The Integration of Event- and State-Based Debugging in Ariadne *

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. Parallel programs often have complex behaviors that require multi-level debugging strategies. Here, we propose an integration of event- and state-based strategies that initially uses event-based behavioral modeling to narrow the focus of attention and then uses state-based techniques to relate observed errors to specific code segments. Our debugger supports this strategy, allowing the results of behavioral analysis to be used in specifying consistent, global breakpoints. These breakpoints differ from those provided by other debuggers in that they are meaningful in the context of the ongoing event-based analysis. In addition, the breakpoints that users select with our debugger would be difficult to set using existing state-based approaches. We demonstrate here that the combination of event- and state-based debugging is significantly more powerful than either strategy alone.

1 Introduction

The complex behavior of parallel programs often requires a multi-level debugging strategy. Such a strategy might, for example, employ event-based techniques [1, 3, 7, 9] 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 could be 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 could be used. This strategy effectively addresses many of the issues in parallel debugging. The initial use of event-based techniques focuses the user's attention on manageable portions of the state space and (as we will see below) provides the basis for establishing consistent, meaningful, global breakpoints. Event-based techniques can incorporate replay mechanisms that support reproducible execution [13] and logical time transformations to filter out perturbations due to asynchrony [8, 12]. State-based techniques, on the other hand, allow the user to directly examine an execution to an arbitrary level of detail and often make it easier to relate errors to source code constructs.

A number of event- and state-based debuggers already exist. Here we discuss the integration of two – Ariadne [2] and *ipd* [10] – into a single environment. The key to this integration is the use of user-defined abstract events in establishing consistent, global breakpoints: the states we provide are *meaningful in the context of the user's ongoing event-based analysis.* Often these breakpoints would often be difficult, if not impossible, to set using existing state-based techniques.

In Section 2, we give a brief overview of the integrated environment; in Section 3, we demonstrate its use.

2 Setting Breakpoints with Abstract Events

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 [17] but this is overly constraining in multithreaded language environments. Alternatively, local breakpoints can be extended dynamically to global breakpoints [6, 16, 18]. Unfortunately, debuggers that support this typically allow processes not directly involved in the breakpoint to continue executing for some time, potentially obliterating information relevant to the source of the error. We choose instead to take advantage of replay mechanisms: we use the results of *post mortem* modeling to insert directives that will, during replay, halt each process at the latest point at which it could have influenced the event(s) triggering the breakpoint.

Manabe and Imase [15] also use this approach, allowing breakpoints to be to specified as conjunctions and disjunctions of local predicates. 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.

A matched (or partially matched) abstract event can be used to partition the processes of a computation into three equivalence classes: the *core set* containing the processes that executed parts of the abstract event; the *influence set* containing the processes that could have causally affected a component of the abstract event; and the *other set* containing the remaining processes. 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 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 [4]. Whenever a user's breakpointing request would result in an inconsistent state, our debugger automatically constructs an earlier, consistent state using a greedy algorithm.

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 [5]. 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 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. Ariadne uses a simple modeling language based on regular expressions.

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;
```

where `W` and `R` indicate write and read operations respectively and `ch` is a keyword indicating that this is the definition of a local pattern of interaction. The behavior of an activated node was modeled as

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

where the `@` symbol moves the pattern matching to the process that is the destination of the immediately preceding write. A complete description of the modeling language is beyond the scope of this paper but can be found in other references [2].

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 1 for a trace that contained a single originating node sending 5 different queries. As can be seen, Ariadne imposes a tree-hierarchy on the matched events that is based on the user-defined model. Internal nodes represent user-defined abstract events and leaves (not shown in the figure) represent primitive events. This tree is called a *match tree*. For scalability, Ariadne compresses it. Here compression is shown by the stacked boxes below the `query` level (indicating that the subtrees are not shown) and the slider between the two `query` nodes (indicating that 3 of their siblings are not shown). Again, the complete description of this figure is beyond the scope of this paper [11].

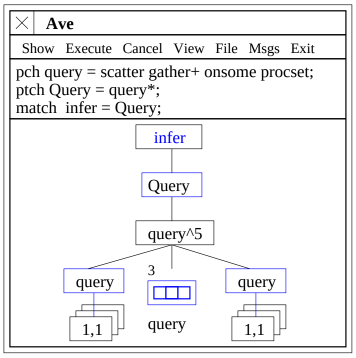

Figure 1: Ave feedback showing coarse grain pattern match in knowledge-base example.

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 which provides a spreadsheet-like interface for computing incremental attributes of a match tree [11]. With a few simple queries, we were able to ascertain the following:

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.)

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.)

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

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?

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.

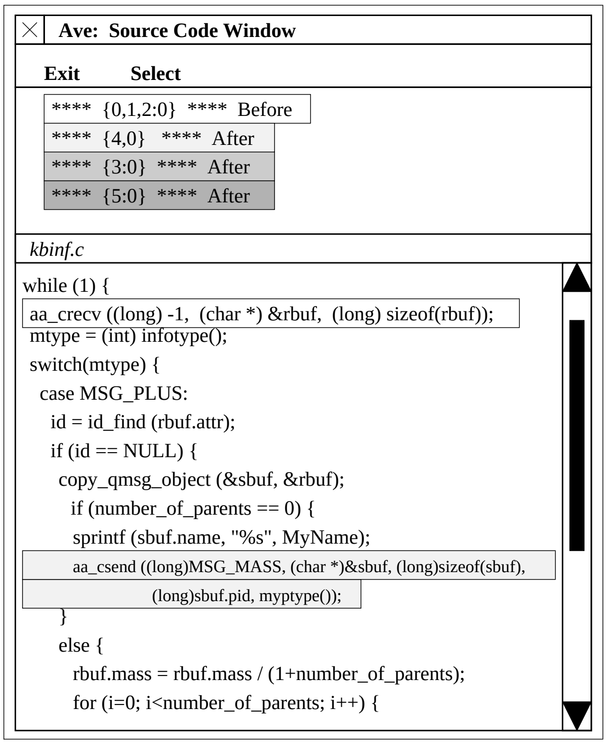

Figure 2: Ave: Source Code Window, showing processes stopped at different points.

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 2. 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.

4 Conclusion

In this paper, we have presented a multi-level debugging strategy which uses event-based behavioral debugging at the highest levels of abstraction and then switches to stated-based mechanisms as the focus of attention narrows. This strategy effectively addresses many of the issues in parallel debugging. The initial use of event-based techniques focuses the user's attention on manageable portions of the state space; it can provide the basis for establishing consistent, global breakpoints using replay mechanisms; and it can support logical time transformations that filter out perturbations due to asynchrony. State-based techniques, on the other hand, allow the user to directly examine an execution to an arbitrary level of detail and often make it easier to relate errors to source code constructs.

Here, we discussed the integration of two existing debuggers, Ariadne and *ipd*. The key to their integration was the use of user-defined abstract events in establishing *consistent, global breakpoints that were meaningful in terms of the user's ongoing event-based analysis of his/her code.* These breakpoints would be difficult or impossible to set with standard breakpoint debuggers.

We are currently extending the implementation to a number of different platforms, focusing on an integration of Ariadne and the Breezy breakpoint debugger [17].

Acknowledgements

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

References

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

[2] 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.

[3] 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.

[4] A. P. Goldberg, A. Gopal, A, Lowry, and R. Strom. Restoring consistent global states of distributed computations. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 144–154, 1991.

[5] M. Greenberg and J. Cuny. Parallelism in knowledge-based systems with inheritance. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 141-145, 1988.

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

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

[8] 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, 1990.

[9] 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, 1990.

[10] Intel Supercomputer Systems Division. Paragon Interactive Parallel Debugger Reference Manual. Paragon Documentation, 1994.

[11] J. Kundu and J. E. Cuny. A scalable, visual interface for debugging with event-based behavioral abstraction. In *Proceedings of New Frontiers on Massively Parallel Processing*, pages 472–479, 1995.

[12] 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.

[13] 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, 1987.

[14] 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.

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

[16] K. Mani Chandy, and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems,* Vol 3, No. 1, pages 63-74, 1985.

[17] D. Brown, S. Hackstadt, A. Malony, B. Mohr. Program Analysis Environments for Parallel Language Systems: The TAU Environment. *Proceedings of the 2nd Workshop on Environments and Tools For Parallel Scientific Computing*, pages 162-171, 1994.

[18] B. Miller, and J. -D. Choi. Breakpoints and halting in distributed programs. *Proceedings of 8th International Conference on Distributed Computing Systems*, pages 316-323, 1988.