Traveler: The Apiary Observatory

Carl R. Manning

Message Passing Semantics Group
MIT Artificial Intelligence Laboratory
Cambridge, Massachusetts USA

Abstract

Observing and debugging concurrent actor programs on a distributed architecture such as the Apiary poses new problems not found in sequential systems. Since events are only partially ordered, the chronological order of events no longer corresponds to their causal ordering, so the execution trace of a computation must be more structured than a simple stream. Many events may execute concurrently, so a stepper must give the programmer control over the order in which events are stepped. Because of the arrival order nondeterminism of the actor model, different actors may have different views on the ordering of events. We conquer these problems by recording the *activation ordering*, the *transaction pairing*, and the *arrival ordering* of messages in the Apiary and displaying the resulting structures in Traveler's window oriented interface under user control.

1. Introduction

Observing and debugging concurrent actor programs on a distributed architecture such as the Apiary poses new problems not found in sequential systems. Since events are only partially ordered, the chronological order of events no longer corresponds to their causal ordering, so the execution trace of a computation must be more structured than a simple stream. Many events may execute concurrently, so a stepper must give the programmer control over the order in which events are stepped. Because of the arrival order nondeterminism of the actor model, different actors may have different views on the ordering of events. We conquer these problems by recording the *activation ordering*, the *transaction pairing*, and the *arrival ordering* of messages in the Apiary and displaying the resulting structures in Traveler's window oriented interface under user control, either after the fact in case of a trace, or while the structure is incrementally constructed in a stepping session.

Actor programs are concurrent at a fine grain, the level of message passing. Each actor can process its messages concurrently with other actors using only its local state; thus concurrency is engendered whenever an actor sends more than one other actor a message. The Apiary architecture takes advantage of this property by migrating actors between the different processors of the architecture to balance the work load across the processors. Each processor in the Apiary keeps a queue of messages which need to be processed by the actors resident in it. The conceptual cycle of the worker is to take a message off the queue, deliver it to an actor which processes the message, and send any new messages to their target actors. Sending the new messages involves either sending a message over the network to another processor if the target of the message is resident there, or just enqueueing it locally if the target actor is local. A debugging system for the Apiary must deal with the fine level of concurrency and the distributed nature of the machine.

Actor programs are currently written in a core actor language called *Acore*. Acore syntax is much like that of Lisp, but expressions and commands in the same context (e.g. a command body or an argument list) may be performed concurrently. Acore forms are interpreted in a message passing style, so for example

```
(1 :+ 2)
```

sends the actor '`1`' the message '`:+ 2`'. Symbols in the keyword package (starting with '`:`') are interpreted as the selector of the message. Function calls are also interpreted as a short hand for message passing with the assumed selector '`:do`', so for example

`(list 1 2)` and **`(list :do 1 2)`**

mean the same thing: the target actor '`list`' is sent the message '`:do 1 2`'. This syntax is used throughout this paper.

2. Observing Transactions

Traditionally, tracing a program involves setting up an output stream (say, the user's terminal) and printing a description of an event on the stream as it occurs. This is feasible because the sequential nature of the program means there is a single total ordering on events, and this ordering corresponds well to the causal ordering of events. In this ordering, any functions called by a procedure are called one at a time in sequence, and all the subroutines called by the function are completed before moving on to the next function. Thus each function call is completed before the next begins, and the source of each subroutine call is apparent. For example, see figure 2-1.

```
Procedure1 a b c
  Function1 a
    subroutine1 a
    <-- a'
    subroutine2 a'
    <-- a''
  <-- a''
  Function2 b
    subroutine3 b
    <-- b'
  <-- b'
  Function3 a'' b' c
  <-- (a'' b' c)
<-- (a'' b' c)
```

Figure 2-1: Traditional Trace

Since everything is sequential, showing events in chronological order produces a stream where causality is apparent; there is no question who called `subroutine3` or where the parameter `b` came from.

However, tracing a concurrent actor program cannot proceed so simply because messages sent to different actors may be processed concurrently. If we were to naively output the events to a stream as they happen, then the stream would only reflect the approximate chronological order of the events rather than the causal order. Since the events do not necessarily have a total ordering, their order may change from trace to trace. In a distributed architecture, not only will the order of concurrent events be nondeterministic, but the order in which descriptions of events arrive at the output stream will also be nondeterministic. Thus we sought a method of recording the causal order of events which could be implemented in a distributed fashion.

Given an object oriented distributed computing environment, the obvious thing to do was to build an object oriented distributed recording mechanism. A message and its target comprises a task; each task execution is a message reception and is considered an event. For every event, a task record is created which records the target and the contents of the message, the task record of the event which activated it (the activating event), and the task records of the events which it activates (the activated events). Thus the *causal* or *activation ordering* of a transaction is recorded in this doubly linked graph of task records. See figure 2-2.

In order to produce this graph while the program is running, each task in a transaction being recorded carries an extra parameter which holds information about recording. The Apiary emulator traps on the presence of information in this parameter and performs some special processing to record the task. If the parameter is empty, then the emulator does not trap, makes no recording, and runs at full speed.

A reference to the activating task record is passed using the recording parameter of the task. After a task is

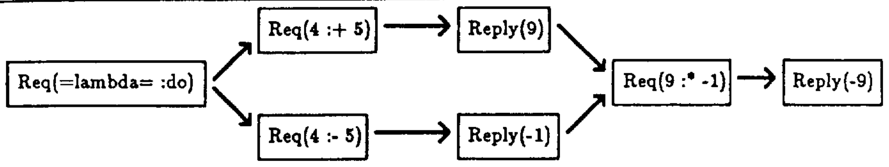


Figure 2-2: Graph of task records

This is the graph of task records for the computation performed by the expression

```
((lambda () ((4 :+ 5) :* (4 :- 5)))
```

`:Do` is the selector used for function calls.

dequeued, but just before it is delivered to the target actor, a task record for the task is created. The passed task record is the activating task record for the new task record. A message is sent to the activating task record to link it with the new task record. Finally the message is delivered to the actor with the new task record in the recording parameter; this will be passed onto any new tasks the actor creates.[1]

Now that the causal ordering of the tasks in a transaction is recorded, the question arises about how to display it. Since the traditional trace displayed only a single sequence of events and we wanted to display a partially ordered graph of events, it seemed logical to try to find a graphical way of using the screen and somehow gain another dimension to the display. A *Tree display*, which showed the graph as a tree of event nodes connected by lines representing the causal links, was tried; a simple display looked somewhat like figure 2-2 rotated 90° clockwise (except that joins were not displayed; see footnote). An *Actor Lifelines display*, which looked something like the event diagrams in [Clinger 81] and [Agha 86] rotated 90° counterclockwise, was also tried. These graphs were fine for showing the parallelism and structure in the computations, but they failed to be useful debugging tools because they obscured too much information about what the computation was doing and why. With all the nodes and lines, there wasn't much room to display much information about what actors were receiving messages, or what the content of the messages was. Another major problem was that these displays did not correspond to the abstractions made by the programmer in the program. The programmer thinks in terms of function and method calls and the results returned by these; in the event graphs there was little to link the requesting event with the corresponding reply, and the display wasn't suited to displaying it. In light of this experience, a display of the computation in terms of nested transactions, much like a traditional sequential trace, seems to be the answer.

Thus one goal was to create a display which looked something like a nested trace. For example, for the computation of figure 2-2, it might look something like figure 2-3.

There are several things to note about this format. The pairing of requests and replies makes it apparent which response corresponds to each request. The nesting of subtransactions preserves some of the causal information, but it does not show which of the subtransactions were performed concurrently and which must have been performed sequentially. However, we have found that the clues for debugging are generally found in the text of the descriptions (what is *that* actor doing here?, why did *that* function get called?) and in the nesting of transactions (what method was this called from?); causal relations between transactions within a method are much less important and are easily deduced from the source code (or they can be annotated on an individual task basis

[1]This method does not quite produce the graph shown in figure 2-2; in particular, of the tasks forming a join (e.g. the replies from the sum and difference), only the last to arrive will be linked as causing the continuation to continue (e.g. start the multiplication). The reason for this is that the replies are sent to a joining continuation actor, but the continuation actor just remembers the values previous to the last without activating any more tasks. Improving on this would require knowing how the joining continuation actor decided when to continue, an added complication we have been able to do well without so far.

```
Request(<lambda> :Do)
  Request(4 :- 5)
  Reply(-1)
  Request(4 :+ 5)
  Reply(9)
  Request(9 :* -1)
  [Tail Recursion]
Reply(-9)
```

Figure 2-3: Design of the Transaction Display

A possible transaction display for the computation `((lambda () ((4 :+ 5) :* (4 :- 5))))`.

from the Traveler's display).

This display seems to work well, but more information is needed before it can be generated from the recorded graph of the transaction. The pairing of requests with the corresponding response is crucial to the display, but the graph does not directly contain information linking pairs of requests with responses. A first attempt is to just indent a step for each request and outdent for a reply, but this approach doesn't work if there are tail recursive replies which close off more than one request.

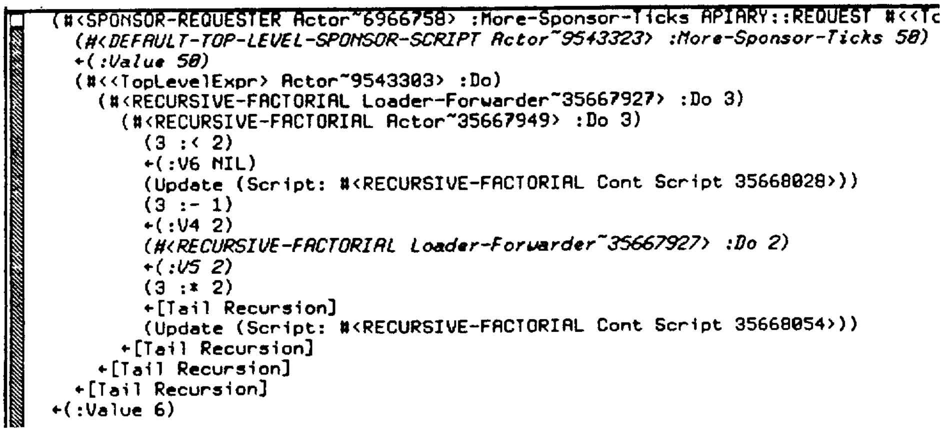
Each request contains a *customer* to which the reply for the request should be sent. One way to make the pairing is to start with the request and search the activated graph for the reply which is sent to the customer of that request. However, such a search would consume large computational resources, especially in a large graph with many requests and replies, so a more efficient approach was needed. Thus we use *transaction customers* to put in a bit more effort to store the pairing information while recording.

The role of a transaction customer is to intercept the reply before it is sent to the real customer, record the information about the reply in a reply task record, link the reply task record with the request task record, and forward the reply to the real customer. Therefore, at the time a request task record is made (just before the request is delivered to the actor), a transaction customer is created with references to the request task record and the real customer, and substituted for the real customer in the request task. Thus the transaction customer masquerades as the real customer, and when it receives the reply, performs its duties and forwards the reply to the real customer.

With the transaction pairing links added by the transaction customers, the recording mechanism for transactions is complete, and generating a trace display is not difficult. A transaction display for Traveler is shown in figure 2-4.

The implementation of the transaction display for Traveler includes the ability to selectively open and close transactions to expose or hide subtransactions, giving the user control over the level of detail shown, much like outline processing in some modern word processors. Closed transactions (those whose subtransactions are hidden) are displayed in italics. Traveler also provides the ability to save a snapshot of the entire display in an editor buffer, so if the display is still too large to conveniently view on the screen even when irrelevant transactions are closed, it may be printed out.

The tracing facility provided by Traveler so far has proven to be very useful in localizing problems in small applications. Even when the program hangs, a trace of an incomplete transaction can be produced to find the reason for hanging. The ability to selectively open and inspect subtransactions has proven to be a great help in dealing with the complexity of small and medium small applications, but for larger applications it has become apparent that a form of selective tracing (e.g. displaying only messages to certain actors) is also needed to cut down even further on excess information.



```
(#<SPONSOR-REQUESTER Actor~6966758> :More-Sponsor-Ticks APIARY::REQUEST #<<Tc
  (#<DEFAULT-TOP-LEVEL-SPONSOR-SCRIPT Actor~9543323> :More-Sponsor-Ticks 50)
  ←(:Value 50)
  (#<<TopLevelExpr> Actor~9543303> :Do)
    (#<RECURSIVE-FACTORIAL Loader-Forwarder~35667927> :Do 3)
      (#<RECURSIVE-FACTORIAL Actor~35667949> :Do 3)
        (3 :< 2)
        ←(:V6 NIL)
        (Update (Script: #<RECURSIVE-FACTORIAL Cont Script 35668028>))
        (3 :- 1)
        ←(:V4 2)
        (#<RECURSIVE-FACTORIAL Loader-Forwarder~35667927> :Do 2)
        ←(:V5 2)
        (3 :* 2)
        ←[Tail Recursion]
        (Update (Script: #<RECURSIVE-FACTORIAL Cont Script 35668054>))
      ←[Tail Recursion]
    ←[Tail Recursion]
  ←[Tail Recursion]
←(:Value 6)
```

Figure 2-4: Transaction Snapshot

A snapshot of a Traveler screen showing the recursive factorial of 3. The transactions in italics have their subtransactions hidden.

3. Stepping Tasks and Transactions

Stepping in a sequential programming system is traditionally done either by modifying the compiled code of the program to be stepped to introduce breakpoints between statements, or by using an interpreted version of the program so it can be stepped with an interpreter. Both of these pose problems in a concurrent system where the program may be shared – modifying the program to introduce breakpoints or replacing the compiled version with the interpreted version also affects anything else which calls the program concurrently. Although an interpreted stepper can afford a source code oriented view of the stepping process, it cannot step compiled code. Yet it may be important to see how a compiled system actor interacts with other actors, so it would be nice if there was a way to step compiled programs without modifying them.

Actor programs on the Apiary already have a set of built-in breakpoints: each time a message is sent, the message may potentially need to be transmitted to another processor. Therefore there is a break each place a message is sent, and it becomes natural to step by messages. The acceptance of a message by an actor (the processing of a task) is an event; as the result of processing the message, new messages may be sent. Since message passing occurs at the level of function and method calls, message events have proven to be a small enough step size.

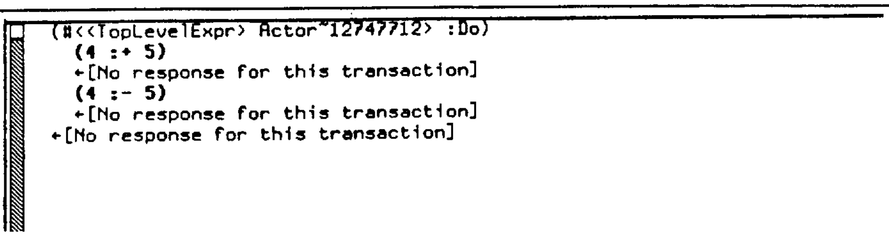
In order to step by event without modifying compiled code, we again make use of the recording parameter of tasks and the trap which checks it. To step a single task, an actor called a *collector* is placed in the recording parameter; the idea is that all new tasks created when the stepped task is performed will be sent to the collector instead of being delivered to their targets. When the recording trap finds a collector in the recording parameter, it sets a flag in the parameter before running the task. The recording parameter is passed on to all new tasks created by the actor; just before these tasks are delivered, they will also trigger the recording trap. This time, since the flag is set, the recording trap sends the tasks to the collector instead of delivering the message to the target actor.

Thus, by using collectors, we can build a stepper which starts with a task, steps it by sending it with a collector, and then displays the new tasks returned to the collector. When another unstepped task is stepped, the cycle is repeated.

Concurrency in the program means that many tasks will be available for stepping simultaneously. If several of these concurrent transactions interact with the same history sensitive actors, it may be important to test different orders of execution. To control the order tasks are performed, the user must be able to select which task to step next from the pool of unstepped tasks.

Stepping concurrent programs also introduces a problem of presentation – how can the pool of unstepped tasks be displayed so that their relation to the operation of the program isn't lost? A traditional stepper displays the stepping process as an incomplete trace in progress. With a little effort we can take advantage of a dynamic window interface to display stepping concurrent programs as a concurrent trace in progress. The display dynamically expands as more activated transactions are filled in between the outer transactions.

In Traveler the same display is used displaying a trace as displaying the stepping progress, so once a transaction is completely stepped, the display which results is the same as if the program had simply been traced. See figure 3-1. Since the same display is used, the facilities for opening and closing transactions are still available to control the display.



```
(#<<TopLevelExpr> Actor^12747712> :Do)
  (4 :+ 5)
  ←[No response for this transaction]
  (4 :- 5)
  ←[No response for this transaction]
←[No response for this transaction]
```

Figure 3-1: Snapshot of the Stepper Display

This is a snapshot of the stepper in the middle of computing the value of

((lambda () ((4 :+ 5) :* (4 :- 5))))

Unstepped tasks are shown in boldface and may be stepped by selecting them with the mouse.

Traveler also provides stepping by transaction, but with a new twist. Steppers usually allow you to step through a procedure call in one step, so you don't have to waste time stepping through procedures which you think are working fine. Traveler provides this capability by allowing you to step through a transaction in one step, so you can step from a request to its corresponding reply without stepping through the subtransactions needed to compute the reply. The difference is that the transaction is recorded, so that if the reply does not turn out as expected, you can open the transaction and examine its trace to find out what happened.

Displaying the stepped tasks as a trace and stepping by transaction both pose problems in implementation: how do you catch the response for a transaction and display it correctly in the trace? To solve this problem we use the same technique we used in building the trace, the transaction customer. When we step a request, we insert a transaction customer to take care of the reply. The transaction customer keeps track of the correct location in the screen's transaction structure for the reply, so when it receives the reply it is displayed in the correct place ready for stepping.

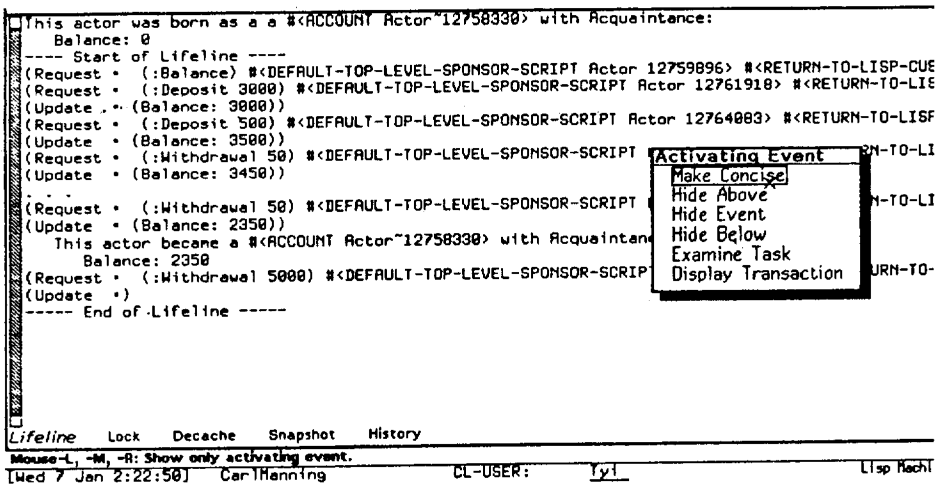
This stepping facility has proven to be very useful for small and medium small applications, but as we move into larger applications, we have found that there is too much careful but tedious stepping just to get to the problematic parts of the program. A breakpoint mechanism is needed so that we can start stepping in the middle of a program.

4. Viewing Lifelines

When there are history sensitive objects in a concurrent computing system, the ordering of events at an object becomes important to observe. We noted above that it is important to be able to control the ordering of concurrent transactions which interact with a common, history sensitive actor; when examining a trace, it sometimes becomes important to know how things happened from a particular actor's viewpoint. For example, there may be many concurrent transactions dealing with the same bank account, and we think we've constrained the ordering of the transactions so that no withdrawals will bounce, but if we find that a transaction occasionally does bounce it is useful to find out in what order the transactions really did arrive at the bank account so we can figure out what went wrong. We record this *arrival order* of recorded tasks at an actor in the actor's *biography*; the display of this biography is called a *lifeline*.

Recording the biography is fairly straightforward; each time we create a task record, we add it to the corresponding target actor's biography. The only subtle point is that we store a copy of the target actor's state in the task record rather than just a reference to the target actor; thus when we look back at the biography we can see a history of states as well as a history of tasks, and better understand why the actor behaved as it did as it evolved into its current state.

Displaying the biography is also straightforward; we display the list of tasks in the order the arrived. Traveler provides a few conveniences: users may change the level of detail with which a task is displayed, or they may hide the events between disparate events in the lifeline to juxtapose them for better comparison. See figure 4-1 for an example lifeline.



```
This actor was born as a a #<ACCOUNT Actor~12758330> with Acquaintance:
   Balance: 0
---- Start of Lifeline ----
(Request •   (:Balance) #<DEFAULT-TOP-LEVEL-SPONSOR-SCRIPT Actor 12759896> #<RETURN-TO-LISP-CUE
(Request •   (:Deposit 3000) #<DEFAULT-TOP-LEVEL-SPONSOR-SCRIPT Actor 12761918> #<RETURN-TO-LIE
(Update ,•· (Balance: 3000))
(Request •   (:Deposit 500) #<DEFAULT-TOP-LEVEL-SPONSOR-SCRIPT Actor 12764083> #<RETURN-TO-LISF
(Update  • (Balance: 3500))
(Request •   (:Withdrawal 50) #<DEFAULT-TOP-LEVEL-SPONSOR-SCRIPT                    N-TO-LI
(Update  • (Balance: 3450))
. . .
(Request •   (:Withdrawal 50) #<DEFAULT-TOP-LEVEL-SPONSOR-SCRIPT                    N-TO-LI
(Update  • (Balance: 2350))
   This actor became a #<ACCOUNT Actor~12758330> with Acquaintan
      Balance: 2350
(Request •   (:Withdrawal 5000) #<DEFAULT-TOP-LEVEL-SPONSOR-SCRIP                   URN-TO-
(Update  •)
----- End of Lifeline -----

Lifeline   Lock   Decache   Snapshot   History
Mouse-L, -M, -R: Show only activating event.
[Wed 7 Jan 2:22:50]   CarlManning          CL-USER:      Tyi                          Lisp Machl
```

Figure 4-1: Lifeline Snapshot

This is the lifeline of a bank account; the ellipses indicate portions of the lifeline are hidden to juxtapose different parts of the lifeline.

In our experience so far, lifelines haven't been used as often as transaction traces or even stepping, but they were very helpful when they were needed. The small programs we've tried so far are probably not complex enough for arrival ordering to get too far out of hand, and I expect lifelines may come into more use as more complex programs are tried. However, lifelines are useful for studying actors which arise as history sensitive managers as programs become more complex. The current format of displaying only the arriving tasks in the lifeline has also been a limitation on their use; it would be a better idea to present a lifeline of transactions where possible so that not only can you find what messages arrived, but you can study what the actor did with each message.

5. Summary and Future Work

Traveler is an integrated set of tools for examining actors and actor computations built on a window interface. In addition to the transaction and lifeline displays mentioned here, the current state of an actor may be examined simply by mousing any actor in the display. Traveler's panes can be reconfigured in several sizes to fit the task at hand, and each of the work panes can hold any transaction, lifeline, or examination.

All communication with actors to determine their state and for printing them is done through message passing. However, since debugging tools need to work even when there are problems with locked serialized actors (especially if the locking is the problem), communication with the user's actors is done with special `system-requests` which bypass the normal locks and handlers. Thus the debugging system bypasses fragile and possibly broken user code while still taking advantage of the distributed message passing nature of the system while it is investigating.

Of course, all this recording does come at a price. While recording, what is normally just two events, a request and a reply, now expands into at least 4 additional events to link the records. Three actors are created: the request record, the reply record, and the transaction customer. Because of this overhead, in our current emulation system programs can take from 5 to 20 times as long to run. Most of this overhead is due to the paging that goes on because of the high memory allocation rate.

Traveler has proven to be a very useful tool for observing and debugging concurrent actor programs on the Apiary. The strategy of recording a concurrent transaction and then examining the record is successful: transaction oriented tracing and stepping have been a huge leap forward over the chronological tracing and stepping facilities which existed before it, allowing us to observe and debug more complex programs. As we develop even larger programs, we are finding a few improvements and additions can be made; in particular, selective tracing and breakpoints are needed.

For the future, we plan to make the improvements I've suggested, as well as work on other tools. In particular, a source-oriented stepper (based on earlier work of Henry Lieberman) is under development. This stepper will display source code and allow concurrent expressions and commands within the code to be stepped by selection; when a value is produced, it replaces the expression which produced it in the code. An interpreter for our core actor language, Acore, is also under development to support this stepper.

Acknowledgements

Many thanks to the members of the Message Passing Semantics Group, who continue to serve as a user community, providing ideas and feedback on Traveler, and especially to Chun Ka Mui, who did much of the early work on Apiary debugging tools. This work was supported by a grant from the System Development Foundation.

References

[Agha 86] G. Agha.
Actors: A Model of Concurrent Computation in Distributed Systems.
MIT Press, Cambridge, Mass., 1986.

[Clinger 81] W. D. Clinger.
Foundations of Actor Semantics.
AI-TR- 633, MIT Artificial Intelligence Laboratory, May, 1981.