from Lemma 3,

$\textit{event\_distance}(S_g,\ S_{tentative}) < \textit{event\_distance}(S_B,\ S_{tentative})$

which contradicts our initial assumption that S_B and S'_B are the states with the minimum event_distance with $S_{tentative}$ that precedes it.

So S_B must be unique.

3. *Claim: If the states in proj($S_{tentative}$, core(A)) are pairwise concurrent, then $S_{tentative}$ is consistent.*

Assume the statement is false: $S_{tentative}$ can be inconsistent even if states in *proj*($S_{tentative}$, *core(A)*) are pairwise concurrent.

Processes belonging to the set *other(A)* are always stopped at their initial states, before they interact with any other process in the system. Hence, all states in *proj*($S_{tentative}$, *other(A)*) are trivially pairwise concurrent with states in *proj*($S_{tentative}$, *influence(A)*), and with states in *proj*($S_{tentative}$, *core(A)*).

For $S_{tentative}$ to be inconsistent, either of the following two conditions must hold:

(a) $\exists\, x_{i,j} \in$ *proj($S_{tentative}$, influence(A))*, and $\exists\, y_{k,l} \in$ *proj($S_{tentative}$, core(A))* such that $x_{i,j}$ and $y_{k,l}$ are not concurrent.

(Note $x_{i,j}$ is the state initiated on process i by the event $e_{i,j}$ and $y_{k,l}$ is the state initiated on process k by the event $e_{k,l}$)

So either $x_{i,j} \rightarrow y_{k,l}$ or $y_{k,l} \rightarrow x_{i,j}$

For a user request of a breakpoint immediately before an abstract event A, local breakpoints on the processes in *influence(A)* are set immediately after *sunrise(early(A), influence(A))*.

According to our breakpoint scheme, $e_{i,j} =$ *sunrise (early(A),i)*.

Now, $x_{i,j} \rightarrow y_{k,l} \Longrightarrow e_{i,j+1} \rightarrow e_{k,l}, \Longrightarrow e_{i,j+1} \rightarrow$ *early(A)*

which contradicts our initial assumption that $e_{i,j}$ is the latest event on process i to precede *early(A)* (from the definition of *sunrise*).

Also, $y_{k,l} \rightarrow x_{i,j} \Longrightarrow e_{k,l+1} \rightarrow e_{i,j}$,

which contradicts our assumption that $e_{i,j}$ precedes *early(A)*.

Likewise for *late(A)* when the user requests a breakpoint immediately after the occurrence of A.

(b) $\exists\, x_{i,j}, y_{k,l} \in$ *proj($S_{tentative}$, influence(A))*, such that $x_{i,j}$ and $y_{k,l}$ are not concurrent.

Without loss of generality, assume $x_{i,j} \rightarrow y_{k,l}$.

By definition, this implies $e_{i,j+1} \rightarrow e_{k,l}$.

For a user request of setting a breakpoint before an abstract event A,

$e_{k,l}$ is the latest event on process k that precedes *early(A)*.

This implies $e_{i,j+1} \rightarrow$ *early(A)*,

which contradicts our initial assumption.

Likewise for *late(A)* when the user requests a breakpoint immediately after the occurrence of A.

So, $S_{tentative}$ must be consistent if the states in *proj*($S_{tentative}$,*core(A)*) are pairwise concurrent.

5.1.2 Algorithm

The actual algorithm that implements the scheme mentioned above is shown in Figure 5.8 and Figure 5.9. The algorithm has three phases: (1) initialization where algorithm-specific data structures are initialized (in practice, this takes place during the modeling itself); (2) consistent cut where the processes in the core set of the abstract event are placed in a pairwise consistent state according to the user request; and (3) backing up where processes in the influence set (which may have been placed in later states during the second phase) are backed up to the latest states where they could have influenced the processes in the core set.

1. *Initialization:*

Let relevant(A) := $\left\{ \begin{array}{l} \text{early(A) if request is to set a breakpoint before(A)} \\ \text{late(A) if the request is to set a breakpoint after(A)} \end{array} \right.$

$\forall$ event e, consistent_flag(e) := $\left\{ \begin{array}{l} \text{-1 if e} \rightarrow \text{relevant(A)} \\ \text{0 if relevant(A)} \rightarrow \text{e;} \end{array} \right.$

influence(A) := $\emptyset$
other(A) := $\emptyset$

2. *Consistency Check:*

```
for x ∈ core(A)
  e := first event on process x
  while ( the state terminated by e ≠ proj(S_tentative, x)
    if (get_consistent_flag(e) = 0)
      break;
    e := get the next event on process x;
  endwhile
endfor
```

3. *Back-up Phase:*

```
for x ∈ influence(A)
  e := first event on process x;
  while (consistent_flag(e) = 1) do
    if (e ↛ A)
      break;
    e := next event on process x;
  endwhile
endfor
```

Figure 5.8. Greedy breakpoint setting algorithm.

Proof of Correctness:

1. *Algorithm terminates.*

   Claim: get_consistent_flag() terminates.

   The input argument of the function can be any event. The outer loop where the function is initially invoked only examines the events in the processes belonging to $core(A)$ that precede the events in the sets *early(A)* or *late(A)*. So the domain of the argument that corresponds to the initial invocation of *get_consistent_flag()* is finite.

```
get_consistent_flag(Event e)
{
  if (consistent_flag(e) has been defined)
    return consistent_flag(e);

  if (e is a receive event) {
    w := the corresponding Send event of e;
    consistent_flag(e) := get_consistent_flag(w);
  }

  if (e is a send event) {
   if (pid(E) ∈ influence(A)) {
     if (E is the first event on pid(E))
       consistent_flag(e) := 1;
     else
        consistent_flag(e) := get_consistent_flag(preceding event of e);
    }

    if (pid(e) ∈ core(A)) {
      if the user request was to set a breakpoint before A, and E → early(A)
        consistent_flag(e) := 1;
      if the user request was to set a breakpoint after A, and E → late(A)
        consistent_flag(e) := 1;
      }
    }
  return consistent_flag(e);
}
```

Figure 5.9. Function get_consistent_flag()

At each recursion level, an event is considered that precedes the event under consideration in the previous recursion level. So the domain of the argument to the function decreases monotonically. Since it returns a value of 1 when the domain is empty, the function terminates.

Since the *for* loop examines each event in the processes belonging to $core(A)$ that precede the events in the sets *early(A)* or *late(A)* a constant number of times (once), the algorithm terminates.

2. *A local state in a process belonging to a core set is moved backward in time iff $S_{tentative}$ is inconsistent.*

   The breakpoint state of a process ($x \in core(A)$ is set at $s_t(e) \rightarrow$ *proj(*$S_{tentative}$*, x)* (i.e., the breakpoint on process x is set at an earlier state than that requested by the user through the abstract event), iff, e is the earliest event on x such that *relevant(A)* $\rightarrow$ e (that is the only time *consistent_flag(e)* = 0).

   Claim: Any consistent global state containing $s_s(e)$ cannot precede $S_{tentative}$

   Also for $\forall\ e'' \rightarrow e$ on x, $e'' \rightarrow$ relevant(A), (from the definition of e above). Hence any consistent state containing $s_t(e)$ must precede $S_{tentative}$.
   So, *proj(*S_B*, x)* = $s_t(e)$. The algorithm stops each process in the core set at the latest state that could precede $S_{tentative}$ and can still be a part of a consistent state.

3. $\forall x \in$ *influence(A), the algorithm stops the computation at proj(*S_B*, influence(A)).*

   In step 2, the following conditions hold.

   (a) No event on the processes belonging to the *influence* set have their *consistent_flag* set to 0.

   (b) $\forall e \in$ *influence(A),* such that, $e \rightarrow$ relevant(A), consistent_flag(e) = 1.

So after step 2, $\forall e \in$ *influence(A)* if consistent_flag(e) = 0, then $e \not\rightarrow$ relevant(A).

Since, in Step 3, we ensure that,

$\forall e \in$ influence(A), such that $e \not\rightarrow$ relevant(A), consistent_flag(e) = 0,

the processes in the *influence* set are stopped at the correct states.

In the preceding sections, we have introduced our breakpoint scheme based on abstract events and proved that our greedy algorithm indeed finds a unique closest consistent state preceding a user-specified inconsistent global state. Now we present sample debugging sessions that illustrate how such a breakpointing scheme can reasonably be used.

5.1.3 Example: Linear Optimization on Hypercube

Linear optimization problems involve solving *underconstrained* linear systems while maximizing/minimizing constraint expressions. In general the problem involves a m×n linear system, with $m < n$. A fundamental theorem of linear programming states that an optimal solution, if one exists, occurs when exactly m columns of the matrix have non-zero elements. Thus the algorithm degenerates into using a Gauss-Jordan technique on an under-constrained system where the column, on which *elimination* takes place, is chosen according to the coefficients of the constraint expressions.

In the parallel implementation, each processor is given one or more columns of the matrix and the constraint vector. The column to be eliminated is chosen by the minimum coefficient of the constraint vector. On each iteration, processors exchange the local minimum coefficients with increasing distant processors in a tree reduction/broadcast as shown in Figure 5.10; processors receiving a constraint coefficient less than their local minimum, make the received coefficient their minimum for subsequent exchanges. Thus *log(n)* exchanges are necessary for determining and broadcasting the global minimum. The target column and the position of the pivot row are also passed around with the constraint coefficient. If the local minimum is replaced by the received constraint coefficient, a local copy of target column and the pivot row position is also updated. The iterations are repeated until convergence is reached.

The parallel version of our program produced different results than the sequential version. In an 8 processor system, there will be log(8) = 3 exchanges in each iteration.

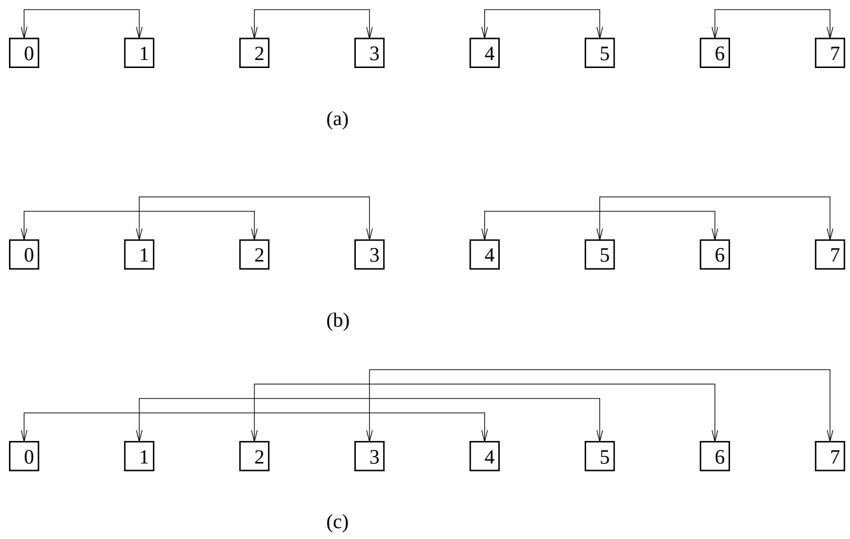


Figure 5.10. Leapfrogging communication pattern in linear optimization algorithm. Exchanges over a distance of 1 (a); exchanges over a distance of 2 (b); and exchanges over a distance of 4 (c).

The behavior of such exchange patterns can be described coarsely in Ariadne as

```
set procset = {0..7};
ch msg = W @ R;
pch exch = msg onall procset;
ptch phase = exch^3;
match lopt = < phase* >;
```

This model did not match the program's behavior as shown in Figure 5.11.

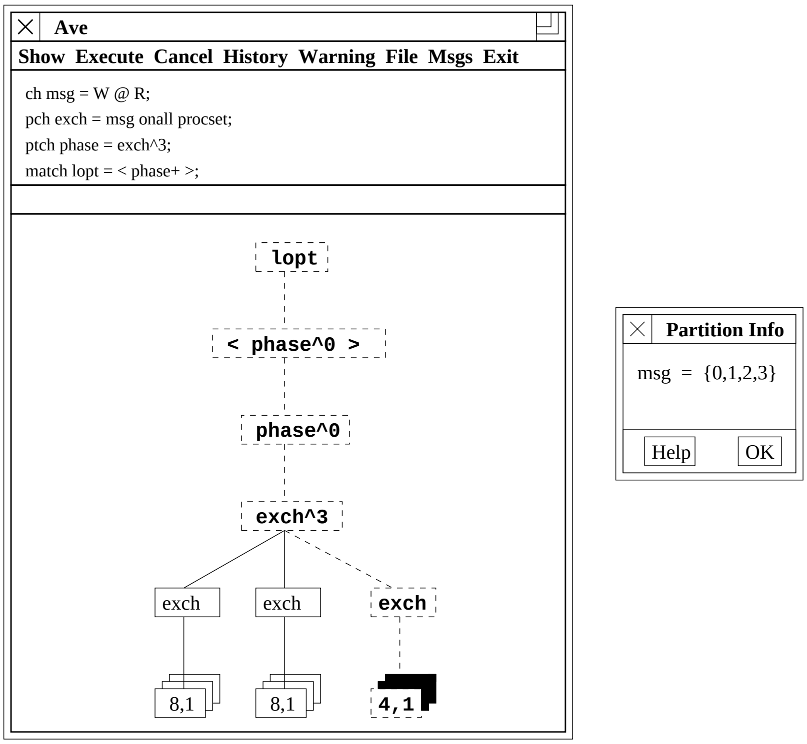


Figure 5.11. Ave feedback showing match failure in linear optimization

Each **exch** pattern was supposed to match on all 8 processors, but the third instance of the pattern **exch** matched on only 4 processors, as indicated by the integer pair (4,1) in the stack of boxes. The user can select the summary box, and get the feedback, shown in the window named *Partition Info*, that only processors 0,1,2 and 4 completed the third **exch**. Why did the model not match on the other processors? What are they doing in the third iterations? Why are processors 5,6 and 7 behaving differently than the rest? Event-based modeling can only examine observed behavior, and identify anomaly in program behavior. The underlying cause of an anomalous behavior can only be determined by examining local state information relevant to the anomaly. Our questions can only be answered when we examine the local states of processes 4,5,6, and 7 at the third iteration. This is the contribution of Ave/Ariadne: Its event-based modeling component can be used to identify anomalous behavior, and its breakpoint setting mechanism can be used to set globally consistent breakpoint immediately before the occurrence of an error so that state changes that lead to the error can be identified by selective execution of the program from the breakpoint. To see what has gone wrong we can set a breakpoint after the second `exch` event, by selecting it with the mouse button, and issuing the command

```
break after select;
```

The program was reexecuted under replay, and an enquiry with Ariadne about the global breakpoint state produced the source code window shown in Figure 5.12, where the source lines at which processes are waiting are highlighted. Additional output at the top of the screen identifies the processes waiting at specific source lines: Processors *0,1,2* and 4 are waiting after the `receive` call, while processors *3,5,6* and 7 are waiting before the same `receive` call in the source code. The source code window uses the concept of equivalence classes – processes that are waiting at the same statement are grouped together, and the source line is highlighted with a distinct color. In this case, processes were stopped at the same place in the source code. Within a given source line (all of which correspond to communication events), however, processes can be in two positions – before executing the communication library call, or after executing the communication library call: these are highlighted in the upper window.

The initial Ave feedback said nothing about the expected *precedes* relationship. Ariadne's matching occurs in two separate stages – first the required events are matched, if the required number of events occur in the trace file, only then the user-specified logical assertion is checked. In this case, since Ariadne did not find three `exch` events, it did not check the assertion. The request for setting breakpoint, however, prompted Ariadne to caution us (on the bottom of Figure 5.12) that it needed to readjust the local breakpoints on processors 4,5,6 and 7 in order to satisfy global consistency. This is likely due to a violation of a *precedes* relationship.

Why are the four processors stopped before the receive event, instead of actually doing the receive? Since the breakpointing scheme is replay-based, we can give additional output identifying the events that were most recently seen or next to be encountered. For this example the output will appear as:

Processes	When	Receive/Send	From/To
`0,1,2,3`	After	Receive	`2,3,4,5`
`4,5,6,7`	Before	Receive	`0,1,2,3`

In the second iteration, processors are supposed to communicate over a distance of 2. This means processes *4,5,6* and *7* are waiting to receive messages from wrong processors. Why are they behaving differently than other processors? Are they somehow in different iterations than the other processors? To answer these questions we need to examine the variable corresponding to the iteration count. The values of this variable will not be available within the framework of standard event-based debugging, unless all the variable values are initially logged

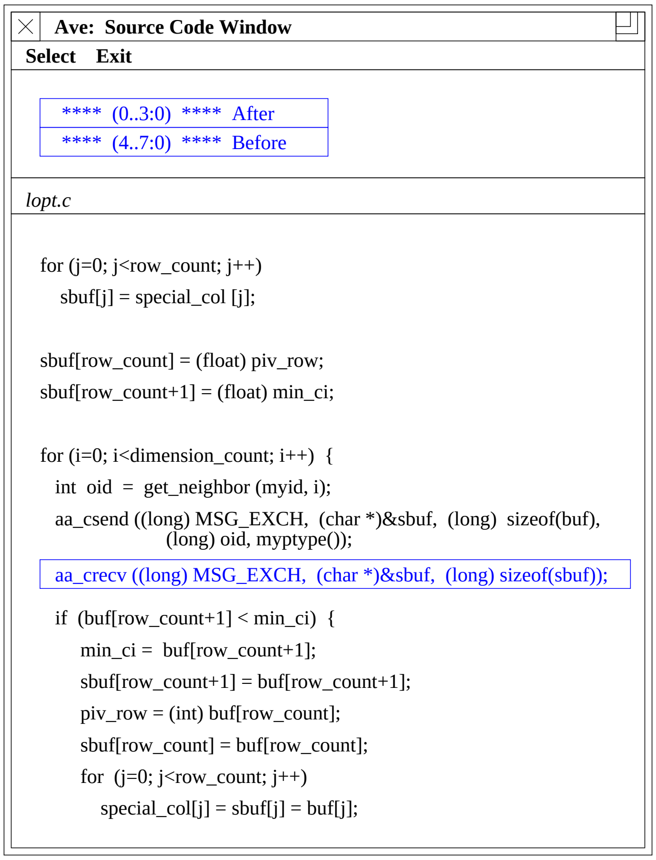


Figure 5.12. Source code window showing process position at global breakpoint (Linear Optimization).

which requires potentially enormous amount of storage space. At this point in our debugging session, we need to examine the local states of our program. We could have used any vendor supplied state-examination debugger. For this example which was written for Intel Paragon, we used the vendor-supplied *ipd* debugger.

5.1.4 The State-Based Component

ipd [44] provides comprehensive services for state examination and modification similar to those available on standard, sequential debuggers [2, 87]. We could, for example, examine the call stack at the breakpoint we've established as with the query shown in Figure 5.1.4.

```
(all:0) > frame

  ***** (0..3:0) *****
    aa_break_point() [avelib.c{}#11]
    aa_crecv() [avelib.c{}#99]
    main() [lopt.c{}#81]
    _crt0_start() [crt0.c{}0x000101e0]
  ***** (4..7:0) *****
    aa_break_point() [avelib.c{}#11]
    aa_crecv() [avelib.c{}#95]
    main() [lopt.c{}#81]
    _crt0_start() [crt0.c{}0x000101e0]
```

Figure 5.13. Call stack information from *ipd.*

For this particular example, having raised the question of whether or not all even processes are on the same iteration, we could look at the value of the variable *cnt* in the function *main().* The `display` command to *ipd* achieves our purpose:

We checked the iteration count i with the *ipd* command

```
(all:0) > display main()i

  ** lopt.c{}main()i **
    ***** (all:0) *****
    i = 1
```

indicating that all the processes are in the same iteration.

This points to a synchronization error. Processes 4,5,6,7 have received messages in the second iteration that they were supposed to see in the third. The user had used a library call, that accepts any message that arrives in the message queue.

The presence of race allows messages sent from processors `0,1,2,3` in their third iterations to overtake those sent from `6,7,4,5` in their second iteration. An explicit message receipt that specifies the identity of the sender process would remove the race condition. Messages would still arrive out of order, but they will be received in the correct order. Without the event-based modeling the user would have very little idea about where to start the investigation. How would one to know that a breakpoint needs to be set at the end of the second iteration, and not the first? One can always suggest that the user set breakpoint after every iteration, and check the identity of the sender process. But this scheme is unusable where programs have a large number of iterations, and the error happens to show up at the later phase. More importantly, had we been setting breakpoints without the benefit of replay, setting of breakpoints after each exchange (a send followed by a receive) would have produced the correct behavior! The breakpoint would have acted as a synchronization agent, eliminating the *race condition* that existed, and processors would have seen the messages in the correct order.

5.2 Example: Binary Image Compression (Continued)

In Section 3.2.3 we presented a partial debugging session with a parallel program for binary image compression. With our initial coarse grained modeling, we found out that the successive iterations are not synchronized properly since the expected *precedes* relationship did not hold between abstract events corresponding to individual iterations. We used Ave's query interface to ascertain that the failure of the temporal relationship is real and not an artifact of the coarseness of the initial model. We would now like to find out the missing synchronization in the source code. Event-based modeling can only highlight the error in terms of user-defined abstraction, it cannot identify the source code construct responsible for the error. To do that we need to set a breakpoint before the erroneous abstract event. The user can select the second **compr** event, and set a breakpoint

```
break after(select);
```

The program was re-executed under replay, and it stopped with the source code output shown in Figure 5.14,

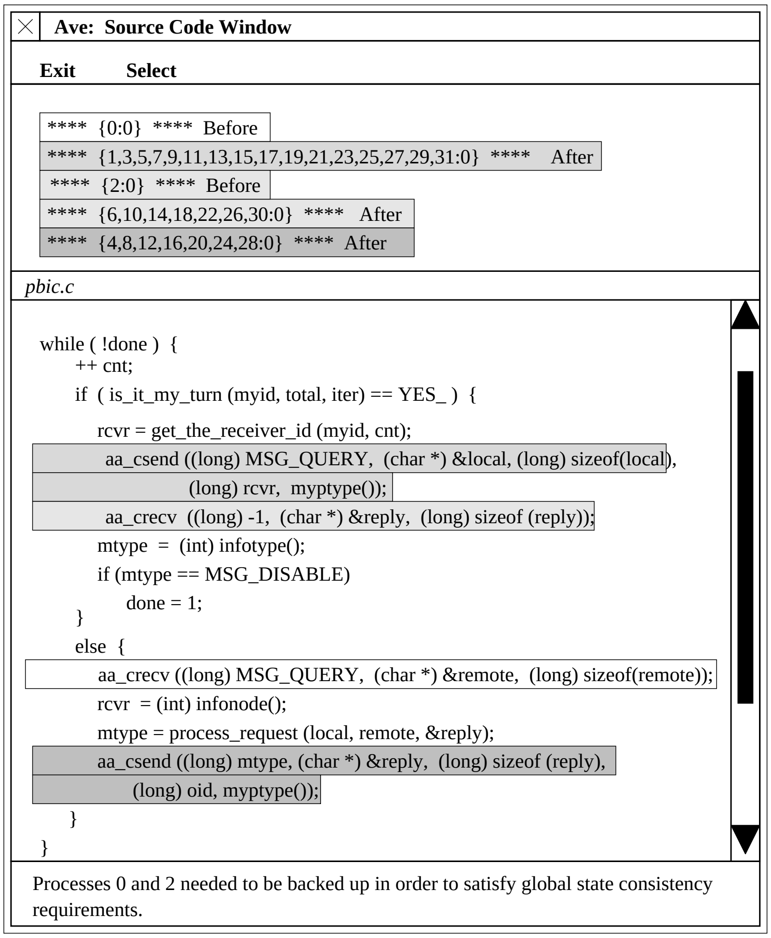


Figure 5.14. Source code window showing process positions at breakpoint (Binary Image Compression).

The message window output for our example appeared as below.

Processes	When	Receive/Send	From/To
`0`	Before	Receive	`4`
`1,3,5,7,9,11,13,15`	After	Send	`0,2,4,6,8,10,12,14`
`17,19,21,23,25,27,29,31`	After	Send	`16,18,20,22,24,26,28,30`
`2`	Before	Receive	`0`
`4,8,12,16,20,24,28`	After	Send	`6,10,14,18,22,26,30`
`6,10,14,18,22,26,30`	After	Receive	`4,8,12,16,20,24,28`

In the algorithm odd processes perform the exchanges in the first iteration, and do not take part in the communication phase any longer. The odd processes are stopped at the stage where they could still influence `compr[2]` which preserves the information that may be relevant to the error. In a correct execution, all even processes should be stopped *After* a communication event – the initiating ones *after* receiving the replies, and the *non-initiating* ones *after* sending those replies. The output, however, identifies process 0 and 2 as waiting *Before* receiving a message, but at different part of the source code. We used *ipd* to examine why processes 0 and 2 behaving are differently than the other even processes:

```
(all:0) > display main()cnt

  ** pbic.c{}main()cnt **
     ***** (1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31:0) *****
     cnt = 1
     ***** (0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30:0) *****
     cnt = 2
```

Since the variable `cnt` was used to hold iteration number, this output indicates that the odd processes are stopped in the first iteration, while the even processes are stopped in their second iteration. As we have seen in the message window output, process 2 is waiting to receive a message from process 0. So it must must have already sent a message to process 0. We looked at the message queue of process 0 to find an unprocessed message from process 2. Our replay-based technique has once again found a message race.

Without the event-based modeling the user would have very little idea about where to start the investigation. Sure, in retrospect, the user can set the same breakpoint by specifying four different local predicates on four different sets of processes. But how is he/she going to guess that a breakpoint needs to be set just after the second iteration, and not the first? More importantly how do you identify the second iteration? Note the odd numbered processes exit the `while` loop after the first iteration, and even numbered processes execute different branches of the `if` statement. There is no way for the user to stop the computation at this global state which we easily specified with the help of abstract events. The user would have to set three different breakpoints using conditional state expressions, and stop the odd numbered processes once they are outside the `while` loop, and the even processes at separate branches of the `if` statement. Further, without the benefit of replay, event his complex breakpoint would be useless against race conditions.

To continue with the binary image compression program, we modified it to remove the synchronization bug, but it still produced wrong results. The parallel version of the program at times produced compressed images that are smaller in size than those produced by the sequential version. In the parallel version of the algorithm, the processors receiving messages tagged as `succ` drops out of the computation, and their local images are merged with their sibling's image. Thus the size of the compressed image should be inversely proportional to the number of processors receiving `succ` messages.

Ariadne allows the users to log message contents during program tracing, and add new attributes dynamically[1]. In the reinstrumentation phase, we logged the contents of the `query` messages. Each `query` message consists of an integer, that indicates whether the sender has a homogeneous or a non-homogeneous image. The content is stored in the primitive events as an attribute named `value`.

The user used the same model as shown in Figure 3.6, which now matched the behavior. We argued earlier that a comprehensive debugging environment needs to support the alternation between state- and event-based debugging, and this is a perfect example of that methodology. The number of `disable` messages is a decreasing function (the curve has no inflexion) in a correct execution since a processor that could not compress its local image once can not do so later. To verify this we issued the following query:

```
with compression foreach compr compute cnt = count(compr_succ);
with compression show count(compr) wrt compr;
```

The line plot shown in Figure 5.15 identifies a problem with the algorithm. The third iteration shows no successful compression; yet the fourth and the fifth iterations have successful compressions.

We decided to investigate the behavior by displaying the message contents during the successful compressions. Once in the third iteration no two local images could be compressed, the message contents have to be 2 signifying that the local image cannot be compressed further. A `query` message with `value = 2` should never be a part of a successful compression, since the value of 2 already identifies

[1]The user still needs to recompile the program. However, the program execution can be controlled by the original trace, and thus the causal relationship between events remain unchanged even with the introduction of additional overhead.

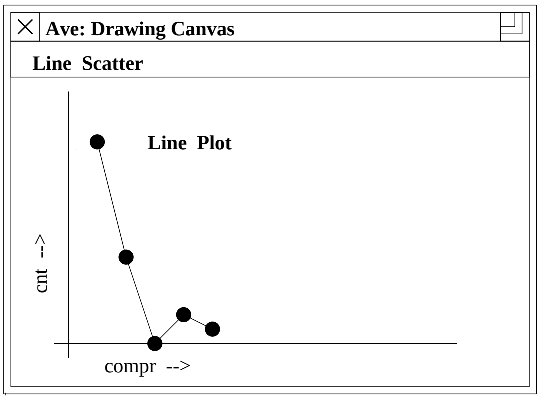


Figure 5.15. Ave feedback showing number of successful compressions is not a decreasing function.

the local image to be uncompressed. Either processors are sending wrong messages in the 4-th and 5-th iterations or messages are getting processed wrongly (starting at the 4-th iteration or even earlier). For the commands

```
with compression foreach compr_succ compute value = value(W_query);
with compression show value(compr_succ) wrt compr;
```

Ave displayed a Scatter Plot as shown in Figure 5.16, where the message contents in the fourth and the fifth iterations were indeed 2. So there must have been something wrong with how these messages were processed.

The user can now set a breakpoint after the processors receive the `query` messages in iteration 4, and examine the states of the ones that received a `query` message with `value = 2`. This can be accomplished by using a variant of the filtering scheme we introduced in Section 4.2. The user first identifies the messages tagged as `query` that belong to a successful compression and carry a value of 2 with:

```
with compression::compr_succ
  foreach R_query {eq(value(R_query),2)} add marker bp;
```

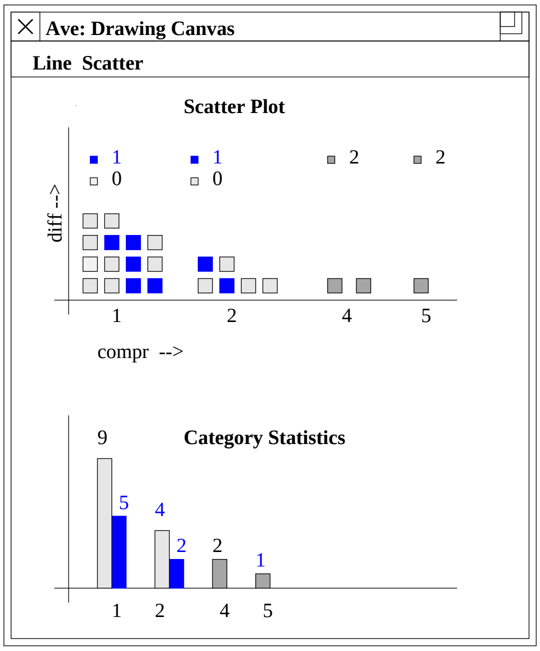


Figure 5.16. Ave feedback showing **`disable`** message sends even for non-homogeneous regions.

Each relevant event now carries a marker **`bp`**. We can set a breakpoint :

```
break after (select wrt bp);
```

`select wrt bp` selects those events that are marked **`bp`**.

The program when re-executed under replay stopped at the breakpoint with the source code window output as shown in Figure 5.17. As can be seen, only processes 0 and 16 are waiting before receiving a query; they are the target processes. We used *ipd* to verify that the messages indeed are carrying a value of 2 (this could also have been found out earlier by investigating the match tree).

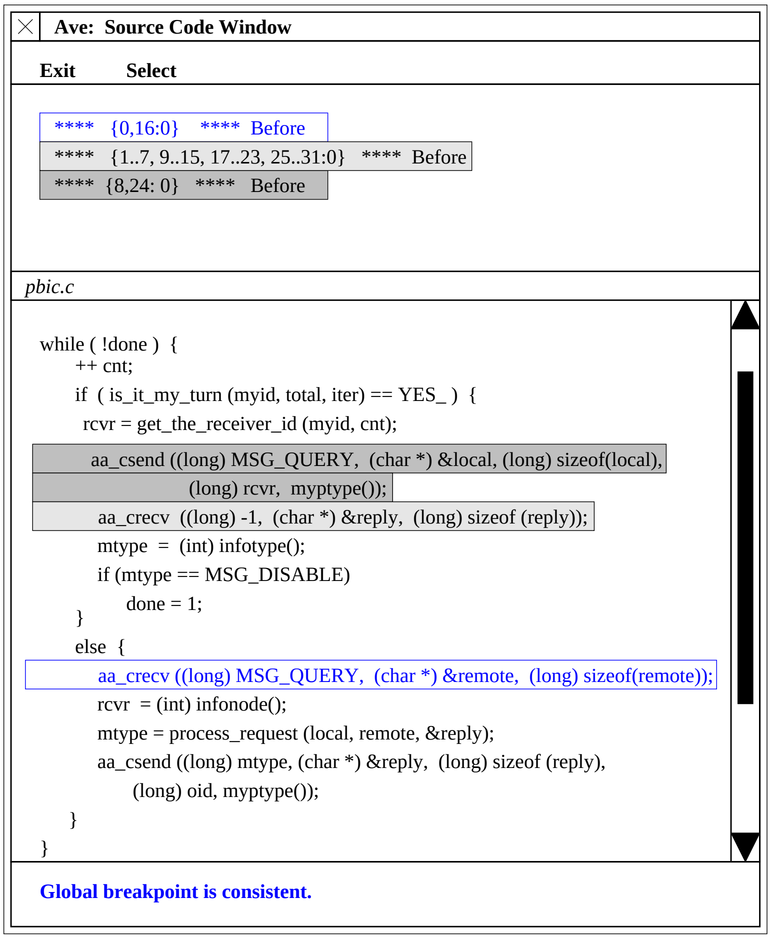


Figure 5.17. Ave source code window (Binary Image Compression, Second Breakpoint).

```
(0,16:0) > display main()remote.value

  ** pbic.c{}main()remote.value **
     ***** (0,16:0) *****
     remote.value = 2
```

At this point, we needed to execute processes 0 and 16 in isolation to see how they processed the message. First, we checked the image content of both the processes with *ipd*.

```
(0,16:0) > display main()local.value

  ** pbic.c{}main()local.value **
    ***** (0,16:0) *****
    local.value = 2
```

Since both processes have the same local image, and both have received identical messages, executing one of them in isolation is adequate. We executed process 0 under the control of *ipd* to find the function `process_query()` that decides what reply to send, did not have a special check for the case when the message content is 2.

5.3 Tracking Elusive States with Event-Based Breakpointing

In this section, we demonstrate the use of our integrated debugging environment on a parallel program for querying a knowledge base [34]. The knowledge is stored hierarchically as objects and classes of objects related by the *is-a* relation. To avoid redundancy, object attributes are stored as high as possible in the hierarchy and are then inherited at the lower levels. Thus, for example, "warm-blooded" is an attribute of "mammals" and "dog" *is-a* "mammal," so we can infer that any "dog" is "warm-blooded." To query such a database, the attributes of an object are found by following its *is-a* links outward in a spreading activation; as attribute values are encountered, they are returned to the originating node. The process is complicated by exceptions: "birds fly" but "penguins are birds" and "penguins don't fly." As processing proceeds, returned exceptions may invalidate earlier responses. Thus, the originating node must wait until all of the activity resulting from a query has terminated before replying. To detect termination, a *mass* attribute is attached to each message. The originating node sends out a total mass of 1, each activated node sends as much *mass* as it receives, and the originating node waits until it has received back a *mass* of 1. Multiple queries can be active in the knowledge base simultaneously. The nodes initiating the queries do not contain any part of the database.

The program uses five different message types. It was instrumented so that these message types appear as labels on interprocess read and writes within the trace. `plus` and `minus` messages spread activity through the network. `plus` messages request attribute values and `minus` messages indicate that existing values are overridden by an exception. `found`, `ignore`, and `mass` messages return answers to the originating node. `found` signals a valid attribute value, `ignore` notes an exception to a previously reported value, and `mass` indicates that no attribute value was found.

We tested our program and discovered that it did not operate correctly even on simple queries. It never found any attributes at all. To debug it, we first coarsely modeled its intended behavior.

For clarity, responses to the originating node were modeled by the writing of those messages as

```
ch response = W_found or W_ignore or W_mass;
```

and the receiving of those messages as

```
ch gather = R_found or R_ignore or R_mass;
```

The behavior of an activated node was modeled as

```
ch scatter = (( W <:  @ R response )* :> )*;
```

The behavior of the originating node was more complex because it must also collect responses. It was described using the above patterns as

```
scatter gather+
```

This behavior was bound to concurrent executions on a process set (here, **procset**), with

```
pch query = scatter gather+ onsome procset;
```

The entire series of queries over the lifetime of the program was defined as

```
ptch Query = query*;
```

To match this expected behavior against the execution trace, we used

```
match infer = Query;
```

which assigned the result of that match to the identifier **infer**.

For this example, the intended patterns of behavior were found in the execution trace and the match succeeded. Ariadne's feedback is shown in Figure 5.18 for a trace that contained a single originating node sending 5 different queries.

This first step in debugging did not show any obvious error in query handling. It is possible that the interprocess communication patterns were, in fact, correct, but it is also possible that our model was too coarse to detect the error. To rule out the latter, we examined the matched behavior more thoroughly using Ariadne's query facility as before.

With the following queries we verified certain assertions about the program execution.

1. The number of **found** messages received by the originating node was one more than the number of **ignore** messages received. (Thus, it appeared that a valid attribute value was found but somehow discarded.) The required

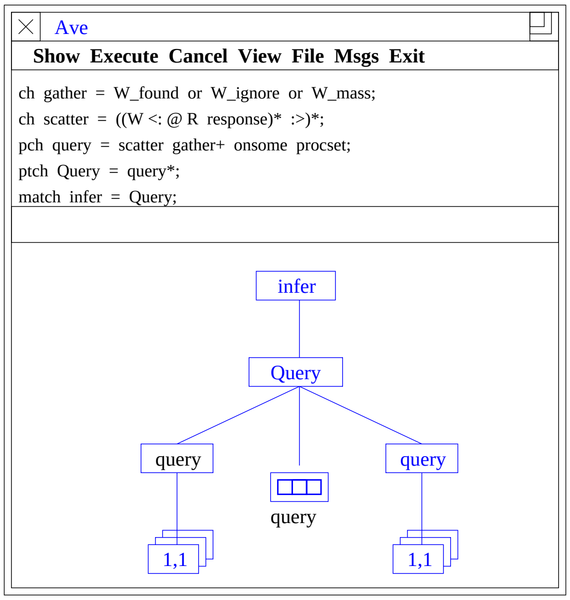


Figure 5.18. Ave feedback showing coarse grain pattern match in knowledge-base example.

commands were

```
with infer foreach query compute num_found
        = count (query wrt [R_found]);
with infer foreach query compute num_ignore
        = count (query wrt [R_ignore]);
with infer foreach query compute num_diff
        = num_found(query) - num_ignore(query);
```

2. A node that sent a **mass** message did not send any other message to the originator for this query. (Thus, the originating node was not receiving extraneous **mass** messages that might cause it to reply prematurely.) The required commands were

```
with infer foreach query compute mass_set =
        pid(leaf(query wrt [W_mass]));
with infer foreach query compute other_set =
```

```
        pid(leaf(query wrt [W_found W_ignore W_plus W_minus]));
    with infer foreach query compute diff =
        disjoint (mass_set(query), other_set(query));
```

3. `ignore` messages were sent only after a node received a `minus` message. (Thus, extra `ignore` messages were not overwriting good values.) The required commands were

```
    with infer foreach W_ignore compute flag =
        equal_string (tag(left), "ignore");
```

After this examination, we were confident that the interprocess communication patterns of the program were correct. Using event-based analysis we had not been able to detect any errors. We had, however, considerably narrowed our focus of attention: since the interprocess communication appeared correct, it was likely that the error was within the sequential calculation of mass or attribute values. Event-based debugging would not be useful in tracking this kind of an error, thus, at this point we switched to the *ipd* state-based debugger.

To use state-based debugging, we must be able to easily specify breakpoints that are consistent and meaningful to the programmer. What state would that be in our example? Mass and attribute values are processed both at the activated nodes and at the originating node. The obvious place to stop the program would be in the middle of a query, at a point where all of the activated nodes have just finished sending their replies and the originating node has not yet processed their replies. This state is difficult to specify in standard, state-based manner. Multiple queries can be present in the knowledge base and different nodes may handle the queries in different orders (if at all). We do not know in advance which nodes are activated by a particular query or even which queries will be of interest. Which processes should be stopped for a particular query? How will we know which query to stop them on? It is *impossible* to identify our target state in such a system using standard breakpointing scheme that uses conditional state expressions.

The specification of this breakpoint in Ariadne, however, is simple: the user scans the match tree, selects any matched **query** node, and requests that a breakpoint be set just after its **scatter** subpattern.

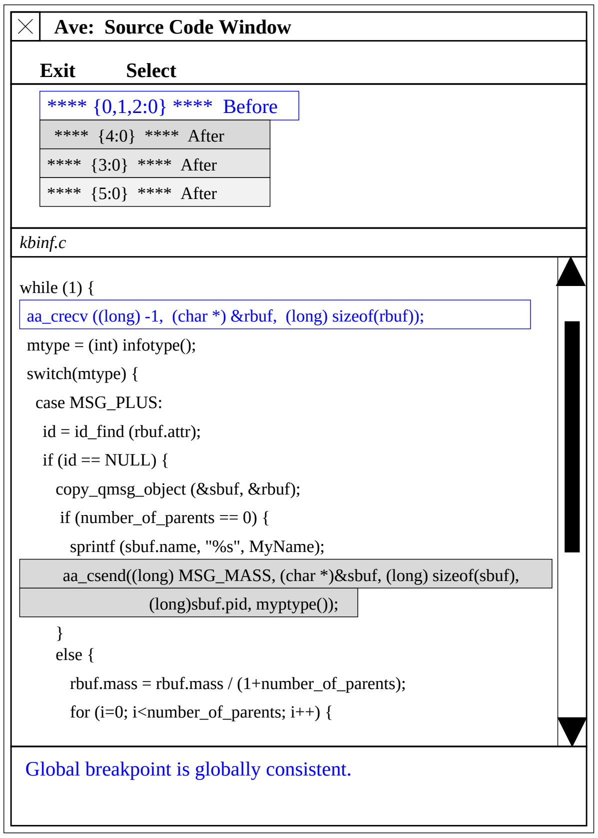


Figure 5.19. Ave source Code Window, showing processes stopped at different points.

Having done this, we replayed this execution of the program, stopping at the designated breakpoint as shown in the Ave Source Code Window in Figure 5.19.

Highlights in the figure indicate the source lines at which processes have stopped. From the output at the top of the screen, we can see that processes 0, 1, and 2 are stopped at the line indicated by the clear box, process 4 is waiting at the send operation indicated by the lightly shaded box, and processes 4 and 5 are waiting at source lines not currently visible in the window.

To continue debugging, we next choose to selectively execute the originating node (process 0) by setting a local breakpoint after each message receive with

```
(0:0) > break kbinf.c{}#412
(0:0) > continue
```

We examined the messages as they were received. The first **found** message, for example, was examined with

```
(0:0) > display main()rbuf

  ** kbinf.c{}main()rbuf **
    ***** (0:0) *****
    rbuf.mtype = found
    rbuf.sender = 3
    rbuf.element = Penguin
    rbuf.attribute = Locomotion
    rbuf.value = NULL
```

Notice that while the message was of type **found**, it did not contain a **value** (that is, **rbuf.value** was NULL). In fact, none of the **found** messages we saw contained any attribute values. Thus the problem was not in the originating node but at the responding, activated nodes (which were apparently not sending values in their messages). These nodes were already stopped immediately after they had sent those messages. A look at the preceding code section revealed that the programmer forgot to copy the result of the query to the send buffer.

Even if we allow overlapping queries initiated by different hosts to be active in the database, Ariadne will find the same consistent breakpoint as long as no host is allowed to submit a query before it receives the result of the previous query. The consistency requirement may, however, result in multiple queries being active during the replay. This will no doubt complicate examination of local states, since the user may no longer be able to isolate the effects of a single query. There is no way to avoid this: if the original execution exhibits interleaving behaviors, there is no way to untangle the local state changes corresponding to those behaviors without compromising the reproducibility of the execution.

5.4 Other Sample Debugging Sessions

5.4.1 Boolean Query in Parallel

This is a simulation of a distributed databases where boolean queries, whose answers can often be found without executing the whole query, can be asked. Some examples of boolean queries are "Is the value of the integral over the interval [a,b] greater than x?" or "Are there more than x employees with a particular advanced degree?" In the first case, the answer may be found by integrating the function over a subinterval, and in the second the query can be stopped as soon as the count of employees with the required advanced degree exceeds x. In case of a distributed database, a query is subdivided and given to all the worker processes, the worker processes may need to communicate among themselves for purposes of load balancing and such. Once the manager has enough responses to answer the queries, the workers are notified to stop their work on that particular query. Our test case of 20 queries showed wrong results being computed for the 9-th and the 15-th queries.

In our case, each *worker* has a database of functions, and the *manager* gets queries that ask whether the integral of a particular function over a given interval is greater than a specified value. The manager divides the interval in n equal sub-intervals and sends each worker a sub-interval, the function identifier, and the cutoff value; the messages sent are each tagged as `start` messages. This can be modeled at the chain level as:

```
ch start_query = (W_start <:  @ R_start :>)*;
```

Each worker uses adaptive quadrature method to solve the integral, by subdividing the interval into smaller subintervals. Since the rate of change of the function may not be uniform over all intervals, some workers would have a heavier load than the others. So each worker, when its local stack of subintervals reaches a threshold, sends the extra work to the manager for redistribution. The sending of the extra work by the worker (with messages tagged as `full`) and its redistribution by the manager (with messages tagged as `work`) can be modeled at the chain level as:

```
ch overflow = (W_full <:  R_full (W_work <:  R_work :>)* :>)*;
```

Once they are done solving the integral over the assigned interval, the *workers* send a `succ` message with the answer to the manager, which can be modeled as

```
ch success= W_succ @ R_succ;
```

When the manager can deduce the answer of the query from the accumulated partial result, it sends a MSG_DONE message to all workers telling them discontinue work on the current problem. This can be modeled as:

```
ch finish = M_done @ R_done;
```

The complete behavior can be written as

```
set manager = {0};
set workers = {1..7};
ch start_query = (W_start <:  @ R_start :>)*;
ch overflow = (W_full <:  R_full (W_work <:  R_work :>)* :>)*;
ch success= W_succ @ R_succ;
ch finish = M_done @ R_done;
pch Start = start_query onall manager;
pch Work = (overflow or success)* onall workers;
pch Finish = finish onall manager;
ptch Query = < Start Work Finish >;
match Quad = < Query+ >;
```

The Ave feedback in Figure 5.20 shows that the **precedes** relationship between the three abstract events within the 8-th query did not hold. As in the previous example, Ariadne stops the matching as soon it finds the first error. Since the abstract event Work did not precede the abstract event Finish as we had expected, we set a breakpoint after the Work abstract event in the 8-th query.

The program, when reexecuted under replay, produced the message window output shown in Figure 5.21, and the source code output shown in Figure 5.22. We noticed another strange thing – although there was a precedes failure, the setting of a breakpoint after the Work abstract event did not result in a backing up of any local breakpoints. The program was a true MIMD program, with manager executing a different program than the workers. Ave shows only the source file executed by the

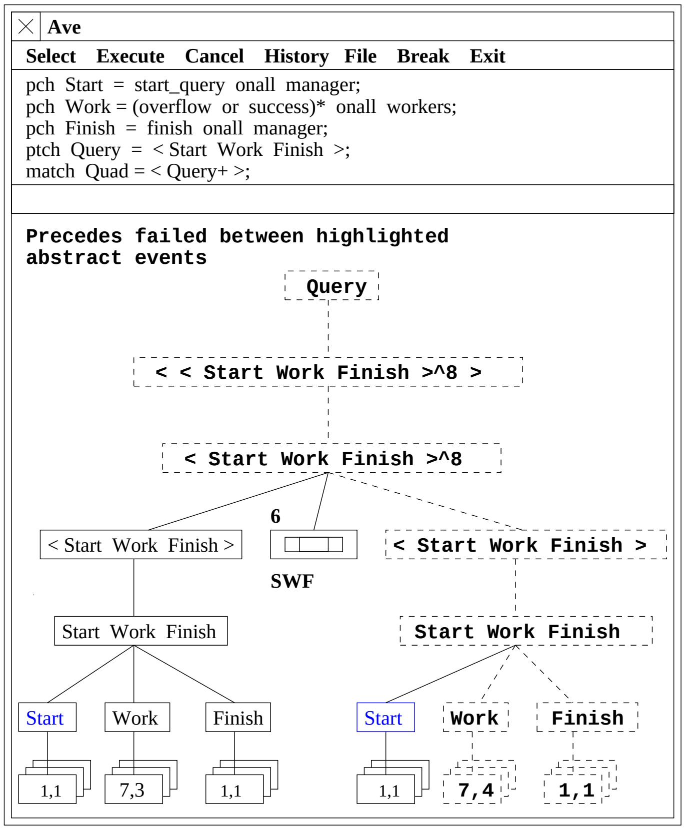


Figure 5.20. Ave feedback in the boolean query example.

manger (in this case process 0) (*manager.c*) initially as shown in Fig 5.22. But the user can easily select the `Select` button to see what the other relevant source files are, and select the *worker.c* file to check on the breakpoint state of the workers as shown in Figure 5.23.

The manager process shown in Figure 5.22 should have been stopped after it has received all the `succ` messages from the workers. Why is it waiting after the send of a `work` message? Figures 5.21 and 5.23 also indicate a problem. Why are the workers stopped at different points in the source code? Are they processing different queries? We used *ipd* to see examine the query each worker is processing.

```
(all:0) > display main()query_count

  ** aquad.c{}main()query_count **
    ***** (0:0) *****
    query_count = 9
    ***** (1..7:0) *****
    query_count = 8
```

The variable **query_count** was used to keep track of the queries being processed by different processes. The output thus showed all the workers (processes 1 through 7) processing the query 8 (**query_count = 8**), while the manager (process 0) has started processing query 9, even before it has received the **succ** messages corresponding to query 8 from processes *2,3* and *5*. Examination of the source code of the file *manager.c* revealed an error in the way the manager decided whether it has received all **succ** messages. The manager would send multiple **work** message to a worker, but would only wait for a single **succ** message.

Is it possible to identify the same global breakpoint using source line numbers on the local processes? The answer is "No". The end of the **Work** abstract event is marked by the *last* **succ** message sent by the worker for a given query. Since a worker can receive multiple intervals corresponding to the same query, the number of **succ** messages can not be pre-determined. Thus we cannot set a breakpoint using a conditional expression based on the message type and the number of times the loop

Processes	When	Receive/Send	From/To
0	After	Receive	5
1,4,6,7	After	Send	0
2,3,5	After	Send	0

Figure 5.21. Message window showing latest/impending events at the consistent global state.

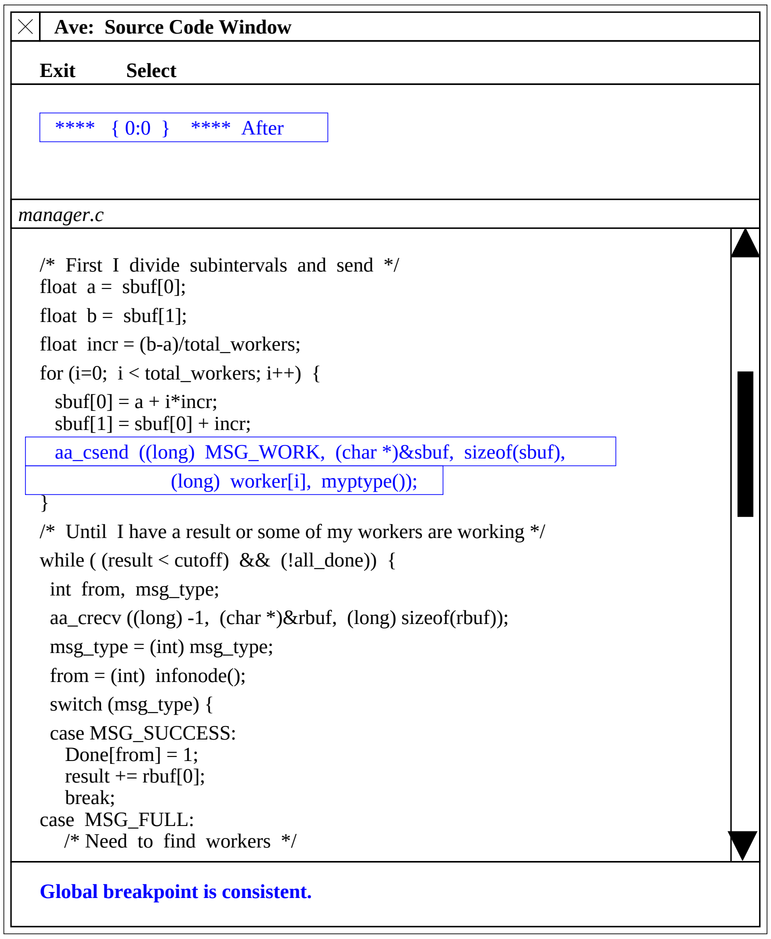


Figure 5.22. Ave source code window for the manager (Boolean Query Example).

has been executed. The same limitation prevents us from using conditonal state expressions to identify the state before the `Finish` abstract event. Note however, if the programmer sets a breakpoint after the 8-th iteration and examines the message queue, he/she will find one or more extra messages in the queue that belong to query number 8 (that will be received as a part of query number 9). Although helpful in this setting, we have already shown that such breakpoints often work as synchronization points and are not generally helpful in detecting errors due to race conditions.

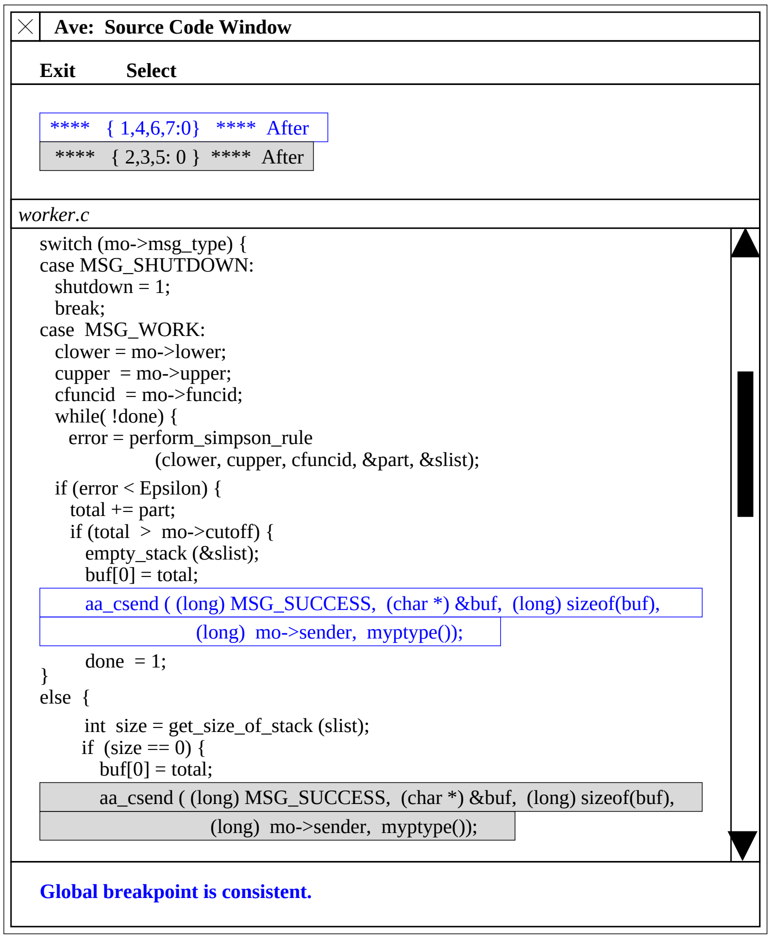


Figure 5.23. Ave source code window for the workers (Boolean Query Example).

5.4.2 Finding Biconnected Components in a Network of Processors (continued)

In the last chapter, we introduced this example to highlight Ariadne's filtering capabilities and its support for imposing multiple views on a behavior. The distributed algorithm to find biconnected components of a network starts with building a spanning tree of the network; identifies the articulation points in the spanning tree; and then performs a distributed election to identify the links belonging to the same biconnected component. By definition, biconnected components should be link-disjoint. In Section 4.2, however, Ave queries detected an anomaly in which abstract events representing different biconnected components were found to share a common edge (1,3). Note the biconnected components are only identified at the end of the distributed election. Articulation points initiate the election by sending messages to potential candidates to start an election, the links between an articulation point and the potential candidates are specially marked so as to prevent messages related to one election to be forwarded to processors participating in an election in a different biconnected component. The event-based modeling pointed to an anomaly in our prevention scheme that ensures biconnected components are link-disjoint. But how did the prevention scheme fail? Were some of the links to potential candidates not getting marked? Was there some error in the logic that determined potential candidates?

Event-based modeling cannot identify the source code construct responsible for the failure without capturing each local state modification in the event trace. This is prohibitively expensive. More importantly, this will make the modeling of behavior so complex that the initial rationale for using event-based modeling – that it allows for examining program behavior in the abstract – would be lost. The most effective way to track the error in source code is through state examination based on the anomaly in the abstract behavior. Since both computations laid claim on the link between nodes 1 and 3 a good choice of meaningful breakpoint might be to stop before either of the computations has a chance to use the link. Since we do not know which of the computation was the first to use the link we set two breakpoints – one in the context of the color computation 7, and the other in the context of the color computation 6. The re-execution of the program under replay will automatically stop the computation before the first breakpoint.

Selecting each `color_phase` abstract events in turn, we issued the following commands

```
set link_set = {(1,3), (3,1)};
with select foreach event
    {eq(disjoint(link_tuple(event),link_set), 0);}
        add marker bp;
 set break before (select wrt bp);
```

where `event` is a keyword that matches any generic event. The first command is a set definition. The second adds a marker `bp` to all events that corresponds to the interactions between processes 1 and 3. `select wrt bp` identifies those events with the marker `bp`.

Ariadne did not complain about setting the breakpoints, so we were sure that they did not conflict. Replaying the program stopped the computation with the Ave source code window output as shown in Figure 5.24 and the message window as depicted in Figure 5.25.

As we expected the computation is stopped immediately before processes 1 and 3 could interact with each other in a successful coloring computation (ignoring any `init` messages). We examined the local state of the process 3. Each process in the computation uses two separate buffers for sending and receiving. Since process 3 is waiting before a send, we can look at the content of its buffers to see the message that has caused it to send the `clr` message to process 1.

```
(3:0) > display process_message()rbuf

  ** bcc.c{} **
    ***** (3:0) *****
    rbuf.pid = 5
    rbuf.oid = 3
    rbuf.tag = "clr"
    rbuf.contents = 7
```

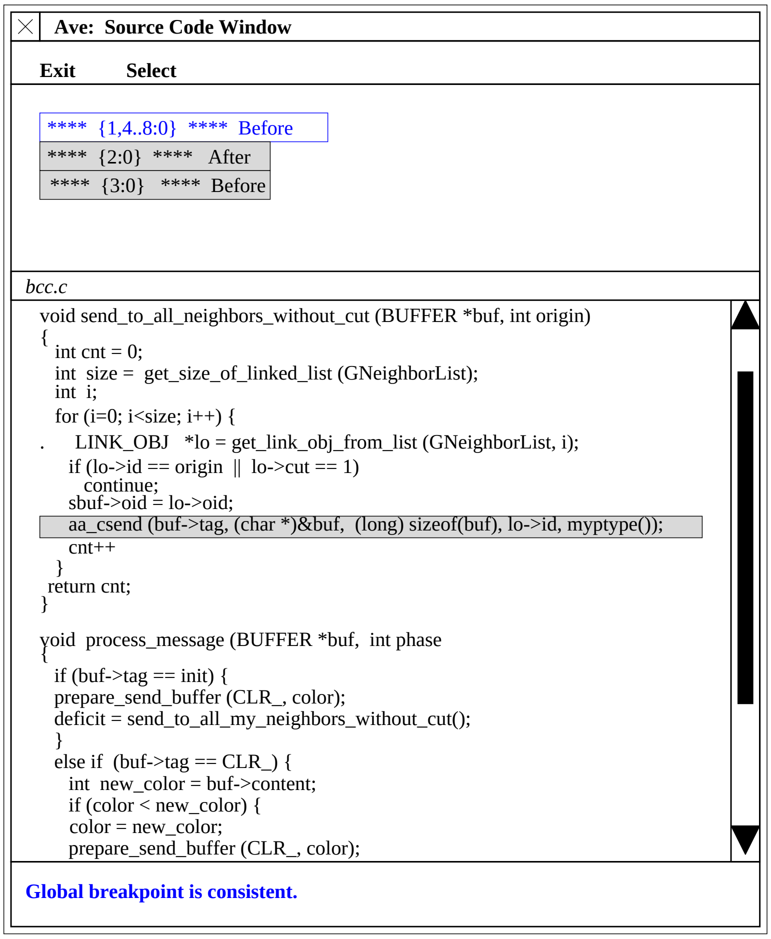


Figure 5.24. Ave source code window in the biconnected component example.

Process 1 was replying to a `clr` message sent by process 3. We already know that process 3 is an articulation point. As a part of the computation in phase 2, the links incident upon each articulation point are divided into two sets – cut set and non-cut set. Messages received from a member-link of the non-cut set should only be forwarded through other member-links of the non-cut set; messages received through a link in the *cut-set* should not be forwarded at all. So the next step was to investigate the characteristics of the link (5,3). Each process keeps a linked list of links incident upon it. The search was a bit tedious, since *ipd* does not provide

Processes	When	Receive/Send	From/To
1,7	Before	Receive	3,5
2	After	Send	3
3	Before	Send	1
4,5,6	Before	Receive	3,3,5
8	Before	Receive	1

Figure 5.25. Message window output for biconnected component algorithm.

any way to print the contents of a whole linked list. But the *ipd* output did point to the error.

```
(3:0) > display process_message()
      *((LINK_OBJ *)GNborList->next->next->contents)

  ** bcc.c{} **
    ***** (3:0) *****
    node = 5
    pre_order = 7
    cut = 1
    link_type = "child"
```

The link (5,3) was indeed in the cut set of process 3, and any message from process 5 should not have been forwarded to another. A check in the code for **send_forward()** revealed that the programmer had omitted the check altogether. As a result all messages received through the links in the **cut-set** were only forwarded to the members in the *non-cut* set. The messages received through the non-cut links were handled properly.

Running the updated code produced 4 biconnected components instead of 3. The link between process 1 and process 3 received a color of 2 and formed a separate biconnected component. This link was earlier included in 7 due to the erroneous conditional expression. This error could have been caught in the initial modeling phase if we had included the information regarding the local process state in the trace file after each phase of the computation. Nevertheless the example proved the versatility of the breakpoint setting mechanism due to its close relationship with the event-based modeling.

5.5 Conclusions

Parallel programs require a multi-level debugging strategy that can adapt to the needs of the programmer as the debugging session progresses. Ave/Ariadne provides an environment where the programmer is capable of changing back and forth between event- and state-based analysis techniques. The usual debugging process in Ave/Ariadne progresses methodically with the result from each step guiding the next. Initial examination starts with post-mortem event-based modeling of the program behavior at an abstract level. Analysis done in the post-mortem modeling phase is used to guide the replay-based interactive phase to stop the computation in a meaningful consistent global state, and examine the local states in an arbitrary level of detail. Ave/Ariadne's breakpointing scheme features:

- Specification of a breakpoint in terms of an anomalous abstract event, instead of ad-hoc specification in terms of predicates on local states. This scheme ensures that user would only examine the local states most likely to have caused the error.

- Specification of breakpoint states that are almost *impossible* to identify, since the number of events that must occur before the state is reached cannot be determined at compile-time.

- Stopping the computation at a globally consistent state, even if the user request corresponds to an inconsistent global state.

We have found that this integrated approach of setting breakpoint using abstract event can provide services that either event- or state-based approaches cannot provide individually or even working together with the traditional breakpoint specification techniques.

Chapter 6

Conclusions

In this chapter we summarize the work, highlight our conclusions and identify areas for future work.

6.1 Summary and Contributions

Asynchronous parallel programs are difficult to debug. Traditional state-based techniques without powerful abstraction mechanisms cannot accommodate the sparsity of consistent global states among the large state spaces, nor can they handle the effects of asynchrony. Event-based techniques offer an alternative for initial stages of debugging where user-specified models of intended program behavior are compared to actual program behavior as captured in execution traces, and errors are detected in terms of the user-defined abstraction. Event-based behavioral abstraction can contend with the large state space by allowing hierarchical modeling; it avoids the issues of global consistency; and it can perform the modeling in logical time to filter out perturbations due to physical asynchrony. Current event-based debugging tools, however, have three significant limitations: they do not provide sufficient feedback on match failures; they do not scale well for massive parallelism; and they are not very adept in tracking errors back to the source code.

In this dissertation, we address each of these three problems. Specifically, this dissertation contributes:

A Simple Modeling Approach. Ariadne uses a simple modeling language for specification of coarse grained models of program behaviors. The language supports the principle of successive refinement, wherein the user can start with a simple model and refine it as he/she gets more feedback from the debugger. Ariadne supports simple yet powerful filtering mechanism, comprehensive logical time manipulation facility, and a powerful functional query interface that aids the user specifying and exploring complex program behavior. A structured hierarchy of three definitional levels, the handling of logical time and a simple filtering scheme allows Ariadne to express complex program behaviors with scalable models. More importantly, the parsing interpretation of model matching, allows Ariadne to detect the first occurrence of a mismatch between the model and the program behavior, and enables it to provide detailed and precise feedback regarding the mismatch.

A Scalable Visualization Scheme. The Ave visual interface provides scalable visualization of the match tree, automatically partitioning the processes into equivalence classes based on their abstract behaviors. It treats the match tree visualization as a spreadsheet interface allowing the users to compute derived attributes of abstract events, which can then be partitioned into equivalence classes based on derived attributes. Ave's scalable visualization is based on this user-directed partitioning of abstract events (i.e., abstract behaviors) into equivalence classes of derived attributes. It incorporates scalable visualization techniques used in visualizing performance data. Ave in conjunction with Ariadne provides a query language that allows the user to control the proportion between the scalability of the visual feedback and its precision.

An Abstract Event-Based breakpointing Scheme. We provide an implementation of an integrated debugging environment, where the results of the event-based modeling is used in conjunction with the scalable visualization of the behavior to specify a globally consistent breakpoint. This approach allows the user to re-execute the program under the control of the trace file, and examine the program state to an arbitrary level of detail. The event-based component of the environment makes the powerful abstraction mechanism available to the user to help narrow the focus of investigation to the offending part of the behavior; and the integration of the vendor-supplied debugger enables us to to investigate the global breakpoint state, specified in terms of an erroneous abstract event, to an arbitrary level of detail. This behavioral approach of state selection bridges the gap between the two approaches: it enables the use of a powerful abstraction mechanism so lacking in purely state-based debuggers, and it allows us to track an error back to the offending source code construct that has so far been unavailable with purely event-based debuggers.

To this end, we designed an integrated debugging environment, where the results of the initial modeling phase is used to guide a subsequent state-based interactive phase, where consistent global breakpoints are set in terms of abstract events, and global states are explored with vendor supplied state-based debuggers. The integration of two seemingly disparate methodologies allows our debugger to contend with a large state space and the effects of physical asynchrony without losing our ability to examine local states in arbitrary levels of detail.

Currently available debuggers are only good for particular stages of debugging. Although it was long recognized that debugging strategy needs to evolve as one detects the fault and continue to isolate it to a particular code segment, no implementation of such integrated environment was proposed.

6.2 Limitations of Ave/Ariadne

Ave/Ariadne is an effort to integrate the two disparate methodologies of debugging. It depends largely on the ability of replaying the program to gather additional state information. We thus inherit the limitations of any replay-based scheme. Firstly, for a long running program the collected trace would be large. To ameleorate the effects of tracing overhead would require Ave/Ariadne to incorporate consistent checkpointing and optimal tracing as outlined by Netzer and Xu [69]. Secondly, Ave/Ariadne guarantees reproducible execution only for passive message passing programs. Implementing a a replay facility in an active message passing environment would require kernel support. Most importantly, our replay scheme assumes that the program is unoptimized. In presence of aggressive optimizations, the replay scheme may not work. With code hoisting, strength reduction, common subexpression elimination and other such optimizations, however, even a simple sequential program execution differs from the expectation presented by the source code abstraction. The issue debugging optimized code thus needs to be addressed separately.

6.3 Future Research

Ave/Ariadne is a research tool designed to introduce a methodology for comprehensive debugging of parallel programs, and to prove the feasibility of implementing such a debugging environment. The implementation and subsequent use of Ave/Ariadne have proved the effectiveness of such a comprehensive debugging approach. In doing so, Ave/Ariadne has opened up new research directions. Extensions to Ave/Ariadne should include:

1. *Incorporating data-structure interactions in the modeling language.*

   Ariadne can model data structure interactions explicitly in the modeling language. But to be effective, we need a way to automatically capture the data structure interactions in our traces. Once those interactions are captured in the trace, the data structure dependencies can be specified by the user incrementally, as a part of the query language, so as to not to make the models cumbersome.

2. *Exploring ways to handle object-oriented languages.*

   Object-oriented languages are good candidates for event-based debugging, because of the complexity of control-flow, and the ease with which event-generation can be encapsulated within the individual objects as part of the method invocations. However, message passing in object oriented languages is implicit, and thus whether the modeling approach used for explicit message passing programs can be implanted as-is in the object-oriented domain remains to be seen.

3. *Integrating performance debugging under the same framework.*

   Ave/Ariadne deals with multiple dimensions of time. Performance modeling and debugging can thus be provided as a special case of correctness debugging, since it only deals with a one-dimensional time.

4. *Using the spreadsheet based query language on data structures.*

   Often, a program has to deal with large data structure, which must meet certain integrity constraints for the program to be correct. Currently, there does not exist any easy way to verify such constraints. Ariadne's query language can be used in conjunction with Ave-like scalable visualizations for verifications of such integrity constraints. This would aid in the debugging of large sequential programs as well.

5. *Incorporating facilities to handle optimized code debugging.*

   As we mentioned earlier, aggressive optimization can even alter a sequential program's execution profile. Most sequential debuggers require that the program be compiled with optimization turned off. The problem of debugging optimized code has many similarities with parallel program debugging. Both require abstraction facility to hide the low level detail of the actual execution and yet maintain enough information to present the user with a meaningful view of the program behavior. The modeling and scalable visualization techniques developed in the context of parallel program debugging in Ave/Ariadne can be modified to facilitate debugging optimized code.

Appendix

Ave/Ariadne User Manual

Ariadne: Specification of Behavioral Models

Ariadne is a tool to explore and model Lamport's *Happened Before* graph of a parallel program execution. The nodes of this graph are events that occur at different threads of execution. The edges of the graph are of two kinds: ones that signify *program order*, and the other that signify *causal order* due to inter-process interactions. Ariadne specifies the program behavior in terms of sets, chains, p-chains and pt-chains.

Set Description

Ariadne supports description of sets in two different ways: using explicit enumeration, and through set operations of union, intersection and cartesian product. The explicit enumeration syntax is:

```
set <set-name> = {l1..u1 [by x1], l2..u2 [by x2], ..., ln..un [by xn]}
                   [with imap <imap-name>];
```

We will discuss the concept of *imap* shortly. **set** is the keyword signalling set definition. **set-name** is the user selected name of the set, l_1, u_1, x_1,l_2,u_2, x_2,...,l_n,u_n,x_n are required to be integers given as explicit constants. The terminology is:

l_i is called the lower limit of dimension i.
u_i is called the upper limit of dimension i.
x_i specifies the increment to be used while enumerating the range specified by l_i and u_i, the default increment being 1.
$(u_i$-$l_i)/x_i$ will give the size of dimension i.

Thus one can specify a set consisting of elements 1 through 101 with the construct

```
set onetoomany = { 1..101 };
```

where **set** is the keyword signalling a set definition. Since sets normally consists of process-identifiers, the user may choose to have a set that includes every third process. This can be done with

```
set everythird = { 1..101 by 3};
```

Often programmers form a logical n-dimensional grid on a set physical processors. When instrumentation is done at run-time, the process identifiers are single integers, from which different dimensional indices are computed by the user. Ariadne allows the user to use the logical view of the n-dimensional co-ordinate system. The user needs to specify the relation that will map such *logical* n-dimensional co-ordinates to the process-identifier specified in the trace. This mapping can be specified by through index-mappings. The syntax of index-mappings is:

```
imap <imap-name> = <expr> ;
```

where **imap** is a keyword signalling *index mapping* definition, **imap-name** is the name given to this expression by the user, and **expr** is any arithmetic expression that describes the mapping.

For instance,

```
imap X = index1*10 + index2;
imap Y = index1*10 + index2*5 + index3 + index4;
```

provides one way of mapping elements of two-dimensional sets to the process identifiers. **index1**, **index2** etc are all reserved words, only to be used in the context of index mappings.

Thus the user can specify a two dimensional set as

```
set 2Dset = { 1..10, 1..10 } with imap X;
```

The second way of specifying sets is through combining one or more defined sets. The syntax is:

```
set <set-name> = <set-expr> [with <imap-name>];
```

where **set-name** is the user-defined name of the new set. **set-expr** is any arithmetic expression where the identifiers are pre-defined sets and the operators are restricted to *union*, *section* (meaning intersection) and *cross* (meaning cartesian product).

For instance, we can define two sets A and B as:

```
set A = {0..10, 0..10} with imap X;
set B = {0..10, 5..15} with imap X;
```

and then use them to define new sets

```
set Union2d = A union B with imap X;
set Inter2d = A section B with imap X;
```

```
set Cross4d = A cross B with imap Y;
```

The default index mapping is the identity mapping, which is only meaningful for one-dimensional sets. If index mapping is not specified during set definition, the current index mapping is used during run-time to resolve the set elements. Note no error checking is performed: a 3-dimensional set may have an index mapping that only involves two indices and so on.

Chain Descriptions

Chains are local view of computation, and are primarily used for specifying a sequence of events. The execution graph of a parallel program consists of nodes that signify events and edges that signify two different relationships that could exist between events: one that events are related by the same program order (that they occured on the same process), and the other that events are related by *causal order* (that they are related by the sending and receiving of messages or some other form of comuunication). In our regular expression, events are specified using identifiers, while the program order edge is specified by a space, and the *immediate* casual order is symbolized by the `@` symbol. Note program order and causal order are distinguishable only in the chain level where primitive events are involved. At higher levels, we have abstract events and there we do not explicitly distinguish between the program order or causal order edges.

The chain definition has the following syntax:

```
ch <chain-name> = <chain-expr> [wrt wrt-list];
```

where `chain-name` is the name of the user-defined chain. `chain-expr` is a regular expression enhanced with the operators `@`, <: and :>, the significance of which will be discussed shortly. An identifier in the `chain-expr` can be a primitive event or a user-defined chain. Recursive definitions are not allowed. `wrt-list` is a comma separated list of primitive events. We will discuss the optional `wrt`-clause later in this section. `ch` and `wrt` are keywords in Ariadne.

Formally, a chain is a *tree* that is to be matched against the execution history graph. Nodes in the tree can represent communication events: *ReadEvents* denoted by `R`, or *WriteEvents*, denoted by either `W` or `M` (depending on whether it is a single write or a multicast), Events can be suffixed with explicit tags such as `W_pivot` and `R_elim`.

For instance, the user can define a chain

```
ch msg = W R
```

to specify the the pattern of an occurence of `W` and an occurence of `R` in the same process. The concatenation specifies the program order, and is normally denoted by space. The associated section of the execution graph is shown in Figure A.1.

The matching process for a chain always begins on a single process. It may stay entirely on that process, as in the above examples, or it may migrate to other

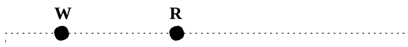


Figure A.1. Depicitng program order in chain description.

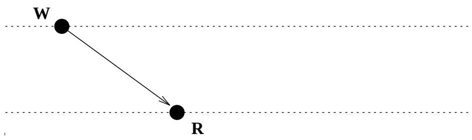


Figure A.2. Depiciting causal order in chain description.

processes by following a communication arc. An `@` occurring between a **`WriteEvent`** and a **`ReadEvent`** event denotes a migration; matching is suspended on the writing process and initiated on the receiving process. Thus

```
ch msg = W @ R
```

defines a pattern in which the `W` is matched on the initiating process and the `R` is matched on the receiving process. The corresponding section of the execution graph is shown in Figure A.2.

Similarly,

```
ch msg_and_write = W @ R W
```

describes a pattern where in the first `W` is matched on the initiating process and the remaining `R` and `W` are matched on the receiving process. The execution graph is again shown in Figure A.3.

If an `M` precedes an `@`, matching is simultaneously initiated on all receiving processes.

The following chain definition follows a message routed through different processes.

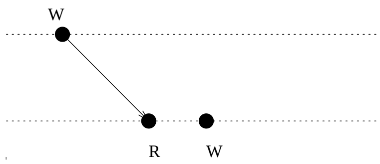


Figure A.3. Movement of matching context between different processes.

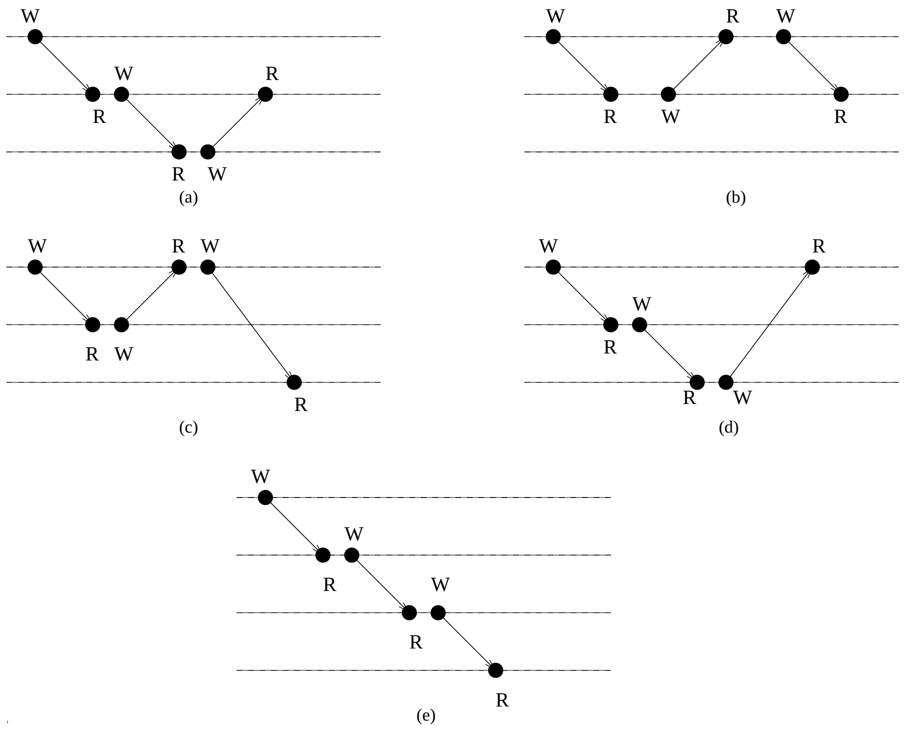


Figure A.4. Coarseness at the chain level. The same chain description can match any one of the five behaviors shown in (a) through (e).

```
ch msg = W @ R W @ R W @ R
```

This definition can match 5 different execution graphs as shown in Figure A.4.

Note the `@` symbols moves the context of the pattern from one process to another. Once matching has been completed on the receiving process(es), it may resume on the writing process where it had been suspended. If this is the case, it is indicated by "marking" the point of return before migration with the symbol `<:`. The symbol `:>` indicates that the matching should return to the most recent process on which it had been suspended, at the point of suspension. Thus

```
ch two_bcast = M <: @ R :> M @ R W
```

models two consecutive multicasts on a single process; in the first case, the matching control returns to the initiating process, in the second it does not. The corresponding time diagram for a 4 process system is shown in Figure A.5.

The other directives to the debugger come in the form of filtering and assertions. Often it is the case, that we only would like to consider a subset of events in the trace file for the purpose of behavioral modeling. One way to do it would be filtering the events using event attributes as is done in the context of databases using *select* statements. We also provide a *dynamic select* facility at the chain level using the `wrt` clause.

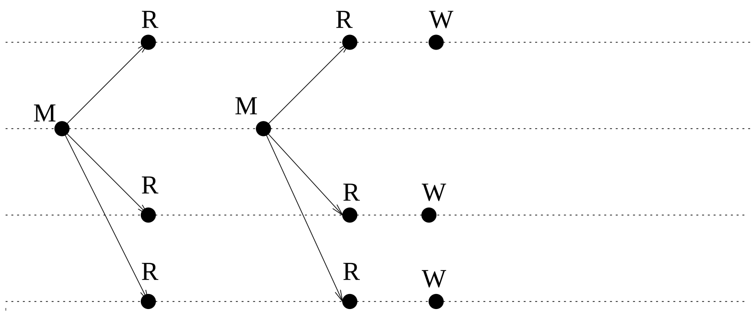


Figure A.5. Saving and restoring of matching context.

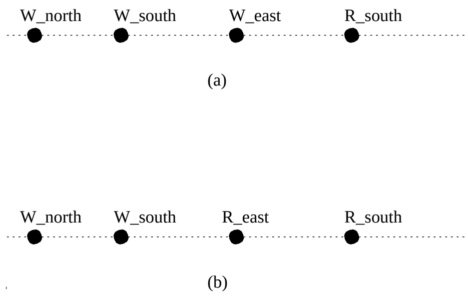


Figure A.6. Filtering at the chain level: execution graph that matches the chain description (a); an execution graph that does not match the chain description (b)

```
ch msg = W_north R_south wrt [W_north, R_south];
```

tells that if the debugger only considers the W_north and R_south events in the trace, then it could see a pattern where a W_north event would be followed along a program order edge by a R_south event. Note that such filtering relaxes the constraint of *immediate* program order.

This kind of filtering can also be used to check atomicity constraints. Say we expect to see in the program order a `W_north` event followed by (not necessarily immediately) a `R_south event`, but without an intervening R_east event. This can be expressed as,

```
ch msg = W_north R_south wrt [ W_north, R_south, R_east];
```

Since the filter allows all three event types to be considered for matching, a presence of intervening `R_east` would led the match to a failure, but the absence will result in a successful match. Figure A.6 shows a section of execution graph that will match the chain description and one that will not.

One can also control the matching based on attributes. One restriction that is imposed to keep the recognition algorithm simple, is that assertions of attributes

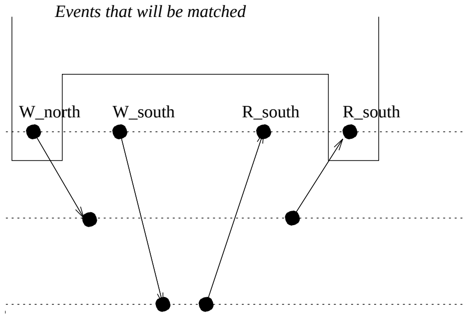


Figure A.7. Fine grain behavior specification.

should only involve events already matched. This ensures that we never discard a event based on some assertion on yet unseen events. The description

```
ch msg = W_south {x = oid(W);} R_south {aof eq(pid(R),x);};
```

matches a W_south event followed by a R_south event in the same process such that the `R` event is sent by the process that received the `W` event. `pid` and `oid` are inherent attributes of each communication events, and refer to the process-identifier of the sender and that of the receiver respectively. Figure A.7 highlights the effect of the filtering on the matching.

As we mentioned before, chain description can be quite coarse. Consider a case, when a process broadcasts a message to all other processes, and depending on the response chooses a recipient for its next message. The user description

```
ch find_recipient = <:  M @ R W @ R :> W @ R;
```

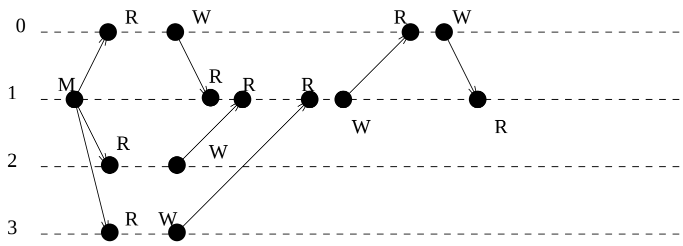
can match both the execution graph shown in Figure A.8.

The first graph is the one that the user expected, and the second graph is a wrong execution where the process does not wait to gather the responses before choosing the actual recipient. Such *scatter-gather* patterns are common in parallel programs, but cannot be described fully in Ariadne. The reason is simple, we have used augmented regular expression with *Push* and *Pop* directives to facilitate specification of trees, however the *scatter-gather* pattern is a directed acyclic graph. We will discuss how to handle such directed acyclic graphs in a two-step matching process when we discuss the `match` statement.

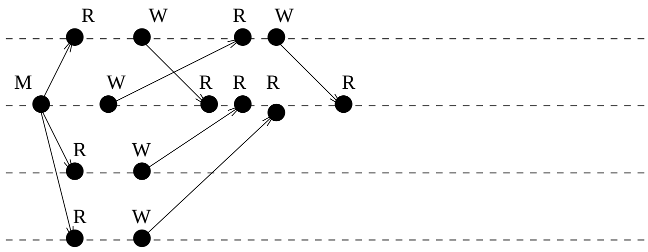
P-chain Descriptions

P-chains bind a behavior (chain description) to a set of processes and describe concurrent execution of a set of chains. The syntax for `p-chain` is as follows:

```
<pch-name> = <ch-desc> <onword> <set-desc>;
<ch-desc> = <chain-name> | <chain-expr>;
```



(a) Execution graph matching the user's expectation



(b) Execution graph that does not match what the user expected

Figure A.8. Scatter-gather patterns in chain description. Execution graph that matches user expectation (a); and one that does not match (b).

```
<onword> = ONALL | ONSOME;
<set-desc> = <set-name>;
```

The following p-chain definition

```
pch comm = msg onall procset;
```

binds the chain description *msg* to the process set named *procset*.

There are two distinct patterns that a p-chain can match – one signified by the keyword `onall` which said the *chain* has to match on all processes in the given set. The other keyword is `onsome` that signals that the p-chain needs to match on atleast one process in the given set. This helps us model data dependent communication pattern, where the number and identity of processes taking part in a canonical behavior cannot be known beforehand.

Pt-chain Descriptions

Pt-chains are patterns over p-chains representing sequences of concurrent events over time. The syntax is much like the regular expression-based syntax of chain definition, but does not include the `@`, `<:` or `:>` symbols. Instead three different angular brackets, – < > to indicate the **precedes** relationship holds between the enclosed abstract events, |[ ]| to indicate that the enclosed abstract events are pairwise **parallel**, and { } to indicate that the user does not care about the relationship between the enclosed abstract events – are present. The syntax is:

```
ptch <ptch-name> = <ptch-expr> [wrt <wrt-list>];
```

Other Facilities

In order to be useful, Ariadne has to provide other commands for doing post-mortem event-based debugging. These include loading of trace files, moving the focus of attention to different phases of the computation trace, doing coarse grained static filtering etc.

1. Loading of trace file is accomplished by the command

   ```
   load <trace-file-name>;
   ```

   Each entry in the trace file is supposed to contain the pid of the process where the event occurs, the pid of the other process (if any) involved in that event, the event type, and any user defined tag associated with the event. Ariadne assumes *processor consistency* model of parallel computation – namely between two processes messages are received in the order that they were sent. It automatically calculates *Lamport Timestamp* for events based on this assumption, and can handle *dummy* events used to handle edge conditions.

2. Matching is done when Ariadne is issued the command
   match <match-variable> = <Pt-Chain>; This matches the **Pt-Chain** expression, and assigns the matching abstract event tree structure to the **match-variable**.

   We will now discuss how to match the *scatter-gather* directed acyclic graph structure discussed previously. Note that the DAG in question can be considered as a superposition of two rooted trees – one signifying the causal ordering, and one the program-order of the sending process. Thus we can ensure that the pattern matched is indeed the one the user intended, by doing a two-step matching.

   ```
   ch co_find = <: M @ R :> W @ R;
   pch CoFind = co_find onsome 0..3;
   match first = CoFind;
   ch po_find = M R+ W @ R;
   pch PoFind = po_find onsome 0..3;
   match second = PoFind with [ first ];
   ```

   The last statement does the matching, but it only considers the events that were consumed by the match variable **first**. Thus the second matching acts as an assertion. If it succeeds, then the user knows that everything has gone well. If it fails then something is clearly wrong.

3. Coarse grained filtering is achieved using cursors. Ariadne allows the user to limit the focus of attention by using the *left* and *right* cursors. These are events that can be used to delineate the area of interest for the purposes of behavioral modeling. Only the events between the *left* and the *right* cursors are used for modeling purposes. Ariadne allows user to divide the entire execution trace in different *phases*. The user can move the cursors through commands
   `mlc Phase Cold 1`
   moves the left cursor to the first `Phase` event in each process tagged as *Cold*. By default the *left* cursor is stationed at the beginning of the trace file. The command
   `mrc Phase Hot 3`
   moves the *right* cursor to the third `Phase` event in each process tagged as *Hot*. By default the *right* cursor is stationed at the end of the trace file.

Ave: Query Language

Ave uses scalable visualization techniques to give hierarchical feedback on matches. We describe Ave's query language and its pre-defined functions.

Pre-defined functions

The scalable feedback of Ave can be used as a tree-structured spreadsheet to facilitate functional query on the match. Some of the pre-determined functions are listed below.

- `count ( <abstract_event_name> ) wrt <wrt_list>`
  counts the number of events of type specified in the `wrt_list` in the selected `abstract_event_name`. Returns an integer.

- `owner ( <abstract_event_name> ) wrt <wrt_list>`
  returns a set of process identifiers that idenitifies the processes where the events of type `wrt_list` belonging to `abstract_event_name` occurred.

- `abs( <list> )`
  returns a list of numbers that contain the absolute values of `list`.

- `prod ( <list> )`
  returns a scalar that is the product of the elements in the `list`.

- `sum ( <list> )`
  returns a scalar that is the sum of the elements in the `list`.

- `distinct ( <list> )`
  returns 1 if all the elements in the list are distinct, 0 otherwise.

- `equal ( <list> )`
  returns 1 if all the elements in the list are equal, 0 otherwise.

- `leq ( <list> )`
  returns 1 if all the elements taken in order from `list` are ordered by the relation $\leq$.

- `ltc ( <list> )`
  returns 1 if all the elements taken in order from `list` are ordered by the relation $<$.

- `geq ( <list> )`
  returns 1 if all the elements taken in order from `list` are ordered by the relation $\geq$.

- `gtc ( <list> )`
  returns 1 if all the elements taken in order from `list` are ordered by the relation $>$.

- `parent (<abstract_event>)`
  returns the parent of `abstract_event`.

- `child (<abstract_event>, <which>)`
  returns the `which`-th child of `abstract_event`. The `which` argument is optional and defaults to the first child. `which` can be any expression. *last* is a ident reserved to denote the *last* child of an abstract event.

- `sibling (<abstract_event>)`
  returns all the siblings of `abstract_event`.

- `left (<abstract_event>, which)`
  returns the `which`-th sibling to the left of `abstract_event`.

- `right (<abstract_event>, which)`
  returns the `which`-th sibling to the right of `abstract_event`.

Incremental Attribute Computation

Ave allows the user to compute additional attributes for abstract events once the matching is done. The syntax of the command is

```
with <abstract-event> foreach <descendant-event>
compute <attribute-name> = <expression>;
```

which computes an attribute named `attribute-name` according to the evaluation of the `expresion` foreach descendant of `abstract-event` named by `descendant-event`.

Partitioning of Behavior

Each abstract event (a node in the match tree visualized by Ave) consists of a set of abstract/primitive events (children of the node in the match tree). A user can partition the set of children based on some attribute. Each partition must be given a name. The Ave-induced default partitioning at the p-chain level is assigned the name *default.*

The syntax is

```
with <abstract-event> foreach <descendant-event>
create partition <partition-name> wrt <attribute-name>
```

creates a partition named **partition-name** foreach descendant of **abstract-event** named **descendant-event** with the **attribute-name** as a predicate.

The user can choose to display the match tree using the new additional structure created due to the partition by the command

```
view <abstract-event> through partition <partition-name>;
```

Setting breakpoints

The syntax is:

```
      break <when>(<abstract-event>);
<when> = before | after
```

<abstract-event> can be the name of a match variable or the identifier **select** that points to the abstract event selected by clicking the mouse button.

The statement

```
break after(select);
```

sets a consistent global breakpoint *after* the selected abstract event.

The user can place any number of breakpoints as long as the global states resulting from two breakpoints do not conflict.

Setting of GUI Parameters

The user can set certain parameters related to *Drawing Canvas* window. All parameters are integers. The syntax is:

```
const <param-name> = <param-value>;
```

The list of parameter values and their significance is given below.

1. max_color: maximum number of colors to be used for display purposes. Default value is 8. The absolute maximum allowed currently is 16.
2. color_collapsed: color to be used to display collapsed node. Default is 2 (green).
3. color_error: color to be used to highlight error. Default is 1 (red).
4. color_normal: color to be used for regular nodes. Default is 0 (blue).
5. scatter_plot_x_sep: separation between rectangles within a scatter plot. Default is 2.
6. scatter_plot_y_sep: vertical separation between rectangles within a scatter plot. Default value is 2.
7. scatter_plot_max_x_dim: Maximum number of pixels alloted along the x-axis to draw one scatter plot. Default is 128.
8. scatter_plot_max_y_dim: Maximum number of pixels alloted along the y-axis to draw one scatter plot. Default is 128.
9. line_plot_x_sep: Number of pixels between two points on a line plot.
10. ctg_stat_x_sep: Number of pixels between two bars in category statistics.
11. ctg_stat_width: Width of each bar in category statistics.
12. ctg_stat_height: Width of each bar in category statistics.

Trace File: Formats and Statements

The trace file must be a single file. The first line in the trace file should be:

`<version-num> <max-proc-id>`

The default `version-num` is 1. In order to change the value use

`const version_num = <new-value>;`

`max-proc-id` gives the maximum process identifier in the trace file.

The other entries in the trace file contains 7 fields.

`<pid> <oid> <mtype> <mtag> <msize> <bpflag> <iflag>`

where

1. `pid`: process identifier where the event took place
2. `oid`: identifier of any other process related to the event. For a `Write` event, this will be the identifier of the process where the message was sent. For a `Read` event, this will be the identifier of the process that sent the message.

3. **mtype**: message type – Read, Write, Multicast, Other.

4. **mtag**: user defined tag for the event.

5. **msize**: size of the message. Can be 0 for non-communication events.

6. **bpflag**: breakpoint flag

7. **iflag**: used for instrumentation. If non-zero there are **iflag** number of additional attributes associated with the event. The user is supposed to notify Ariadne of additional attributes – their names, and the events they are attached to. If nothing is notified, the attributes are ignores.

How to declare additional attributes

The first five fields **pid, oid, mtype, mtag, msize** are primitive attributes of an event. But Ariadne allows the user to add attributes to communication events in an effort to capture state information. One can, for instance, add the values of variables `x` and `y` at the local thread every time a message is received. The user can inform Ariadne about this, by specifying

```
add int x, y to R;
```

which adds the new attributes `x` and `y` to each `R` event. The syntax is

```
add <type> <attr-list> to <event-list>;
```

Also, the message tags related to each message vary from program to program. So the user needs to tell Ariadne, what message tags to expect. This can be done with

```
add utag <tag-name>;
```

Miscellanous Statements

Comments

Comments in Ariadne follows the Modula-III scheme, mainly because of its ability to nest comments. Thus `(*` signify the start of comment, and `*)` signify the ned.

for statement

This also follows the Modula-III scheme. The syntax is

```
for (<id> = <low> to <up> [<by> <int>]) begin <statement-list> end
```

restore statement

Restores already matched events to the event list. The syntax is

```
restore (<abstract-event>);
```

print statement

Prints contents of various names. Syntax is

```
print (<ident>);
```

BIBLIOGRAPHY

[1] Abrams, M., Doraswamy, N. and Mathur, A. Visual analysis of parallel and distributed programs in time, event and frequency domains. *IEEE Trans. on Parallel and Distributed Systems*, Vol. 3, No. 6, pages 672-685, November 1992.

[2] Adams, E. and Muchnick. S. S. Dbxtool: A window based symbolic debugger for Sun workstations. *Software Practice and Experience*, Vol. 16, No. 7, pages 653-669, July 1986.

[3] Ahuja, S., Carreiro, N. J., and Gelernter, D. Linda and Friends *Computer*, Vol. 19, No. 8, pages 26-34, August 1986.

[4] Alverson, G., Griswold, W., Notkin, D. and Snyder, L. A flexible communication abstraction for nonshared memory parallel computing. *Proceedings of Supercomputing '90*, 1990.

[5] Bacon, D. F., and Goldstein, S. C. Hardware-assisted replay of multiprocessor programs. *ACM SIGPLAN Notices*, Vol. 26, No. 12, pages 194-206, Dec 1991.

[6] Baeten, J. C. M., and Weijland, W. P. *Process Algebra.* Cambridge University Press, *Cambridge Tracts in Theoretical Computer Science*, 1990.

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

[8] Baiardi, F., De Fransesco, N. and Vaglini, G. Development of a debugger for a concurrent language. In *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 4, pages 547-553, April 1986.

[9] 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.

[10] Bingham, T., Hobbs, N., and Husson, D. Experiences developing and using an object-oriented library for program manipulation. *ACM SIGPLAN Notices*, Vol. 28, No. 10, pages 83-89, October, 1993.

[11] Brinskma, E. A tutorial on LOTOS. In M.Diaz, ed. Protocol Specification, Testing and Verification, V, pages 171-194, Elsevier, 1986.

[12] Brown, M. H., and Sedgewick, R. A system for algorithm animation. *Computer Graphics*, Vol. 18, No. 3, pages 177-186, Mar 1984.

[13] Bruegge, B. and Hibbard, P. Generalized path expressions: A high level debugging mechanism. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium in High-Level Debugging*, pages 34-44, 1983.

[14] Carchiolo, V., Di Stefano, A, and Faro, A. ECCS and LIPS : Two Languages for OSI systems specification and verification. *ACM Transactions on Programming Languages and Systems,* Vol. 11, No. 2, pages 284-329, April 1989.

[15] Chandy, K. M., and Lamport, L. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, Vol. 3, No. 1, pages 63-65, February 1985.

[16] Choi, J. D., and Min, S. L. Race frontier: reproducing data races in parallel program debugging. *Proceedings of PPoPP 1991.*

[17] Cleaveland, L, Parrow, J., and Steffen, B. The Concurrency Workbench: A Semantics based Tool for the Verification of Concurrent Systems. *Proceedings of the Workshop on Automatic Verification Methods for Finite State Machines,* pages 24-37, February 1991.

[18] Cooper, R., and Marzullo, K. Consistent detection of global predicates. *ACM SIGPLAN Notices,* Vol. 26, No. 12, pages 167-174, 1991.

[19] Couch, A. A. Categories and context in scalable execution visualization. *Journal of Parallel and Distributed Computing*, Vol. 18, pages 195-204, 1993.

[20] Couch, A. L., and Krumme, D. W. Multidimensional spreadsheets in a graphical symbolic debugger for the Ncube. *Proceedings of the 6th Distributed memory Conference*, pages 205-209, 1993.

[21] Cuny, J. E., Hough, A. A., and Kundu, J. Logical time in visualizations produced by parallel programs. *Proceedings of Visualization '92*, pages 186–193 (1992).

[22] Cuny, J. E., Forman, G., Hough, A., Kundu, J., Lin, C., Snyder, L., and Stemple, D. The Ariadne Debugger: Scalable Application of Event-Based Abstraction. *ACM SIGPLAN Notices*, Vol. 28, No. 12, pages 85–95, 1994.

[23] Di Maio, A., Ceri, S., and Crespi-Reghizzi, S. Execution monitoring and debugging tool for Ada using relational algebra. *Proceedings of Ada in use, Ada International Conference*, pages 109-123, May 1985.

[24] Emrath, P. A., and Padua, D. A. Automated detection and nondeterminacy in parallel programs. In *ACM SIGPLAN Notices,* Vol. 24, No. 1, pages 89–99, 1989.

[25] Feldman, S., and Brown, C. IGOR: A system for program debugging via reversible execution. *ACM SIGPLAN Notices,* Vol. 24, No. 1, pages 112-123, 1989.

[26] Fidge, C. J. Partial orders for parallel debugging. *ACM SIGPLAN Notices*, Vol. 24, No. 1, pages 183–194, 1989.

[27] Forin, A. Debugging of heterogenous parallel systems. In *ACM SIGPLAN Notices,* Vol. 24, No. 1, pages 130–140, 1989.

[28] 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. *ACM SIGPLAN Notices*, Vol. 24, No. 1, pages 163–173, 1989.

[29] Gait, J. A debugger for concurrent programs. *Software Practice and Experience*, Vol. 15, No. 6, pages 539-554, 1985.

[30] Garcia, M. E., and Berman, W. J. An approach to concurrent systems debugging. Proc. of the 5th International Conference on Distributed Computing Systems, pages 507-514, 1985.

[31] Garcia-Molina, H., Germano, F., and Kohler, W.H. Debugging a distributed computing system. *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 2, pages 210-219, March 1984.

[32] Godefroid, P., and Wolper, P. Using partial orders for the efficient verification of deadlock freedom and safety properties. In K.G. Larsen and A. Skou, eds., Computer Aided Verification, LNCS 575, Springer-Verlag, New York, NY, pages 332-342, 1991.

[33] Goldberg, A.P., Gopal, A., Lowry, A., and Strom, R. Restoring consistent global states of distributed systems. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 144-154, May 1991.

[34] Greenberg, M., and Cuny, J.E. Parallelism in knowledge-based systems with inheritance *Proceedings International Conference on Parallel Processing*, Vol. III, pages 141-145, 1988.

[35] Haban, D., and Weigel, W. Global events and global breakpoints in distributed systems. *21st Annual Hawaii International Conference on System Sciences*, pages 166-174, January 1988.

[36] 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.

[37] Hembold, D., and Luckham, D. Debugging Ada tasking programs. *IEEE Software*, Vol. 2, No. 2, pages 47-57, March 1985.

[38] Hoare, C. Communicating Sequential Processes.

[39] Hough, A. A. *Debugging Parallel Programs Using Abstract Visualizations.* PhD thesis, University of Massachusetts, Amherst, MA 01003, 1991. Also COINS Technical Report 91–53.

[40] Hough, A. A., and Cuny, J. E. Perspective views: A technique for enchancing visualizations of parallel programs. In *1990 International Conference on Parallel Processing*, pages II 124–132, Aug. 1990.

[41] 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.

[42] Hu, A. J., Dill, D., Crexler, A., and Yang, C. H. Higher Level Specification and Verification with BDDs. *Proceedings of the 3rd International Workshop on Computer Aided Verification*, Montreal Canada, July 1992.

[43] Hurfin, M., Plouzeau, N., and Raynal, M. Detecting atomic sequences of predicates in distributed computations. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Computing*, pages 32-42, May 1993.

[44] Intel Supercomputer Systems Division. Paragon Interactive Parallel Debugger Reference Manual. Paragon Documentation, 1994.

[45] Joyce, J. et al. Monitoring distributed systems. *ACM Transactions of Computing Systems,* Vol. 5, No. 2, pages 124-150, May 1987.

[46] Kundu, J. Ariadne User's Manual (in preparation)

[47] Kundu, J., and Cuny, J. E. A scalable, visual interface for debugging with event-based behavioral abstraction. In *Proceedings of New Frontiers on Massively Parallel Processing*, pages 472–479, February 1995.

[48] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, Vol. 21, No. 7, pages 558–565, 1978.

[49] Lauer, P. E., Torrigani, P. R., and Shields, M. W. COSY — A system specification language based on paths and processes. *Acta Informatica*, Vol. 12, pages 109-158, 1979.

[50] Leban, B. Representative Views of Parallel Computation. Unpublished Thesis Proposal, Department of Computer Science, University of Massachusetts, Amherst, 1993.

[51] 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.

[52] LeBlanc, T. J., and Mellor-Crummey, J. M. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, Vol. C-36, No. 4, pages 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*, Vol. 9, pages 203–217, 1990.

[54] Ledoux, C. H., and Stott-Parker, D. Jr. Saving traces for Ada debugging. *Ada in use, Ada International Conference*, pages 97-108, Paris 14-16, May 1985.

[55] Lin, C., and LeBlanc, R. Event-based debugging of object/action programs. SIGPLAN Notices, Vol. 24, No. 1, pages 23-34, Jan 1989.

[56] Lo, V., Rajopadhye, S., Mohamed, M., Gupta, S., Nitzberg, B., Telle, J., Zhong, X. LaRCS: A language for describing parallel computations for the purpose of mapping. Technical Report CIS-TR-90-16, University of Oregon Dept. of Computer Science, 1990.

[57] Long, D. L., and Clarke, L. A. Task interaction graphs for concurrency analysis. *Proceedings of the 11th International Conference on Software Engg.*, pages 44-52, May 1989.

[58] Manabe, Y., and Imase, M. Global conditions in debugging distributed programs. *Journal of Parallel and Distributed Computing*, Vol. 15, pages 62-69, January 1992.

[59] Mattern, F. Virtual time and global states of distributed systems. M. Cosnard et al, eds., "Parallel and Distributed Algorithms", Elsvier Science Publishers B. V. (North Holland), pages 215-226, 1989.

[60] McDowell, C. E. A practical algorithm for static analysis of programs. *Journal of Parallel and Distributed Computing,* Vol. 6, pages 515-536, 1989.

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

[62] Miller, B., and Choi, J.-D. A mechanism for efficient debugging of parallel programs. *SIGPLAN Notices*, Vol. 24, No. 1, pages 141–150, 1989.

[63] Miller, B., Macrander, S., and Sechrest, A. Monitoring distributed systems. *Software Practices and Experience*, Vol. 162, No. 2, pages 183-200, June 1985.

[64] Milner, R. Elements of interaction. *Communications of the ACM,* Vol. 36, No. 1, pages 78-89, January 1993.

[65] Moher, T. G. PROVIDE: A process visualization and debugging environment. *IEEE Transactions on Software Engineering*, Vol. SE-14, No. 6, pages 849-857, June 1988.

[66] Mohr, B., Brown, D., Malony, A. TAU: A Portable Parallel Program Analysis Environment for pC++, *Proc. of CONPAR 94 - VAPP VI, Linz, Austria, Springer Verlag, LNCS 854,* pages 29-40, September 1994.

[67] Netzer, R. H. B. Optimal tracing and replay for debugging shared-memory parallel programs. In *ACM SIGPLAN Notices,* Vol. 26, No. 12, pages 1–11, 1991.

[68] Netzer, R. H. B., and Miller, B. P. Improving the accuracy of data race detection. *Proc. of the 3rd ACM Symposium on Principles and Practice of Parallel Programming*, pages 133-141, April 1991.

[69] Netzer, R. H. B., and Xu, J. Incremental replay of message-passing programs. *Proceedings of Supercomputing '93*, pages 840-849, 1993.

[70] Olsson, R. A., Crawford, R. H., and Ho, W. W. A dataflow approach to event-based debugging. *Software Practice and Experience,* Vol. 21, No. 2, pages 209-229, Feb 1991.

[71] Pan, D. Z., and Linton, M. A. Supporting reverse execution for parallel programs. SIGPLAN Notices, Vol. 24, No. 1, pages 124-129, Jan 1989.

[72] Scheuermann, P., Yaagoub, A., and Aris-Ouksel, M. Compression of binary images on a hypercube machine. *Journal of Parallel and Distributed Computing,* Vol 23, pages 49-59, 1994.

[73] Sequent Computer Systems, Inc. *DYNIX Pdbx Debugger User's Manual.* 1986.

[74] Sequent Computer Systems, Inc. *Guide to Parallel Programming on Sequent Computer Systems.* 1987.

[75] Perl, S., and Weihl, W. Performance assertion checking. *Proc. of the ACM Symposium on Operating System Principles*, pages 134-145, December 1993.

[76] Peterson, J.L. Petri Nets. *Computing Surveys*, Vol. 9, No. 3, pages 223-252, 1977.

[77] Reiss, S. PECAN: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, pages 276-285, March 1985.

[78] Roman, G.-C., and Cox, K.C. A declarative approach to visualizing concurrent computations. *Computer*, Vol. 22, No. 10, pages 25-36, October 1989.

[79] Roman, G.-C., and Cox, K.C. A taxonomy of program visualization systems. *Computer*, Vol. 26, No. 12, pages 11-24, December 1993.

[80] Rubin, R. V., Rudolph, L., and Zernik, D. Debugging parallel programs in parallel. In *ACM SIGPLAN Notices,* Vol. 26, No. 12, pages 216–225, 1991.

[81] Sarukkai, S., and Gannon, D. SIEVE: A performance debugging environment for parallel programs. *J. of Parallel and Distributed Computing,* Vol. 18, No. 2, pages 147-168, June 1993.

[82] Smith, E. T. A debugger for message based processes. *Software Practice and Experience*, Vol. 15, No. 11, pages 1073-1086, November 1985.

[83] Snodgrass, R. A relational database approach to real-time monitoring. *ACM Transactions on Computer Systems*, Vol. 6, No. 3, pages 157-196, May 1988.

[84] Snyder, L. The XYZ abstraction levels of Poker-like languages. *Languages and Compilers for Parallel Computing*, David Gelernter and Alexandru Nicolau and David Padua(eds.), MIT Press, pages 470–489, 1990.

[85] Socha, D., Bailey, M., and Notkin, D. Voyeur: Graphical views of parallel programs. *SIGPLAN Notices,* Vol. 24, No. 1, pages 206-215, 1989.

[86] Spezialetti, M., and Kearns, J. A general approach to recognizing event occurences in distributed computations. *Proceedings of the 8th International Confrenece of Distributed Computing Systems*, pages 300-307, July 1988.

[87] Stallman, R. M. *GDB Manual (The GNU Source-Level Debugger).* Free Software Foundation, Cambridge, MA.

[88] Stasko, J. T. Tango: A framework and system for algorithm animation. *Computer*, Vol. 23, No. 9, pages 27-39, September 1990.

[89] Stone, J. A graphical representation of concurrent processes. In *ACM SIGPLAN Notices,* Vol. 24, No. 1, pages 226–235, 1989.

[90] Tai, K. C., Carver, R. H., and Obaid, E. E. Debugging concurrent Ada programs by deterministic execution. *IEEE Trans. of Software Engineering*, Jan 1991.

[91] Taylor, R. A general purpose algorithm for analyzing concurrent programs. *Comm. of the ACM,* Vol. 26, No. 5, pages 363-376, May 1983.

[92] KSR Corporation Udb: User manual, KSR Corporation. 1993

[93] Valmari, A. Eliminating redundant interleavings during concurrent program verification. In E. Odjik, M. Rem and J. C. Syre, eds, *Parallel Architecture and Language (PARLE)*, LNCS 266, Springer-Verlag, New York, NY, pages 89-103, 1989.

[94] Wittie, L. Debugging distributed C programs by real time replay. *ACM SIGPLAN Notices,* Vol. 24, No. 1, pages 57–67, 1989.

[95] Yeh, W. J., and Yopung, M. Compositional Reachability Analysis Using Process Algebra. *Proceedings of the ACM SIGSOFT Symposium on Testing, Analysis and Verification (TAV4)*, pages 49-59, October 1991.

[96] Zave, P. A distributed alternative to finite state machine specifications. ACM *Transactions on Programming Languages and Systems*, Vol. 7, No. 1, pages 10-36, Jan 1985.

[97] Zernik, D., Snir, M., and Malki, D. Using visualization tools to understand concurrency. *IEEE Software*, pages 87-92, May 1992.