The Ariadne Debugger: Scalable Application of Event-Based Abstraction[1]

Janice Cuny,[2] George Forman,[3] Alfred Hough,[4] Joydip Kundu,[2]
Calvin Lin,[3] Lawrence Snyder,[3] and David Stemple[2]
Revised June 1993

1 Introduction

Massively parallel computations are difficult to debug. Users are often overwhelmed by large amounts of trace data and confused by the effects of asynchrony. Event-based behavioral abstraction provides a mechanism for managing the volume of data by allowing users to specify models of intended program behavior that are automatically compared to actual program behavior [2, 3, 5, 14, 16]. Transformations of logical time ameliorate the difficulties of coping with asynchrony by allowing users to see behavior from a variety of temporal perspectives [7, 15, 19, 21]. Previously, we combined these features in a debugger that automatically constructed animations of user-defined abstract events in logical time [14]. However, our debugger, like many others, did not always provide sufficient feedback nor did it effectively scale up for massive parallelism. Our modeling language required complex recognition algorithms which precluded informative feedback on abstractions that did not correspond to observed behavior. Feedback on abstractions that did match behavior was limited because it relied on graphical animations that did not scale well to even moderate numbers of processes (such as 64). We address these problems in a new debugger, called Ariadne.[5]

Ariadne uses a simple language to specify behavioral abstractions as patterns of events in logical time. These patterns are detected in traces of program behavior by collections of small finite-state recognizers which allow substantive feedback on match failures. There are three salient features of Ariadne: The first is the ability to provide feedback on failures, the second is the scalability of its patterns and non-graphical output, and the third is the conciseness of its modeling language. These features, however, are accompanied by some loss of expressivity. The loss of expressivity means that patterns are often too coarse, matching behaviors in unintended ways. Ariadne compensates for this by providing functions that return the characteristics of matched behaviors. As an example, a user might match a number of multicasts in an execution trace and then use functional queries to determine which processes actually wrote values or where those values were sent. Ariadne's combination of pattern matches and functional queries allows the user to investigate an execution trace thoroughly.

Section 2 provides an overview of our approach and Section 3 provides several sample debugging sessions illustrating its capabilities.

2 Our Approach

Ariadne is a *post mortem* debugger for massively parallel, MIMD message-passing systems. It is designed for correctness debugging, and it supports the user in investigating global interprocess communication patterns.

Ariadne operates on traces produced by parallel programs.[6] Within these traces, processes, identified by integer process ids (PIDs), execute sequences of primitive events. The debugger currently supports four primitive event types: Read, Write, Multicast, and Phase Marker. Reads, Writes, and Multicasts denote interprocess communication, and Phase Markers denote the ends of logical phases of computation. The traces are stored internally in an *execution history graph* where the nodes represent events and the edges

[1] Partial support for this work was provided by the Office of Naval Research under contract N00014-89-J-1368, by the National Science Foundation under grant CCR-9023256, and by the Defense Advanced Research Projects Agency under DARPA project DAAL02-91-C-0051.

[2] Department of Computer Science, University of Massachusetts, Amherst, MA 01003

[3] Department of Computer Science and Engineering FR-35, University of Washington, Seattle, WA 98195

[4] Amerinex Artificial Intelligence Inc., Amherst, MA 01002

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

[6] Currently our traces are taken from a simulator or generated by hand.

represent communication events; from these traces we can derive Lamport's *happened before* relation [17]. A number of debuggers provide visualizations of execution history graphs [9, 11, 12, 21] but such visualizations do not scale well for massively parallel systems. Ariadne allows the user to view the graph but it does not rely on visualization. Instead, it supports interactive, textual explorations of the graph.

Here we briefly describe three aspects of this support: the modeling language, the facilities for manipulating logical temporal orderings, and the functions that are available for investigating the characteristics of pattern matches and mismatches.

2.1 The Ariadne Modeling Language

As mentioned above, previous attempts at using event-based abstraction in debugging have been limited by the complexity of the modeling language. Ariadne's language is quite simple. It employs a three level description of communication patterns in terms of *chains*, *p-chains*, and *pt-chains*.

- *Chains* are patterns representing "local views" of communication. They are described by extensions of regular expressions. When they are matched against an execution history graph, all events in the chain must occur exactly in the order specified with the exception of communication events that are not physically realized because of "edge effects" on process array boundaries [1].
- *p-Chains* are patterns representing the concurrent execution of a chain by a set of processes. They are described by binding a chain to a process set. When a p-chain is matched against a behavior, a copy of its chain is matched on each element of its process set (events can be shared across, but not within, chains).
- *pt-Chains* are patterns representing the logical, temporal composition of a set of p-chains. They are matched in a two step process: first events matching the p-chains are located in the graph and then the specified logical relations between those events are verified. When a pt-chain has been successfully matched, it returns an *abstract event* which is a structure containing the matched instances of events; these instances are removed from the trace and are unavailable for further matching unless explicitly restored.

These 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 [24] and the LaRCS specification language [22]. 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, avoiding the costliness of pattern matching approaches [3, 13] and the expensive implementations of previous languages [16]. At the same time, our matching algorithm can provide precise information on the reasons for a match failure.

2.2 Logical Time Manipulation in Ariadne

Programmers are often confused by the results of asynchronous executions because they can not foresee all possible interleavings of events. In fact, most of these interleavings are irrelevant: the programmer does not care about the arbitrary orderings of events by physical time. Instead, the programmer is concerned with the *logical ordering of events*. At the primitive event level, this ordering is captured by Lamport's *happened before* relation. Other debuggers have used temporal logic to express assertions about *happened before* [6, 10] but they do not use behavioral abstraction. We extend the relation to abstract events, defining three relations: **precedes**, **parallels**, and **overlaps** [15]. Informally, if A and B are abstract 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 an event in A to an event in B and no dependency from an 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 been proposed [8, 16, 18] 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 in the programs we have examined.

Abstract events may interact in complex ways and often the programmer wishes to focus on particular aspects of that interaction, excluding all other aspects. Thus, for example, when debugging a program that used a parallel queue, we were surprised to find that the **overlaps** relation held between **Insert** and **Delete** operations on the same queue location. This interdependence was the result of logical orderings imposed by the mechanisms used to resolve competition for queue locations; these orderings had nothing to do with the correctness of the implementation. When we asked the debugger to ignore all orderings except those imposed by accesses to queue locations, we were able to correctly interpret the behavior of the program [7]. Ariadne, like our previous debugger, allows the user to selectively ignore some logical orderings and thus manipulate logical time to create different *perspectives* on program behavior [15].

2.3 Ariadne Queries

In the design of our language, we traded expressivity for simplicity, so we cannot always describe the intended behaviors precisely. To compensate, Ariadne provides a set of functions that return characteristics of a match. The user can explore a match with queries such as

> Did all matching 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?

At this time, Ariadne provides only rudimentary feedback. Our current work is aimed at expanding its repertoire. Although simple, Ariadne has proved effective in finding a variety of parallel program bugs. We have found it flexible in allowing the user to examine program behavior from different viewpoints. The next section describes several Ariadne debugging sessions illustrating its modeling language and functionality.

3 Sample Ariadne Debugging Sessions

In this section, we illustrate the power of our modeling language, the use of logical time, and our query facilities. We describe four sample debugging sessions using the Ariadne prototype.[7]

3.1 Permutation

Our first example involves a simple permutation of values, stored one per process: each process computes its receiving process, sends its value to the appropriate destination, and reads a new value. The program we debugged worked correctly on small processor arrays but failed on an array of 512 processes.

Summary of Debugging Session. In examining an execution history graph from the 512 process system, we first tried to match the expected pattern for a permutation (that is, a Write followed by a Read on each process). The pattern was not found and we were told that only processes with PIDs less than 256 completed their **W R** chain. In examining the behavior of the remaining processes, we discovered that all of the processes wrote their values to the lower 256 processes. This pointed us to an error in the address calculations.

[7] The Ariadne prototype is only partially implemented. The matching faciltities and many of the query functions have been implemented but the syntax we use here is not yet available in the prototype. In addition, several recent modifications to the language including the use of **WRT** clauses as filters and the shuffle operator on the chain level have not yet been implemented. The examples in this paper, with the exception of the triangulation session, have been run on the prototype although in a few cases, we had to hand compute values returned by query functions.

Debugging Session. We began by defining chain and p-chain patterns to be used in matching.

Chains are described by a slight extension to regular expressions. Our expressions, over primitive event types, use operations of concatenation (denoted by adjacency), alternation (+), shuffle (|), and Kleene Star (*).[8] For this example, we modeled the activity of a process with the pattern

W R

which describes a **Write** followed by a **Read** on a single process. For future reference, we also named the components of the pattern and the pattern itself. The complete chain definition was thus

? PermutationChain = send: W receive: R

where **send** and **receive** name the event matched by **W** and **R** respectively. The Ariadne prompt – **?** – is used in this paper to distinguish between lines of user input and debugger output. p-Chains are described by binding a chain to a set of processes. In this case, **PermutationChain** was bound by

? Permutation = PermutationChain ONALL PROCS

to the set of all processes (denoted **PROCS**). The keyword **ONALL** indicates that a copy of the chain must originate on each member of that set.

pt-Chains are used to describe the temporal composition of p-chains. In this example, we are looking for a single instance of **Permutation** and thus the specification is trivial. We ask for a match with

? PermuteEvent = match Permutation

If the match had succeeded, **PermuteEvent** would have been set to the resulting abstract event. In this case, however, it failed; **PermuteEvent** was set to $\perp$ and the following feedback was given

Match failed: Search failure.
Looking for Permutation.
Found 256 chains on {0..255}; using 256 processes.

This indicates that the search failed while looking for the p-chain **Permutation** and that during the search, 256 complete chains were recognized, one initiating on each of the processes numbered 0 through 255. The last part of the message gives the number of processes that had primitive events matched by the completed chains.

This information told us that no process above 255 did both a **Write** and a **Read**. To investigate further, we asked for additional information about the behavior of a specific process using the **matchchain** command. **Matchchain** attempts to recognize a single chain on a single process and provides feedback on the reason for a failure. In this case, it reported

? matchchain PermutationChain ON 256
Match failed: Chain failure.
Expecting an R but encountered the Right Cursor.

The **Right Cursor** marks the right end of the portion of the execution history graph that we are examining. Thus this response indicated that process 256 wrote a value but did not read one. Since this did not seem to help in locating the error, we tried another tactic.

We broke the pattern into two simple pieces, first matching all of the **Write**s and then matching all of the **Read**s. The dialogue went as follows:

? PWrite = W ONALL PROCS
? WriteEvents = match PWrite
Match succeeded.
Found 512 p-chains on {0..511}; using 512 processes.
? PRead = R ONALL PROCS
? ReadEvents = match PRead
Match failed: Search failure.
Looking for PRead.
Found 256 chains on {0..255}; using 256 processes.

[8] Note that the shuffle operator is expensive to implement. We included it to model the behavior of guarded input commands; thus in practice, we would expect that only a few items will get shuffled at a time. We allow its use only on the chain level.

This is more helpful. It indicates that all **Writes** occurred as expected but no process above 255 received a value. Where did the missing values go? We found out by asking

> **? destinations (WriteEvents)**

This is our first example of a function returning the characteristics of a match. Each such function takes an abstract event as its argument and recursively searches the structure of that event. In the case of the **destinations** function, the structure is searched for **WriteEvents** and the set of destinations for those events is returned. The feedback was

> **Values written to 256 processes: {0..255}.**

indicating that *all* of the values written by the 512 processes were directed at the lower 256 processes. Clearly there was some problem in the address calculation code. In fact, the variable holding the destination PID was mistakenly allocated to be an 8-bit quantity; larger values were truncated. Thus, only processes whose identifier was less than 256 could complete correctly.

If the same address truncation had occurred in a program for compression rather than permutation, it would have been harder to detect. A compression is very much like a permutation except that not all of the processes write or read values. During compression, the nonzero elements of an array (stored one value per process) are moved to the beginning of that array. Only processes with nonzero data values **Write** and only processes at the beginning of the array that are to receive a value **Read**. Thus, the chain definition used above, **W R**, would not work because it is a *process-centered* chain that follows the activity of a single process. Instead, we must use a *data-centered* pattern that follows the path of a communication. This is described by the pattern

> **? CompressionCh = send: W @ receive: R**

The @ moves the *context of the match* – that is it changes the process and the location for matching – to the receiving process. Thus this pattern matches a **Write** on the initiating process and then follows that communication edge to continue matching on the process that is its destination.

Not all of the processes execute the **CompressionCh** pattern (only those initially holding nonzero values), so we can not determine *a priori* the set of processes for binding. Instead we use

> **? Compression = CompressionCh ONSOME PROCS**

where the keyword **ONSOME** indicates that a successful match need only occur on a nonempty subset of the given process set. Now, however, when we do the match, it succeeds despite the presence of the truncation error:

> **? CompressionEvent = match Compression**
> **Match succeeded.**
> **Found 256 p-chains on {1,12,24..33,37,45,47,56,58..65,78,89..112,129..137, 139,141,143..149,156,158,160..189,196,197..234,241,245..258, 267,269,276, 280..298,301,314,324,356,358,367..391,413,415,433..456,470..494 };**
> **using 362 processes.**

Note that the process set is not consecutive because chains are only found on those processes that initially have nonzero values. This result looks correct; there is no indication that some values were written but never read. We could detect this sort of error only by asking if anything remained in the trace after the match. (Remember that a match removes the matching events from the execution history graph.) This can be done with

> **? Left = match REMAINDER**

where **REMAINDER** is a predefined pattern that matches anything. In our example, where the specific trace we were using had 290 processes with nonzero PIDs, the result was

> **Match succeeded.**
> **Found 34 p-chains on {15..18,79,115..126,190..194,335,495..498,501..507};**
> **using 34 processes.**

meaning that 34 "extra" events were found that had not been matched by the **Compression** chains. What were these events? We can find out with

? eventtypes (Left)
34 Writes
0 Multicasts
0 Reads
0 Phase Markers

where **eventtypes** is again a function that returns a match characteristic. In this case, it counts the number of primitive events of each type. The answer indicated that all of the unmatched events were **Writes**. We now are in the same position that we reached in the permutation example: we know that only processes below 256 completed and that some of the values that were written were never read. The **destinations** function would lead us to the error in the same manner as above.

3.2 Gaussian Elimination

Our next example comes from a parallel Gaussian elimination program. The program operated on an input matrix stored one row per process. In reducing that input to an upper triangular matrix, it executed a number of iterations (one for each process), each beginning with the broadcast of a pivot row. The program produced incorrect results when run on large systems of equations; we report here on an instance with 256 equations, running on a 256 processor system.

Summary of Debugging Session. In tracking down the bug, we first determined that all 256 broadcasts occurred and that each process performed exactly one broadcast. We then attempted to ascertain that each broadcast logically **preceded** the next, but this turned out to be untrue, leading us to the error: the programmer had omitted necessary barrier synchronizations between broadcasts.

Debugging Session. We defined a broadcast chain as

? BroadcastChain = senders: M @ receivers: R

In this case, because the send operation was a multicast rather than a single write, the read in the pattern matches *all* of the **Read**s associated with that **M**, enabling us to uniformly handle single writes, multicasts, and broadcasts.

The p-chain specification created a separate broadcast event for each process. Using a for-loop, we defined an array of **Broadcast** p-chains:

? for (i=0; i <= P-1; i++) do Broadcast[i] = BroadcastChain ONALL {i} od

where **P** is a defined constant giving the number of processes in the system. Each of these p-chains will match a process performing a single write that is read by all other processes.

To ascertain that the correct number of broadcasts were performed, we attempted to match a set of **P** broadcasts with

? BCseries = match (Broadcast[])^P

The missing index in **Broadcast**[] indicates that any element can be used, making it a shorthand for

(Broadcast[0] + Broadcast[1] + ... + Broadcast[255])

where + means alternation in our expressions. The match was successful, resulting in

Match succeeded.
Found 256 p-chains on {0..255}; using 256 processes.

Since **BCseries** matches a p-chain, it could potentially match all **P** occurrences on the same process, so we have to look at the output carefully. In this case, the **Broadcast** events occurred on 256 processes and thus, that each process must have initiated one. We then used

? owners (BCseries WRT {receivers})
256 owners: on processes {0..255}.

owners is again a function that determines the characteristics of a match. In this case, its argument is an abstract event that is modified by a **WRT** clause acting as a filter. **WRT {receivers}** makes only events of type **receivers** (as defined in the chain pattern) visible to the search; thus this query determines the set of processes executing **receives** in **BCseries**. It told us that every process executed a receive. We determined the total number of receives with

? count (BCseries WRT {receivers})

65,280 occurrences: on processes {0..255}.

which told us that every process read every broadcast ($255 \times 256 = 65,280$). We now knew that the correct number of events occurred and so we attempted to verify that the correct logical relation – **precedes** – had held between them.

The **Broadcast** events have already been removed from the trace because they were successfully matched above. We could check for the **precedes** relation in two ways. We could restore the **Broadcasts** to the trace and rematch them with

? restore (BCseries)

? OrderedBCseries = match < Broadcast[] * >

where the angular brackets indicate that the **precedes** must hold between matched events. Alternatively, we could use a predicate over abstract events[9]

? e_precedes (BCseries)

In either case the relation **precedes** is checked in the matched abstract event *using its search order*; the result is

Precedes failed.

Broadcast[38] *overlaps* **Broadcast[243].**

It tells us that the first 135 broadcasts occurred correctly but the 135th *overlapped* with the 136th meaning that **precedes** did not hold between them. This must mean that some process – either **38** or **243** – read the broadcast values out of order. This observation led us to the bug: a missing synchronization between broadcasts.

3.3 Delaunay Triangulation

This example demonstrates that Ariadne can model complex behaviors. The program is a parallel version of Bowyer's algorithm to construct a Delaunay Triangulation [4]. In Bowyer's algorithm, points are inserted into an existing mesh one at a time; in our version, they are inserted in parallel. Each point is managed by its own process which communicates with surrounding processes looking for triangles with circumcircles[10] that contain its point. The triangles located in this search are "locked" to prevent concurrent access by other insertions, and the polygonal region they form is modified to add the new point. To avoid deadlock, conflicting requests for locks on triangles are resolved by aborting one of the insertions.

Summary of Debugging Session When our program ran, it completed but the triangles it formed did not meet the Delaunay criteria. The sequences of insertions appeared to be correct. We hypothesized that despite the locking mechanism, some of the insertions interfered with each other. We checked this by looking for insertions that **overlapped**. We found three such pairs and, in examining the processes that executed those insertions, we determined that our locking mechanism essentially locked the sides of triangles but not their vertices.

Debugging Session In the current version of Ariadne, we do not have access to the contents of a message or its type. It is not practical to include the contents of all messages in every trace, but it is possible to use a replay mechanism [20, 23] to acquire additional trace information. We expect to include access to such information in future versions of our debugger. For purposes of this example, we achieved the same affect by modifying the program so it sends different message types on different, named channels. Thus, for example, the request for a lock is sent on port **req** and the response to that request is sent either on **ok** or on **no**. The debugger detects this use of ports. Within patterns, port names are appended to communication events with an underscore; matched instances of these events must have the correct port names.

[9] The prefix e on **precedes** indicates that this is a predicate over a single abstract event, other versions of this same predicate operate over sequences of events.

[10] The circumcircle of a triangle is the circle that can be drawn through all of the vertices of that triangle.

For expository reasons, we model only the part of the behavior relevant to the error, that is, we model only point insertions. These insertions begin with some number of attempts to lock relevant triangles:[11] the initiating process sends a multicast requesting the relevant locks on **req**, the recipients respond on either **ok** or **no**, and the initiating process collects the responses. A simple form of this pattern might be

M_req @ R (W_ok @ R + W_no @ R)

where the control of matching begins on the initiating process with the multicast and splits at the first `@` to proceed independently on each of the receiving processes.

This simple expression, however, is not sufficient because we must also model subsequent behavior on the initiating process. In the case of an unsuccessful attempt, the initiating process subsequently sends an "abort" message and then retries the lock attempt; in the case of a successful attempt, it subsequently attempts to get all relevant triangles to commit to the update. Thus, we must return the control of matching to the initiating process. We indicate this by marking the point of the split with the symbol `<@` (replacing the `@`) and the point of the return with the symbol `@>`. Thus a lock attempt is defined as

LockAttempt = M_req <@ R (W_ok @ R + W_no @ R) @>

Similarly, we define an abort, an unsuccessful attempt to commit, and a successful attempt to commit

Abort = M_abort <@ R @>
CommitNo = M_com <@ R W_committed @ R @>
CommitYes = M_com <@ R start:W_committed @ R @>

where the **CommitYes** includes a tag on events that essentially marks the beginning of the critical region for the insert. The initial, unsuccessful attempts are matched by

Unsuccessful = ((LockAttempt Abort) * LockAttempt CommitNo Abort) *

and the ultimately successful attempt by

Successful = (LockAttempt Abort) * LockAttempt CommitYes

Once the locking attempt succeeds, the initiating process performs the actual insertion of its point by sending a multicast on port **add** and waiting for acknowledgments on port **done**. The pattern is

Addition = M_add <@ R end:W_done @ R @>

where the tag **end** is used to mark the end of the critical region for this insert. The entire chain and the needed p-chains are as follows

? Insert = Unsuccessful Successful Addition
? for (i=0; i <= P-1; i++) do AddPoint[i] = Insert ONALL {i} od

We successfully matched the expected behavior with the command:

? Triangulation = match (AddPoint[]) *

This led us to conclude that all of the needed transactions had occurred. We hypothesized that there must have been some interference between insertions. To check this, we used the following query, looking just at the "successful" portion of the matched additions.

? e_non_overlaps(Triangulation WRT {start,end})
Assertion Failed.
AddPoint[17] overlaps AddPoint[13]
AddPoint[55] overlaps AddPoint[54]
AddPoint[100] overlaps AddPoint[98]

The feedback on the failure of this assertion led us to investigate the pairs of points that had overlapping insertions. We discovered that processes in each pair shared common triangle vertices. This led us to an error in our locking mechanism: in effect, we were locking the sides of the triangle but not their vertices.

Ariadne was designed as a testbed for investigating the utility and limitations of various types of match feedback. The above examples demonstrate successful uses of its current features. In the next section, we give an example of a program for which it was not successful.

[11] Processes recompute the set of relevant triangles immediately before each lock attempt but that behavior is not modeled here.

4 The Limitations of Textual Feedback

In this example, we consider a program that implements a dictionary search in which queries are pipelined from a host to a database of key-ordered records stored in a hypercube. Queries are routed within the cube to the proper node using binary search. More than one query is active at a time. The program as written contained a routing error.

We consider an 8 processor cube with processes having PIDs 0 through 7 and a host process with PID 8. We model the behavior of the program as a series of queries, each query starting at the host, traversing the cube and eventually returning to the host. The chain query uses two features we have not encountered thus far: the definition of a set of processes (**Cube**) and the limitation of a communication event to a set of processes (denoted by # followed by a process set).

? Cube = {0..7}
? QueryChain = W#{8} @ R#Cube (W#Cube @ R#Cube) * W @ R#{8}
? Query = QueryChain ON {8}
? match Query *
Match Succeeded.
Found 2 p-chains on {8}; using 8 processes.

The match succeeds but it does not give us any information about the error. Further investigations using Ariadne did not help. We had better success in debugging this program with our previous animating debugger, Belvedere [14].

In using Belvedere, we also defined an abstract event that matched the entire set of messages associated with a query; the query itself was much more complex (Belvedere uses the EDL modeling language [3]). Initially, the animation was incomprehensible, as shown in Figure 1*a* because the **Query** events overlapped in logical time: each query follows data-dependent paths through the cube, arriving in different orders at different processes. To separate the events, we created a perspective on the animation that included only dependencies caused by **Write** events on the host process (this is the same functionality provided by Ariadne's **WRT** clauses). Two snapshots from these perspective views are shown in Figure 1$b-c$. They portray the same execution trace that we used above with Ariadne. Now, however, the erroneous behavior is easy to spot: in Figure 1*c*, a query crosses a dimension of the cube twice.

As the programmers of this code, we knew that message transmissions should follow the path of a binary search. Once half of the remaining cube is eliminated by a comparison, the search should never go back to that subtree by crossing the same dimension of the cube again. Investigations of this behavior, led us to discover a routing error in the initial calculations of the return path for a query.

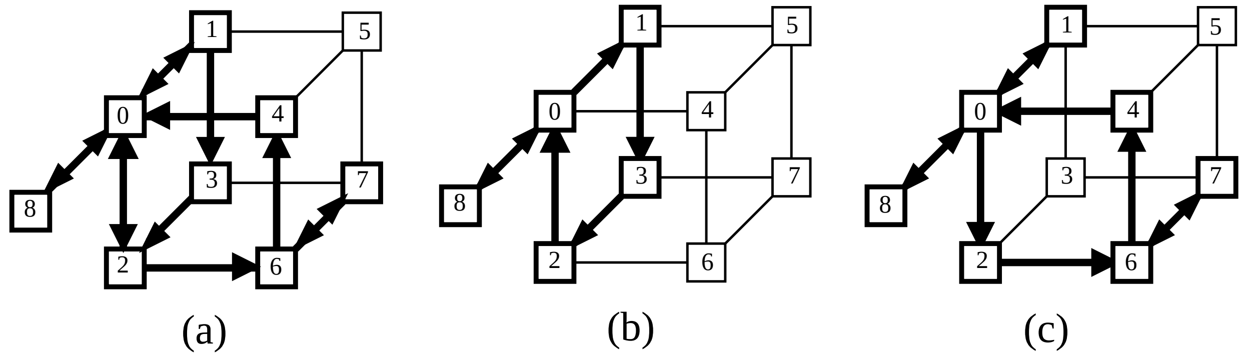

Figure 1: Snapshots from the an animation of the Dictionary Search. Concurrent abstract events (*a*); a perspective view of an abstract events showing the path taken by an individual request (*b*); and a perspective view of a second query showing an extra communication from the front to the back plane of the cube (*c*).

The routing error was immediately apparent from the animation but we could not find it with Ariadne.

It is not possible to concisely describe a query that finds this anomaly; worse, it is unlikely that the programmer would even think to ask such a query. The anomaly was detected as a deviation in a visual pattern. This example serves as an indicator that we will not be able to completely avoid graphical output. In an independent effort, we are developing scalable graphical representations of massively parallel computations and eventually, we expect to combine the two efforts.

5 Conclusion

We have introduced a new approach to the application of event-based abstraction to massively parallel computing. Previous methods were limited by their modeling languages: Sufficiently expressive languages required very complex matching algorithms that admitted only very limited feedback on the extent of a match. In some cases, the feedback was graphically presented in ways that did not scale to massively parallel systems. Our approach uses a simple modeling language that describes global patterns of communication in terms of parallel compositions of local patterns. This produces concise, scalable definitions and it allows for more informative feedback. We compensate for the loss of expressivity by allowing the user to interactively explore the extent to which a model matches the execution trace. We do not rely on graphical renderings and thus our techniques work well for even moderately large numbers of processes. We have implemented a prototype called Ariadne and have illustrated the effectiveness of this approach by presenting sample Ariadne debugging sessions involving actual parallel programs.

Ariadne was designed as a testbed for exploring the scalable application of event-based behavioral abstraction. We are currently evaluating the expressivity of its language and functional queries. In addition, because programmers are reluctant to learn new modeling languages for the sake of debugging, we are considering graphical languages that might make the description of patterns less onerous. We are also designing techniques for producing graphical displays of program behavior that would scale well. Finally, because Ariadne will eventually have to be intergrated into a more complete debugging system, we are investigating extensions to aspects of program behavior other than communication.

6 Acknowledgements

We thank a number of people for their contributions to this work. The Ariadne Development Team designed and implemented the prototype: Ruth Anderson, Sung-Eun Choi, Jeffrey Dean, Donald A. Lobo, Ton Anh Ngo, and W. Derrick Weathersby. Lee Delaney and Patrick Donohue tracked down some of its lingering bugs. Bruce Leban commented on earlier versions of the paper.

References

[1] G. Alverson, W. Griswold, D. Notkin and L. Snyder. A flexible communication abstraction for nonshared memory parallel computing. *Proceedings of Supercomputing '90*, 1990.

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

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

[4] A. Bowyer. Computing Dirichlet Tesselations. The Computer Journal, 24(2), pages 162–166, Feb. 1981.

[5] B. Bruegge and P. Hibbard. 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.

[6] 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.

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

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

[9] 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*, 24(1), pages 163–173, 1989.

[10] G. S. Goldszmidt, S. Katz, and S. Yemini. High level language for debugging concurrent programs. *ACM Transactions on Computer Systems*, 8(4), pages 311–336, Nov. 1990.

[11] 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.

[12] M. Heath and J. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, 1991.

[13] D. Hembold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2), pages 47-57, Mar. 1985.

[14] A. A. Hough. *Debugging Parallel Programs Using Abstract Visualizations.* PhD thesis, University of Massachusetts, Amherst, MA 01003, 1991. Also COINS Technical Report 91–53.

[15] 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.

[16] 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.

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

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

[19] 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.

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

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

[22] 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.

[23] B. Miller and J.-D. Choi. A mechanism for efficient debugging of parallel programs. *SIGPLAN Notices*, 24(1), pages 141–150, 1989.

[24] 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.