Single Stepping in Event-Visualization Tools

Marc Khouzam and Thomas Kunz
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1

Abstract

Event visualization tools are commonly used to facilitate the debugging of distributed applications. Although these tools provide a graphical view of distributed executions, they are frequently insufficient for full debugging purposes. The need for traditional debugging operations is often overlooked when building these tools. One of the most useful operations in traditional debuggers is single stepping. However, the difficulties faced when dealing with partially ordered-executions have deterred the development of a single-stepping method for concurrent processes. This paper describes a single-stepping operations suitable for event-visualization tools. Three different methods for single stepping are presented: *global-step*, *step-over* and *step-in*. Abstraction techniques are commonly used to reduce the overwhelming amount of detail presented to the user. Accordingly, single stepping in the presence of abstraction is considered. The operations defined in this paper have been implemented in Poet, a Partial Order Event Tracer, and examples of the resulting visualizations are given.

1 Introduction

To take advantage of available computer resources in the most efficient fashion, one must look to distributed applications instead of traditional sequential ones. However, distributed applications involve all the complexity of sequential ones, plus problems specific to distributed environments. Hence, tools that help understand the behavior of such applications are becoming more and more common. A useful method for describing and reasoning about a distributed execution is the use of event-visualization tools, which give the user a graphical view of the execution [5, 8, 12, 15, 18].

These tools aim to help the user understand and debug a distributed application. Although visualization is a very useful method for such a task, it is often not sufficient for full debugging purposes. Traditional debugging operations are also needed in such tools, so a user can study a faulty program more efficiently. One of the most frequently used functions of a traditional debugger is single stepping. It allows the user to study the program in an incremental manner, while examining its behavior. Such a facility would be a valuable addition to any event-visualization tool. However, defining the meaning of a *single step* within a non-sequential execution, where events are only partially ordered, is a non-trivial task. This paper describes a well-defined method for single stepping in such a tool, which allows the user to better understand the behavior of the execution. The concepts explained here are currently implemented in Poet, a Partial Order Event Tracer, developed at the University of Waterloo [12, 13].

An example of Poet during the visualization of a PVM (Parallel Virtual Machine) [4] application is given in Figure 1.[1] Within Poet, at the simplest level, each entity exhibiting sequential behavior is represented by a horizontal trace line. These entities can be processes, tasks, threads, objects, semaphores, and so on. In the remainder of this paper, for the sake of

[1] Poet can be used to visualize the execution of distributed applications in a number of target environments, among them OSF DCE, PVM, μC++, and ABC++. The details of this target-system independence are described in [14].

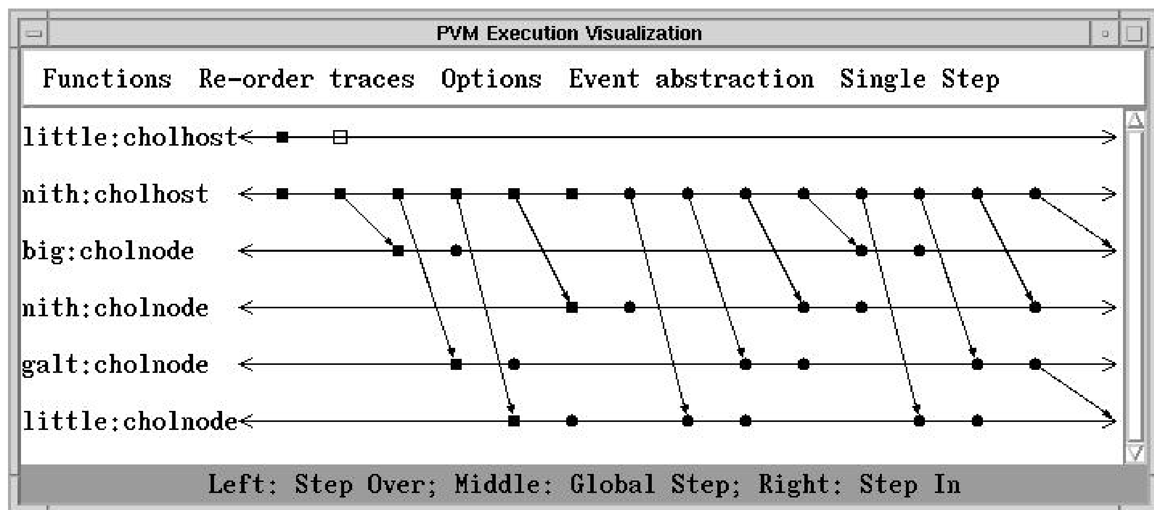

Figure 1: Poet, the Partial Order Event Tracer.

simplicity, the word *process* will be used to refer to any one of these entities. On each trace line, *events* are positioned in their order of occurrence to describe the behavior of a process over time. An event represents some computation and is considered to occur instantaneously. *Primitive events* constitute the lowest level of observable behavior in a distributed execution, such as the sending or receiving of a message, or the creation or termination of a process. On the display, pairwise related events are joined by arrows, such as sending a message from one process to another. At a higher level of complexity, *process clustering* [13] and *event abstraction* [10] are used to simplify the view of the execution. We will discuss these concepts further in Section 6.

A fundamental difference between traditional source-code debuggers and event-visualization tools is that, unlike code statements, the events of an execution are not known *a priori*. Because of this limitation, the part of our research discussed in this paper deals with *post mortem* single stepping, in which the events have been recorded during normal execution. The event traces can then be visualized by the user to understand the execution after the fact or serve as a reference point to deterministically *replay* the distributed execution [16]. In the former case, a user is limited to understanding the execution behaviour by going through it incrementally, at a controlled pace. In the latter case, traditional source-code debuggers can be attached to the individual processes, collecting detailed information about each process during execution, such as variable content, code path taken, etc. [17]

The paper is organized as follows. Section 2 reviews related work. Section 3 gives a short introduction to a few basic theoretical concepts, in particular the definition of the *past* of an event. This concept is fundamental for the remaining sections. Section 4 is the core of the paper: it justifies and describes three single-stepping operations for partially-ordered executions. Section 5 discusses the implementation of these single-stepping operations in the context of Poet. Finally, in Section 6 we briefly discuss how the use of abstraction influences single stepping, and we conclude with the description of future work.

2 Related Work

Distributed applications are inherently complex and large. Accordingly, graphical visualizations can play a significant role in understanding their behaviour. These visualizations are often based on process-time representations

of events which are available to the user after the execution has taken place.

ATEMPT [9] is a *post mortem* event-visualization tool. The authors mention the use of consistent cuts as a way of breakpointing the execution. However, as a replay mechanism is not implemented, such a facility is presumably not yet available. No single-stepping facility seems to be provided in ATEMPT.

Hicks and Berman [6] describe Pangaea, an event-visualization tool allowing both post-mortem debugging and program replay. Pangaea combines the visualization and trace facilities of XPVM [8] with a new event-graph view based on process-time diagrams. No single-stepping or breakpointing facilities are available in this tool.

Paragraph [5] is a post-mortem event-debugger that provides 25 different displays to describe the collected event data. Despite the difficulties inherent in establishing a global clock, Paragraph attempts to use such a mechanism, creating a total serialization of events. The tool adopts a dynamical visualization approach which re-enacts the original live action of the execution by sequentially showing the occurrence of the events. For a more detailed study, the tool provides a sequential single-stepping mode which processes the collected data according to user requests.

Zernik et al. [18] discuss a post-mortem visualization prototype based on process-time representations. By performing different operations on these representations, it is possible to get the equivalent of the debugging operations of a sequential debugger. The authors describe an approach that allows for single stepping in terms of events. The events of the graph are partitioned into three sets, *past*, *present*, and *future* and operations are defined to enable the user to move events from one set to another. Single stepping is performed by repeatedly moving an event from *future* to *present*, and updating the sets accordingly. To select the event that changes sets, a total ordering of events is necessary. Different methods are proposed to restrict the partial order of the events to a total order, such as a global-clock ordering or a user-defined ordering. This tool appears to be the only one defining single stepping at the event level. By requiring a total serialization of events, however, they do not deal adequately with inherently partially ordered executions, such as distributed executions.

3 Theoretical Background

The representation of a distributed execution that we deal with is based on the partial order model using Lamport's *happens-before* relation [11]. Unlike this model, however, we deal not only with asynchronous messages but also with synchronous communication. For this purpose we use an extension of the relation, which is defined in [1]. With this relation, any two events can be compared to see which one occurs first in the partial order. When event a precedes event b we write $a \rightarrow b$. When $a \not\rightarrow b$ and $b \not\rightarrow a$, the two events are said to be *concurrent*, which means that their order of occurrence is not important. When a particular event is of interest, it is often useful to examine the set of events which influence it. The past of an event is used for that purpose.

Definition 1 (Past of an event)
The past of an event a *is the set of events that precede it:* $past(a) = \{b \mid b \rightarrow a\}$.

4 Single-Stepping Operations

This section defines three single-stepping operations for partially-ordered executions. These operations can be incorporated in any event visualization tool that captures the partial order of event occurrence. The next section will show how we incorporated the operations into one such tool, Poet.

4.1 The Execution Sets

Zernik et al. [18] give a method for single stepping in an event-visualization tool based on three sets of events: past, present and future. This section defines three similar sets, the *execution sets*, which are used to allow for a more concurrent approach to single stepping.

While single stepping in a sequential debugger, the execution state of the program can be divided into three parts: executed, ready

and non-ready. A statement is *executed* when the user has stepped past it, the statement that will executed subsequently is *ready*, and finally, *non-ready* describes statements that still depend on the execution of other statements. The current state of an execution can then be uniquely described in terms of three corresponding sets. Because of constructs such as loops, some statements can belong to all three sets at once. Nevertheless, since a sequential execution is totally ordered, these sets are easily computed: the *ready set* consists of exactly one statement, the next one to be executed; all statements having executed at least once are part of the *executed set*; all statements that still require execution after the next operation are part of the *non-ready set*. Whenever a single step is performed, the sets are updated to reflect the new state of the execution.

Similarly, the state of an execution in an event visualization tool for distributed applications can be described by these three execution sets. In this situation however, the sets consist of events instead of code statements. Unlike statements, since a specific instance of an event occurs only once, an event belongs to one set at a time. Furthermore, since events can be concurrent, there will be situations where more than one event can logically be the next one to execute. Therefore, it is generally impossible to determine a unique ready event. In fact, each process could have its own ready event at any time. This makes the computation of the execution sets less straightforward.

For the execution sets to be consistent, two conditions must hold: any event preceding an executed or ready event must be executed, and any event preceded by a ready or non-ready event must be non-ready. The term "preceded" is used with respect to the precedence relation characterizing the ordering of events. By enforcing these conditions, the execution is guaranteed to progress in an orderly fashion, and in particular, to adhere to the total order within each process. An alternate description is that the cut defined by the executed and ready sets is always a consistent cut [2].

Formally, we define the ready and non-ready sets in terms of the executed set as follows:

Definition 2 (Ready and non-ready sets)

- $\mathit{ready} =$
  $\{e \notin \mathit{executed} \mid \mathit{past}(e) \subseteq \mathit{executed}\}$
- $\mathit{non\text{-}ready} =$
  $\{e \notin \mathit{executed} \mid \mathit{past}(e) \not\subseteq \mathit{executed}\}$.

This definition formalizes the intuitive notion behind the concept of a ready event. Each event in the ready set is an event that does not depend on another ready event or on a non-ready event; in other words, it is an event that depends only on executed events. Therefore, for an event to be ready its entire *past* must be in the executed set. Due to the total order of events in the same process, this implies that each process will have at most one ready event. Nevertheless, there will be situations where a process will have no ready events; this occurs when the event that should be ready is preceded by an event not in executed from another process. Furthermore, by Definition 2, we can deduce that the execution sets are mutually distinct.

As the ready set consists only of events that are mutually concurrent, it will generally be smaller than the non-ready. Consequently, computing the non-ready set will usually be more computationally intense. We therefore redefine the non-ready set to facilitate its computation.

Definition 3 (Non-ready set)
The non-ready set can be defined as the events that are not in the executed or ready sets; that is, let $\mathcal{E}$ be the entire set of events then $\mathit{non\text{-}ready} = \mathcal{E} \setminus (\mathit{executed} \cup \mathit{ready})$.

The following subsections define three types of single-step operations. Each operation can be expressed in terms of manipulations on the set membership of events: moving events from non-ready to ready or executed, and moving events from ready to executed. All three operations try to advance the computation by the smallest logical amount. This common concept is explained in more detail in the next few paragraphs.

4.2 The Step Size

Single stepping should allow a user to progress through an execution in small steps. In a se-

quential program, this concept corresponds to the execution of a single assembler or code statement. In distributed applications, due to concurrency, establishing a smallest amount of execution is less straightforward.

A first approach is to allow the execution of only one event that is ready. However, transparently selecting one of many concurrent events will often seem obscure to the user, since the motivation behind it may not be clear or even known. And asking the user to choose which ready event to Execute easily leads to a very slow and cumbersome interface when stepping through a large execution. A second approach, which we prefer, avoids choosing between concurrent events as much as possible. Instead, *multiple* ready events are added to the executed set. The exact number of ready events depends on the specific operation, but the selection is logical and can easily be understood by the user. So in our approach the *smallest* amount of execution is not always a single event, and a single step will often cause multiple events to be executed at once.

4.3 Initialization

Before single stepping begins, the execution sets must be initialized. At the start, when no event has been executed, it is clear that the executed set is empty. Hence, the ready events will be ones with an empty past. Finally, the remaining events are part of the non-ready set. An *update_execution_sets* function computes the ready and non-ready sets according to the new executed set. This function can be implemented very efficiently. In the absence of any abstraction facilities (discussed below), the execution sets are updated in linear time. For a detailed description see [7].

Algorithm 1 (Initialization)
$executed = \emptyset$
update_execution_sets

In some situations it will not be desirable to start the single-stepping session from the very beginning of the execution. To deal with this issue, the initialization algorithm can be modified to assign an arbitrary but consistent event set to the executed set. For example, a user can set a breakpoint at a specific event. When the breakpoint is hit, the execution is stopped in a consistent cut [17]. The executed set can then be initialized with the set of all events that occurred so far and single stepping can resume from this point on.

The following sections describe three single-stepping operations made available to the user: *global-step*, *step-over* and *step-in*. Each of these operations supports particular scenarios encountered during a single-stepping session.

4.4 Global-Step

A simple approach at single stepping in a distributed execution allows the user to request a single step for the entire execution. The operation should cause as little execution as possible while remaining justifiable. In that respect, a global-step causes every ready event to be added to the executed set.

Algorithm 2 (Global-step)
$executed := executed \cup ready$
update_execution_sets

This operation is simple to understand for the user: it allows for an incremental progression through the execution. The global-step operation allows the user to request a small step in the entire execution, without focusing on a particular event or trace. The function actually causes the smallest logical amount of execution possible with respect to the entire set of traces and events.

There are situations where a global-step is too general; one can imagine a situation where a specific process is of interest and the user wishes to focus on that process. To address this concern, we define two stepping operations which focus on an individual process.

4.5 Step-Over

Cases where a process is of particular interest will often arise in a single-stepping session. Imagine that a message is never sent; to determine the cause of the error, the user will want to focus on the particular process where the send event should have occurred and its interactions with other processes. This type of situations requires a mean of allowing for single stepping *in the particular process* up to where

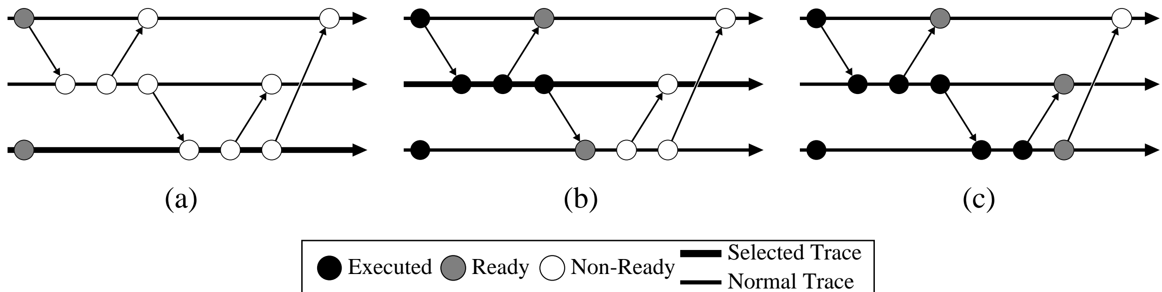

Figure 2: A single-stepping session using step-over.

the message should have been sent; other processes are permitted to execute only when it is necessary for allowing the process in focus to execute; in particular, a process fully concurrent to the one in focus will not execute at all.

Once a process has been specified for such an operation, establishing which event or events should execute is essential. Intuitively, since the operation focuses on a specific process, a single step should cause execution towards the *next event* of the process. Because events in the same process are totally ordered, a simple definition for the next event of a process can be given. In this definition (and all the ones following later), we do not distinguish between a process P and the totally ordered set of events in this process.

Definition 4 (Next event of a process P)
$\mathcal{N}_P$ is the first non-ready event of P:

$$\mathcal{N}_P = \{e \in \textit{non-ready} \cap P \mid e' \rightarrow e \textit{ and } e' \in P \Rightarrow e' \notin \textit{non-ready}\}$$

This definition is given in set notation to allow for generalization in the presence of abstraction. In the current situation however, since the definition is restricted to a particular process, $\mathcal{N}_P$ always consists of at most one (non-ready) event. In the particular case where all the events of P are either executed or ready, $\mathcal{N}_P$ is empty.

With this definition, we define a step-over operation as follows. A process P is selected by the user, then $\mathcal{N}_P$ is determined and made ready by adding the events in its past to the executed set irrespectively of the processes they belong to. This approach is similar to stepping over a function in a sequential debugger. The execution sets are then updated to reflect the new state of the execution.

Algorithm 3 (Step-over(P))
executed := executed $\cup$ *past*($\mathcal{N}_P$)
update_execution_sets

Figure 2 shows an example of a small debugging session using step-over. In the example, the process selected for the operation is represented by an emphasized line. To reach the state of Figure 2(b) from (a), a step-over was performed on the lowest process of the diagram: it forced the Next event to become ready by adding its past to executed. A step-over is then requested in the middle process. The top process is not affected by the operation since its has no events in the past of the Next event which are not already in executed. The example also demonstrates that a step-over operation can be performed on a process that has a ready event (2(a)) or does not (2(b)).

4.6 Step-In

So far, we defined two different single-stepping operations, one which is global to the execution and the other which focuses on a process by stepping from event to event within that process. We now look at a third kind of single step which is intended to support scenarios such as the one described next.

Consider the situation depicted in Figure 3, and imagine that the receive event u is known to be wrong. To determine the cause of the error, the user needs to examine the events in the past of u. Global-step does not allow one to focus on a process and therefore can

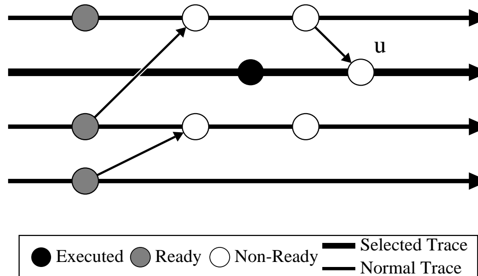

Figure 3: Motivating the step-in operation.

not serve our purpose. On the other hand, although step-over focuses on a process, requesting such a step would cause event u to immediately become ready. What the user needs is a method of single stepping *towards* event u, in such a fashion that it will be possible to find the event(s) causing u to be wrong. We can think of this requirement as a need to step in the past of u, instead of stepping over it. This approach is much like a sequential step-in operation which jumps to the beginning of a function instead of automatically executing it. Based on this requirement and on the traditional step-in function, we define a step-in operation.

The step-in function should cause the smallest amount of execution leading to the eventual occurrence of the Next event of the selected process. When a process is selected for a step-in, two situations are possible: the process has a ready event or it does not. In the first case, as only the smallest amount of execution should occur, the step-in operation will cause only the ready event in question to become executed. This approach truly leads to the smallest possible amount of execution for a single process. On the other hand, if the process has no ready event, its Next event, $\mathcal{N}_P$, is determined and all ready events in its past are added to the executed set. Doing so allows the execution to progress towards $\mathcal{N}_P$.

Algorithm 4 (Step-in(P))

If *(P has a ready event e)*
  Then $executed := executed \cup e$
  Else $executed := executed \cup (past(\mathcal{N}_P) \cap ready)$
Endif
update_execution_sets

The step-in operation is interesting for two main reasons. The first is that it allows the user to cause a single ready event to Execute. The user does so by requesting a step-in for the process containing the specific ready event. In most cases the step-over operation does not allow the user to do this. Instead, a step-over would not only execute the ready event but also everything in the past of the Next event of the trace. The other use of a step-in is the ability to gradually execute events in the past of a specific event, in such a way as to make the execution progress towards this specific event. A step-over would immediately cause the event to become ready instead of gradually getting closer to it.

One will notice that both step-in and step-over cause execution only in the past of $\mathcal{N}_P$, consequently, other events are unaffected. Furthermore, as multiple step-in operations add ready events in the past of $\mathcal{N}_P$ to executed, the execution will progress within this past until it contains are no more ready events. When this situation occurs, it implies that $\mathcal{N}_P$ is itself ready, which is what is accomplished by a single step-over. This shows that the step-in operation performs small execution increments within a step-over.

In addition, it is worth noting that performing a step-in within every process containing a ready event will result in the same overall execution state as a global-step operation. Although this observation shows that a global-step can be emulated with multiple step-in operations, providing the user with the capability to do a global-step simplifies stepping through large distributed executions with many processes and a high degree of concurrency.

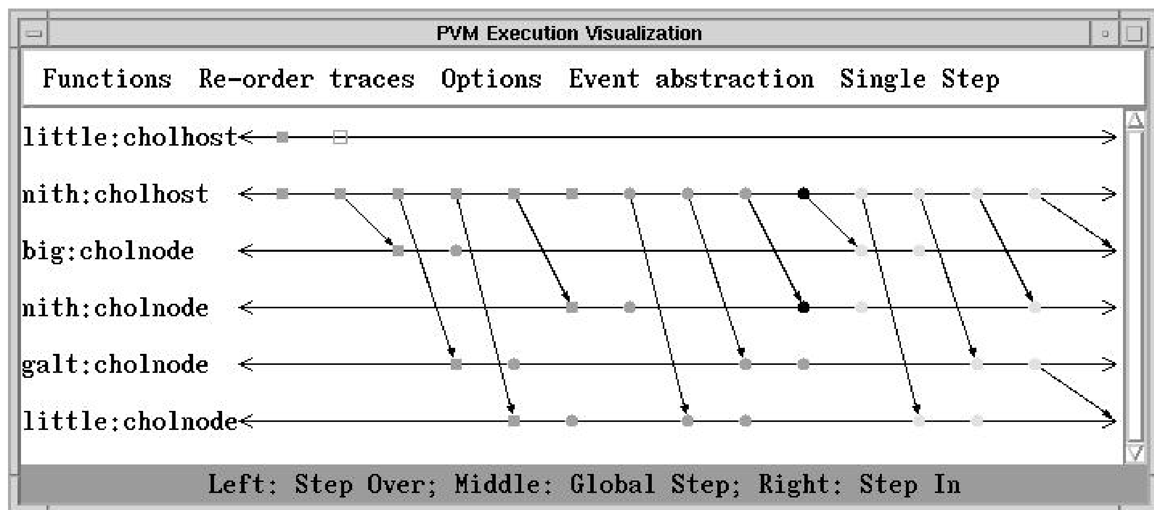

Figure 4: Poet display: initial scenario

5 Single Stepping in Poet

The concepts described above are currently implemented in Poet. When the single-stepping option is turned on, the three execution sets are initialized. The events are colored differently depending on their set membership. In the following gray-scale renderings of Poet's color display, the darkest events are part of the ready set, while the lightest are members of the non-ready set; the shading that falls between the other two represents executed events. Using the mouse, a user can then chose to perform one of the three available step functions. In the case of step-over and step-in operations, the user must select a trace in which the step will take effect. The following figures give examples of each of the three step functions.

Figure 4 shows Poet in the middle of a single-stepping session. The execution consists of six processes communicating asynchronously. The current ready set contains two events, one on trace "nith:cholhost" and the other on "nith:cholnode". These two events are ready because their past is fully executed. On the other hand, trace "big:cholnode" for example, does not have a ready event. As is shown, the first two events of this trace are executed, therefore, its third event is the potential ready one. However, this event cannot be part of the ready set, since it depends on a non-executed event, namely the ready event of "nith:cholhost". All events that are successors to the two ready events are non-ready.

From this situation, performing a global-step would result in the display shown in Figure 5. Both ready events of Figure 4 are now executed and the new ready events have been determined. The execution of the ready event of "nith:cholhost" enabled two events to become ready: its direct successor in the same trace, and the first non-ready event of "big:cholnode". Executing the ready event of trace "nith:cholnode" allowed its direct successor to be ready. All other non-ready events still depend on non-executed ones and therefore, remain non-ready after the global-step.

Instead of a global-step, the user could have selected a step-over from the situation shown in Figure 4. In such a case it is necessary for the user to specify a trace in which the step should be performed. Figure 6 depicts the result of a step-over in trace "galt:cholnode". This operation caused the first non-ready event of the selected trace to become ready; this was accomplished by forcing the execution of the past of this event, i.e., three non-executed events in "nith:cholhost". With these three events now executed, not only does "galt:cholnode" have a ready event but events in "nith:cholhost", "big:cholnode" and "little:cholnode" also be-

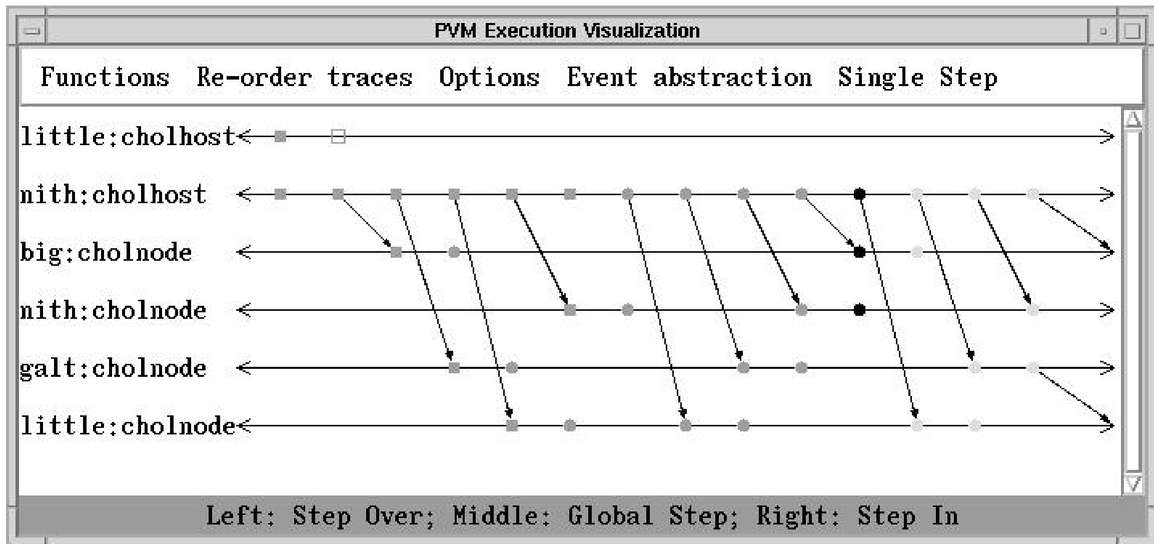

Figure 5: Performing a global-step

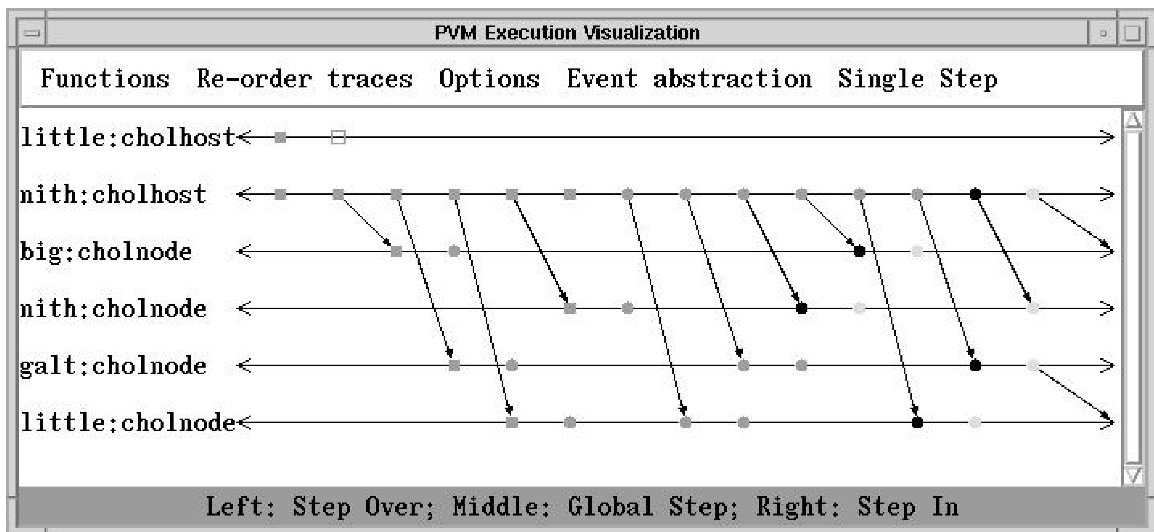

Figure 6: Performing a step-over in "galt:cholnode"

come ready.

Finally, Figure 7 shows the result of a step-in request in trace "galt:cholnode" starting from the situation of Figure 4. Unlike for step-over, the first non-ready event of the trace in question is not forced to become ready; instead, every ready event in its past is added to executed. In this case, the ready event in "nith:cholhost" belongs to the past of the first non-ready event of "galt:cholnode" while the other is concurrent to it. Therefore, only the ready event of "nith:cholhost" is added to the executed set. When updating the execution sets two new events become ready, one on "nith:cholhost" and the other on "big:cholnode".

The current implementation deals with both synchronous and asynchronous communication although only asynchronous messages are

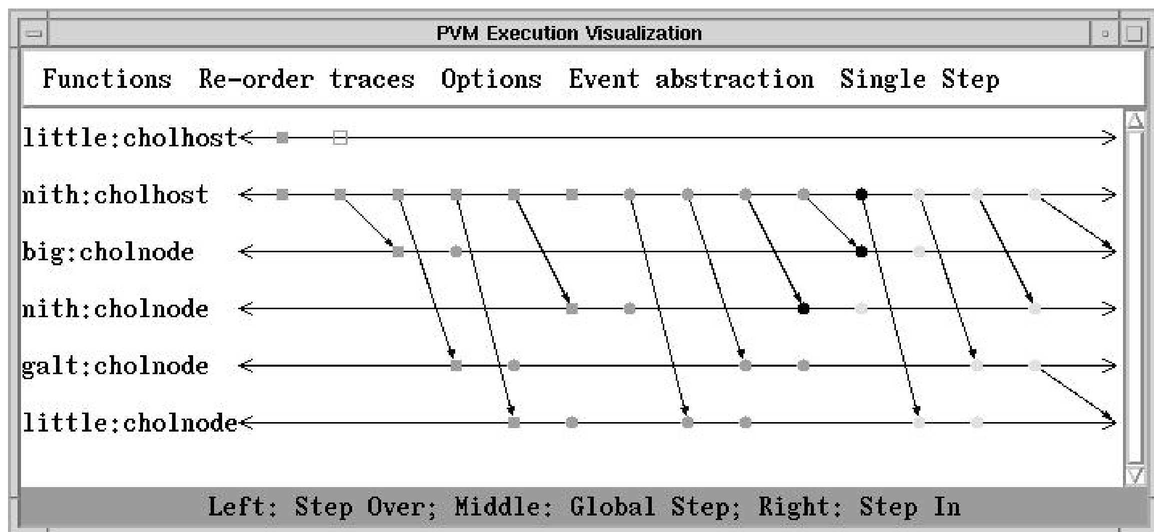

Figure 7: Performing a step-in in "galt:cholnode"

shown in the examples. The colors used to distinguish the execution sets are configurable by the user, and the single-stepping option can be turned on and off at any time. The implementation also handles the clustering of processes and the abstraction of events, as is briefly explained in the next section.

6 Single Stepping and Abstractions

Due to the overwhelming amount of information that can be collected by an event-visualization tool, reducing the quantity of detail presented to the user is necessary. Poet provides two methods for such a purpose: process clustering and event abstraction.

Process clustering, as the name implies, attempts to reduce the number of processes presented to the user. In many situations, certain processes, as well as the interactions between them, are of no interest. If such processes could be grouped, or clustered together, the user would be presented only with a single, higher-level, process-like entity, instead of the many processes hidden within. More details on process clustering can be found in [3, 13].

The introduction of process clustering complicates the task of single stepping. Since a process cluster is made up of more than one process, it must be treated as a partial ordering of events. We have seen that in the case of partial orders, more than one event can be ready at the same time; therefore, when determining the Next event of a cluster, there may be more than a single one. It is necessary to redefine the step-over and step-in operations so as to deal with this possibility. Furthermore, the user's view of events no longer corresponds directly to the low level execution, since some events have been hidden by the clustering operation. To maintain the advantage of clusters, it is important that single-step operations behave as if hidden events did not exist. Currently, the theory behind single stepping with process clusters has been developed and the single-stepping algorithms have been extended accordingly [7].

The second method used to reduce the amount of detail presented to the user is called event abstraction. Instead of clustering processes together, this method groups events into higher-level events. More details on event abstraction are given in [3, 10]. In the presence of abstract events new precedence between events is introduced. This is caused by the dependence created between the events constituting an abstract event. Furthermore, to respect the fundamental goal of event abstraction, each abstract event must be treated as a single en-

tity. This implies that a step function affects an entire abstract event, irrespectively of its constituent events. Once both of these issues are taken into account, the single-stepping operations can be extended to deal with abstract events. The description of these extensions can be found in [7].

7 Conclusions and Future Work

This paper discusses the fundamental issues related to single stepping in event-visualization tools. Three single-stepping operations have been defined to deal with partially-ordered distributed executions. These operations were implemented in Poet, a visualization tool developed at the University of Waterloo. Executions can be visualized post mortem, using the single-stepping operations to incrementally step through and understand the monitored execution. Alternatively, Poet can deterministically replay the execution, using the recorded event data as a reference to break potential (non-deterministic) races [16]. During *replay*, sequential debuggers can be attached to the processes of the distributed application [17]. In such an environment, single stepping allows the user to examine the internal state of each process while incrementally progressing through the execution.

Poet also supports abstraction facilities to deal with the potentially huge amount of event data generated by non-trivial distributed executions. We have collected experience with event trace data consisting of multiple tens of thousands of events, generated by hundreds of processes. The single-stepping operations presented in this paper have been extended to deal with process clusters and abstract events.

An as yet open problem is how to define and implement single stepping in the absence of event data. As mentioned before, our approach assumes that we know about the occurrence of events *a priori*, either because we visualize the execution post mortem or because we have access to the reference event data during deterministic replay. Without this information, it is, for example, not possible to define a step-over operation, which requires knowledge about the past of an event that has not yet occurred. Without this knowledge, it becomes impossible to decide which processes should be blocked and which processes should be allowed to execute and by how much.

Acknowledgments

The work described here has been supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). It benefited from various discussions with the members of the Shoshin research group at Waterloo, in particular Prof. David Taylor.

About the Authors

Marc Khouzam completed his Masters in computer science from the University of Waterloo in September 1996, and his BSc in mathematics and computer science from McGill University in May 1994. He is currently working for Nortel in their public carrier networks group. His e-mail address is mkhouzam@ccnga.uwaterloo.ca.

Thomas Kunz received the Dr. Ing. degree from the Technical University of Darmstadt, Federal Republic of Germany, in May 1994. He is currently an assistant professor at the University of Waterloo, Ontario, Canada. His research interests include load balancing in distributed systems, distributed debugging, management of distributed applications and systems, mobile computing, and reverse engineering/program understanding. He can be reached at: Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1. His e-mail address is tkunz@uwaterloo.ca.

References

[1] J. P. Black, M. H. Coffin, D. J. Taylor, T. Kunz, and A. A. Basten. Linking specification, abstraction, and debugging. CCNG Technical Report E–232, University of Waterloo, November 1993.

[2] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[3] W. H. Cheung. *Process and Event Abstraction for Debugging Distributed Programs.* PhD thesis, University of Waterloo, 1989.

[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

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

[6] L. Hicks and F. Berman. Debugging heterogeneous applications with Pangaea. In *Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 41–50, Philadelphia, Pennsylvania, May 1996.

[7] M. Khouzam. Single stepping in event-visualisation tools for distributed applications. Master's thesis, University of Waterloo, 1996.

[8] J. A. Kohl and G. A. Geist. The PVM 3.4 tracing facility and XPVM 1.1. Technical report, Computer Science & Mathematic Division, Oak Ridge National Laboratory, Oak Ridge, TN, USA, 1995.

[9] D. Kranzlmüller, S. Grabner, and J. Volkert. Event graph visualization for debugging large applications. In *Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 108–117, Philadelphia, Pennsylvania, May 1996.

[10] T. Kunz. Visualizing abstract events. In *Proceedings of the 1994 CAS Conference*, pages 334–343. IBM Canada Ltd. Laboratory and National Research Council of Canada, October 1994.

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

[12] D. J. Taylor. A prototype debugger for Hermes. In *Proceedings of the 1992 CAS Conference*, volume 1, pages 29–42, November 1992.

[13] D. J. Taylor. The use of process clustering in distributed-system event displays. In *Proceedings of the 1993 CAS Conference*, pages 505–512. IBM Canada Ltd. Laboratory and National Research Council of Canada, October 1993.

[14] David J. Taylor, Thomas Kunz, and James P. Black. Achieving target-system independence in event visualisation. In *CD-ROM Proceedings of the 1995 CAS Conference*, Toronto, Ontario, Canada, November 1995. IBM Canada Ltd. Laboratory and National Research Council of Canada. 17 pages.

[15] J. C. Yan. Performance tuning with AIMS - an automated instrumentation and monitoring system for multicomputers. In *Proceedings of the 27th Hawaii International Conference on Systems and Sciences.* ACM, January 1994.

[16] Yuh Ming Yong and David J. Taylor. Performing reply in an OSF DCE environment. In *Proceedings of the 1995 CAS Conference*, pages 52–62, Toronto, Ontario, Canada, November 1995. IBM Canada Ltd. Laboratory and National Research Council of Canada.

[17] I. Yu. Integrating event visualization with sequential debugging. Master's essay, University of Waterloo, 1996.

[18] D. Zernik, M. Snir, and D. Malki. Using visualization tools to understand concurrency. *IEEE Software*, 9(3):87–92, 1992.