A Scalable, Visual Interface for Debugging with Event-Based Behavioral Abstraction

Joydip Kundu

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Janice E. Cuny

Department of Computer Science
University of Oregon
Eugene, OR 97403

Abstract

Event-based behavioral abstraction, in which models of intended program behavior are compared to actual program behavior, offers solutions to many of the debugging problems introduced by parallelism. Currently, however, its widespread application is limited by an inability to provide sufficient feedback on the mismatches between intended and actual behaviors, and an inability to provide output that scales for large or complex systems. The AVE/Ariadne debugging system was developed to address these limitations. Ariadne is a *post mortem* debugger that combines a simple modeling language with functional queries to support thorough exploration of execution traces. AVE is a visual interface to Ariadne that provides scalable, visual feedback. AVE features hierarchical visualizations that reflect the structure of user-defined behavioral models, dynamic attribute calculation, and automatic partitioning of matched behaviors and attributes.

1 Introduction

The debugging of parallel programs is often difficult. Standard state-based techniques [20, 28] are not effective because of the sparsity of consistent global states, the large volume of trace data, the effects of asynchrony, and the lack of reproducibility. 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 [2, 3, 16, 19]. The use of event traces with replay mechanisms avoids issues of global state consistency [5, 27] 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 [8, 23]. Currently, however, event-based parallel debugging tools are limited because (1) they can not provide sufficient feedback on the mismatches between intended and actual behaviors, and (2) their outputs do not scale well for large or complex programs.

The Ariadne[1] debugger [9] was developed to address the first of these limitations. Most existing tools that support event-based behavioral abstraction use a powerful, expressive modeling language that requires complex recognition strategies with only "match/no match" outcomes. 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, Adriadne'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), introduced in this paper, was developed to address the second of these limitations: it provides scalable visualizations of debugging output. Existing parallel debuggers use either process-oriented or event-oriented visualizations. Process-oriented visualizations depict event activity on a set of interconnected process icons [16, 31] and, thus, are limited to systems with relatively small numbers of processes. Event-oriented visualizations allow the user to scroll through event traces or process-time graphs [11, 13, 14, 25] but typically do not show user-defined abstractions. Here we present AVE, a scalable graphical interface for visualizing abstract events that features

[1]In Greek mythology, Ariadne provided Theseus with the thread that enabled him to find his way through the Labyrinth to slay the Minotaur.

- hierarchical visualizations reflecting the structure of user-defined behavioral models,
- incremental, dynamic attribute calculation, and
- automatic partitioning of matched behaviors and attributes.

The *EBBA* toolset also provides visualizations of match trees but it does not have automatic partitioning and dynamic attribute calculation [4]. Issues related to scalable visualization have been explored in the context of performance visualization in other systems [1,6,27]. None of these, however, deal with the multi-dimensionality of logical time or provide complex user-defined abstraction mechanisms.

In Section 2, we briefly describe Ariadne and in Section 3, we demonstrate its use with the AVE interface by stepping through several sample debugging sessions.

2 The Ariadne Debugger

AVE/Ariadne is a *post-mortem*, event-based debugger designed to support the initial stages of debugging for correctness in parallel systems. It facilitates the exploration of execution traces stored internally as *execution history graphs* in which nodes represent events and edges represent Lamport's *happened before* relation [21]. Primitive events include all of the standard interprocess communication events such as *send*, *receive*, and *barrier synchronization* as well as user-specified events such as procedure entries and exits. Eventually Ariadne will have replay capabilities [24] and a facility for incremental instrumentation and re-execution; at the moment, our instrumentation is hand-coded. Ariadne matches models of expected behavior against the execution history graph in order to extract matched or partially matched subgraphs. These subgraphs are then explored through further modeling, functional queries, and manipulations of logical, temporal orderings.

In the next three subsections, we describe Ariadne's modeling language, its query facility, and its use of logical orderings.

2.1 Ariadne's Modeling Language

Ariadne's modeling language is quite simple. Communication patterns are described in terms of a three-level 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 an execution history graph, all events in the chain must occur exactly in the specified order (with the exception of communication events that are not physically realized because of "edge effects" on process array boundaries).
- *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.
- *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 (as discussed below).

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 [30] and the LaRCS specification language [26]. 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[2], avoiding the costliness of pattern matching approaches [3, 15] and the expensive implementations (number of states can potentially be exponential in the length of the parallel expression to be recognized) of previous languages [19]. At the same time, our matching algorithm can provide precise information on the reasons for a match failure.

2.2 Ariadne's Query Facilities

When a pt-chain is matched or partially matched, its events are returned as a *match tree* in which internal nodes represent user-defined abstractions and leaves represent primitive events. 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. To explore these match trees, Ariadne provides a set of functions that return

[2] In most cases the matching can be done in linear time, although the verification of the logical time relationships between abstract events may take polynomial time.

match characteristics providing answers to questions such as

> Did all events occur in parallel?
> Which processes were the destinations of a Write in this match?
> Which processes matched the first Read of the pattern?
> Why did the match fail on process 12?

All functions in Ariadne "walk" match trees. The user can build new functional queries simply by defining a value for the function at nodes in the tree.

2.3 Logical Time Manipulation in Ariadne

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. We extend Lamport's relation to sets of primitive events; defining three relations: **precedes**, **parallels**, and **overlaps** [17] 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 [10, 12, 19, 22] but we have found our definitions to be more appropriate for debugging. In particular, our **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.

3 Sample Debugging Sessions

In this section, we introduce the AVE interface and demonstrate its use with examples from three debugging sessions.

3.1 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 respectively as **R_pivot**, **W_pivot**, **R_elim**, and **W_elim**. 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;**

where **ch** is a keyword signaling a chain definition. Similarly, each process in the elimination step reads the pivot row and then reads and writes each of the rows for which it is responsible. This is modeled as

> **ch elim = R_elim (R_elim W_elim)+ ;**

Putting these together, the behavior of a process during a single iteration of the program is

$$\textbf{pivot} * \ \textbf{elim}$$

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;**

where **set** and **pch** are keywords. The first line defines **procset** to be the set of process ids (PIDs), in this case, the integers from 0 to 8. The second line defines a p-chain **iter** that binds the process behavior to a nonempty subset of processes as indicated by the keyword **onsome**.

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 **match** is a keyword, here indicating 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 1 shows the result of the match.

Match trees are generally quite large; even in this simple example, the full tree has 239 nodes. As a result, AVE automatically compresses trees for visualization. The compression occurs both horizontally and vertically: horizontally by eliding sibling nodes and vertically by pruning levels. Suppressed sibling nodes are represented by a single node with button that can be used for scrolling through hidden constituents; an integer label gives the number of these constituents. Vertical compression is indicated by a summarizing box. Here, the tree is expanded only to the p-chain level as indicated by the boxes below each p-chain. Each box represents an entire subtree rooted at the p-chain node. It is labeled with an integer pair: the first component indicates the total number of behaviors (that is, the number of direct descendents of the p-chain node) and the second component indicates the number of different behaviors. In this case, nine processes participated in the first iteration and they all had the same behavior because there was no pivoting and each process was responsible for just one row. The summarizing label was **9,1**. During the second iteration, eight processes participated but the parent performed a pivot and so there were two different behaviors: the parent executed **pivot* elim** and the remaining processes executed just **elim**. The summarizing label was **8,2**. The user can get a description of the partitioning by clicking on the summarizing node as shown in Figure 1. Here the user clicked on the summarizing node labeled **8,2** (as indicated by the black shading) to get the further information that process 0 had the behavior

pivot^1 elim

and processes 1 through 7 had the behavior

elim

.

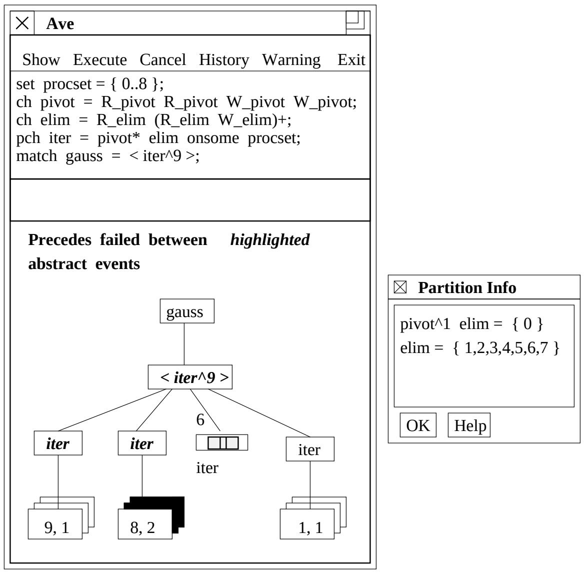

Figure 1: Ave feedback showing **precedes** failure in Gaussian Elimination.

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

Precedes failed between highlighted abstract events

This means that although the expected chains occurred in the execution history graph, the expected logical relations, in this case **precedes**, did not hold between them. The highlighting (bold italics [3]) on the first two instances of **iter** indicates the first place in which it failed to hold. 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.)

3.2 FFT on a Mesh.

This program is an MIMD, message-passing version of the FFT on a mesh with static data partitioning.

[3] In the actual debugger, where we have access to color, the highlighting appears in red.

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 consisting of a send (denoted by the **W**) connected by an edge (denoted by the **@**) to the receive of that same message (denoted by the **R**). The p-chain **comm** binds **exch** to the set of proceses 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.

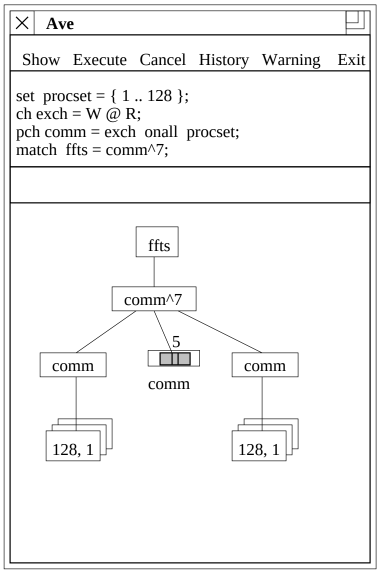

Figure 2: AVE feedback in the FFT example.

In this example, the match was successful and it resulted in the assignment shown in Figure 2. The visualization is a compressed representation the match tree: the full tree has 7 **comm** nodes, 128×7 interior **exch** nodes, and $2 \times 128 \times 7$ leaf nodes each labeled with **W** or **R**.

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 facilities in AVE that enable the dynamic computation of attributes over a match tree. The command

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

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)**). To display these attributes, we use

with ffts show dist(exch) wrt comm;

which results in a scatter plot for each **comm** phase as shown in the window *Drawing Canvas* of Figure 3.

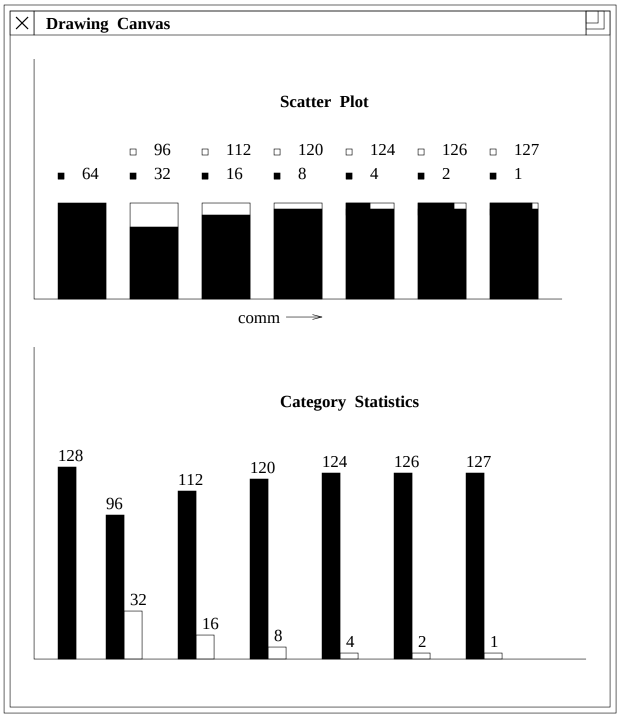

Figure 3: AVE feedback in the FFT example.

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*} [7]. Here, because of reproduction limitations, we set the number of colors to 2 so only black and white were used.

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 [4].

3.3 Dynamic Task Allocation.

This program was taken directly from the *Sequent Programming Guide* [29]. 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 process 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 Figure 4. Notice that the behavior was again compressed at the p-chain level.

To look at the load balancing properties of the program, we computed the following attributes to give 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 Figure 4 (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%!

[4] The fact that there are 96 occurences of the value 32, and 32 occurences 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.

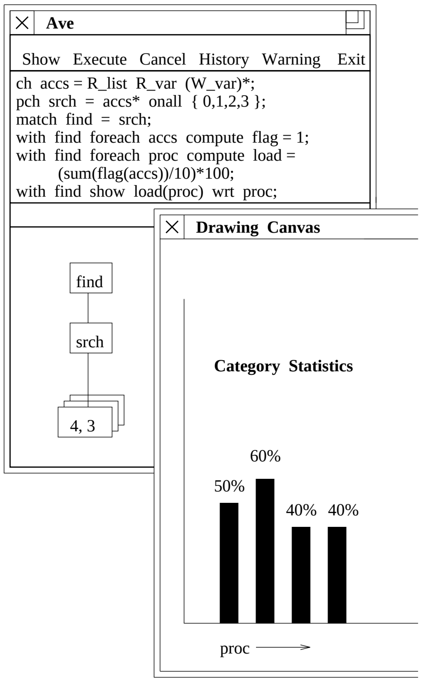

Figure 4: Ave feedback showing that the load is more than 100%.

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

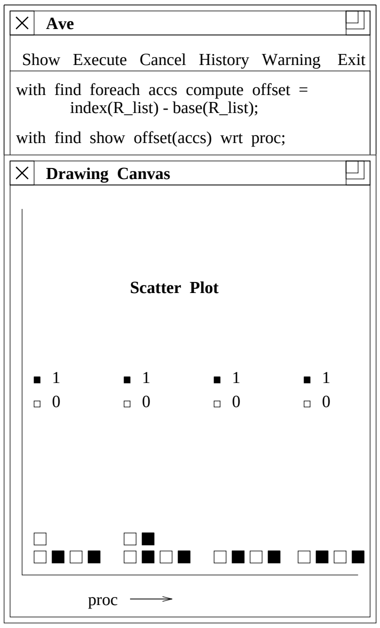

Figure 5: Ave feedback showing that variable **base** took two values in each iteration.

4 Conclusion

This paper introduces the AVE/Ariadne debugging system which provides a new, scalable application of event-based behavioral abstraction to the debugging of parallel computations. AVE provides visualizations of the match tree that free the user from the layout and space limitations of process-oriented approaches. At the same time, AVE imposes the user's own abstractions on the primitive events of an execution trace making it a more powerful, high-level tool than most existing event-based debuggers that rely on low-level displays of a computation. Our interface provides scalability by automatically compressing and abstracting match trees based on the structure of the modeling hierarchy. In addition, it supports the calculation of additional attributes for the tree, displaying a partitioning of the generated attribute domain in terms of color. Ariadne can also repartition the displayed tree structure based on attribute values (not shown here) with a mechanism similar to *set comprehension* that is available in functional languages like *Haskell* [18]. This feature allows the user to determine whether a set of events satisfy a number of different constraints. All functions in Ariadne walk the same *match tree*, making it easy to add new functional queries.

Currently, we are investigating the integration of replay mechanisms into Ariadne in order to support incremental instrumentation to collect state information at any level of detail.

Acknowledgements

George Forman, Alfred Hough, Calvin Lin, Lawrence Snyder, and David Stemple all contributed to the initial design of Ariadne.

References

[1] M. Abrams, N. Doraswamy, and A. Mathur. Chitra: visual analysis of parallel and distributed programs in time, event and frequency domains. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3, No. 6, pages 672-685, Nov. 1992.

[2] F. Baiardi, N. De Fransesco, and G. Vaglini. Development of a debugger for a concurrent language. In *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 4, pages 547–553, Apr. 1986.

[3] P. C. Bates. *Debugging Programs in a Distributed System Environment.* PhD Thesis, University of Massachusetts, 1986.

[4] P. C. Bates, private communication.

[5] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 167–174, 1991.

[6] A. Couch. *Graphical Representations of Program Performance on Hypercube Message-Passing Multiprocessors*, Ph.D. Thesis, Tufts University, 1988.

[7] A. Couch, "Categories and Context in Scalable Execution Visualization," *Journal of Parallel and Distributed Computing*, Vol. 18, pages 195-203 (1993).

[8] J. E. Cuny, A. Hough, and J. Kundu. Logical time in visualizations produced by parallel programs. *Proceedings of Visualization '92*, pages 186–193 (1992).

[9] J. E. Cuny, G. Forman, A. Hough, J. Kundu, C. Lin, L. Snyder, and D. Stemple. The Ariadne debugger: scalable application of event-based abstraction. *SIGPLAN Notices*, Vol. 28, No. 12, pages 85–95, 1994.

[10] C. J. Fidge. Partial orders for parallel debugging. *SIGPLAN Notices*, Vol. 24, No. 1, pages 183–194, 1989.

[11] R. J. Fowler, T. J. Leblanc, and J. M. Mellor-Crummey. An integrated approach to parallel program debugging and performance analysis on large-scale multiprocessors. *SIGPLAN Notices*, Vol. 24, No. 1, pages 163–173, 1989.

[12] D. Haban, and W. Weigel. Global events and global breakpoints in distributed systems. *21st Annual Hawaii International Conference on System Sciences*, pages 166-174, January 1988.

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

[14] M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, Vol. 8, No. 5, pages 29–39, 1991.

[15] D. Hembold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, Vol. 2, No. 2, pages 47-57, Mar. 1985.

[16] A. A. Hough. *Debugging Parallel Programs Using Abstract Visualizations.* PhD Thesis, University of Massachusetts, 1991.

[17] A. A. Hough and J. E. Cuny. Perspective views: A technique for enchancing visualizations of parallel programs. In *1990 International Conference on Parallel Processing*, pages II 124–132, Aug. 1990.

[18] P. Hudak, S. Peyton Jones, and P. Wadler. Report on the programming language Haskell, a non-strict purely functional language Version 1.2. *ACM SIGPLAN Notices*, Vol 27, No. 5, May 1992.

[19] W. Hseush and G. E. Kaiser. 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.

[20] Kendall Square Research. *UDB User's Guide.* KSR1 Documentation, Feb 1992.

[21] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, Vol. 21, No. 7, pages 558–565, 1978.

[22] L. Lamport. The mutual exclusion problem: Part I-A theory of interprocess communication. *Journal of the Association for Computing Machinery*, Vol. 33, No. 2, pages 313-326, April 1986.

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

[24] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, Vol C-36, No. 4, pages 471–482, Apr. 1987.

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

[26] V. M. Lo, S. Rajopadhye, M. A. Mohamed, S. Gupta, B. Nitzberg, J. A. Telle, X. X. Zhong. 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.

[27] Y. Manabe, and M. Imase. Global conditions in debugging distributed programs. *Journal of Parallel and Distributed Computing*, Vol. 15, pages 62-69, January 1992.

[28] Sequent Computer Systems, Inc. *DYNIX Pdbx Debugger User's Manual.* 1986.

[29] Sequent Computer Systems, Inc. *Guide to Parallel Programming on Sequent Computer Systems.* 1987.

[30] L. Snyder. 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.

[31] D. Socha, M. L. Bailey, and D. Notkin, Voyeur: graphical views of parallel programs. *SIGPLAN Workshop on Parallel and Distributed Debugging*, pages 206-215 (May, 1988).