Poet: Target-System-Independent Visualisations of Complex Distributed-Application Executions

T. KUNZ[1], J. P. BLACK[1], D. J. TAYLOR[1] AND T. BASTEN[2]

[1] *Dept. of Computer Science, University of Waterloo, Canada*
[2] *Dept. of Math. and Comp. Science, Eindhoven University of Technology, The Netherlands*
Email: tkunz@uwaterloo.ca

Designing and implementing a visual debugger for distributed programs is a significant challenge. Distributed applications are often large and frequently exhibit a high degree of complexity. Consequently, a debugger must address problems of complexity and scale in at least two ways. First, appropriate user interfaces should allow a user to manage the vast amount of information typically obtained from distributed executions. Second, the tool itself, in handling this information, should be implemented efficiently, providing a user with reasonable response times for interactive use. Our research efforts, concentrating on these problems, have led to the development of Poet, a tool for the collection and presentation of event-based traces of distributed executions. Poet makes as few assumptions as possible about characteristics that must be possessed by all target environments. Information describing each target environment is placed in configuration files, allowing a single set of Poet executables to be used for all target environments. Comparing Poet's performance to XPVM, the standard visualisation tool for PVM executions, reveals that this target-system independence does not impose a performance penalty.

Keywords: programming environments, parallel and distributed debugging, visualisation, event abstraction, clustering, PVM, OSF DCE, C++

Received —; revised —; accepted —

1. INTRODUCTION

One of the barriers to distributed-application development is the lack of good tools to reduce development time and effort. With distribution come concurrency, scale, and complexity. Instead of a single process on a single machine, the application involves a potentially large number of machines, processes, threads, and events. While sequential debuggers and techniques are still useful for distributed applications, they typically do not address concurrency, scale, or overall complexity directly.

One body of research in distributed debugging concerns how to present the behaviour of a large distributed application to a developer so that his debugging effort is reduced, and his understanding of the application enhanced. Another body of research concerns how to extend sequential notions of breakpoints and control of the target application 21, 40. Our efforts have been concentrated on the former, and, in particular, on how to address problems of scale and complexity in the collection and presentation of event-based traces of the behaviour of the target application. They have led to theoretical results on process and event abstraction, practical studies on automatic analysis of partially ordered sets of events (sometimes called event traces), and the development of Poet, a tool for the visualisation of Partially Ordered Event Traces.

An important approach to coping with complexity and scale is to apply abstraction in the process and event domains, so that the developer can view the execution in terms of units larger than processes and primitive events. In addition, a distributed debugger should scale well to large numbers of events and processes. In general, a single level of abstraction is not sufficient. Therefore, we propose a hierarchical approach, which allows the behaviour of a distributed application to be viewed and examined at various levels of abstraction. Starting with a high level of abstraction ensures that the amount of information displayed is small enough to be manageable. The likely cause of an error is detected by successively identifying the erroneous part of an execution and re-examining this part at a lower level of abstraction.

At all levels of abstraction, the user is presented with a similar view of the execution: a set of traces (represented by horizontal lines), each consisting of a sequence of events (represented by symbols such as dots and circles), and a number of arrows linking events in different traces. The placement of events along traces indicates the temporal relationship between events, with the understanding that time flows from left to right. This temporal relationship can be based either on the physical wall-clock time of event occurrence or some notion of logical time, as discussed in more detail below. Traces correspond to processes or clusters. Clusters are obtained when one or more processes or other clusters are combined, yielding a more abstract view of the process structure of a distributed application. Clusters are represented by one or more interface traces, depending on whether the events occurring at the interface are totally ordered or only partially ordered. Interface events in a cluster are those that have a partner outside the cluster; internal events, messages, and process creation or destruction inside a cluster are omitted from the display. Events can be divided into primitive events and abstract events. Primitive events are the lowest-level observable events occurring in an execution. Abstract events are simply sets of primitive events. Figure 1 is a screen dump showing both process and event abstraction. To distinguish a cluster from an application process, Poet highlights cluster traces, typically with a blue band "behind" the trace; in this black-and-white rendering, we have instead shown cluster names in upper case. The example shows clusters with a single, sequential interface trace only. Primitive events are depicted as open and filled dots and squares. Abstract events spanning several traces are represented by rectangles that intersect those traces containing events belonging to the abstract event.

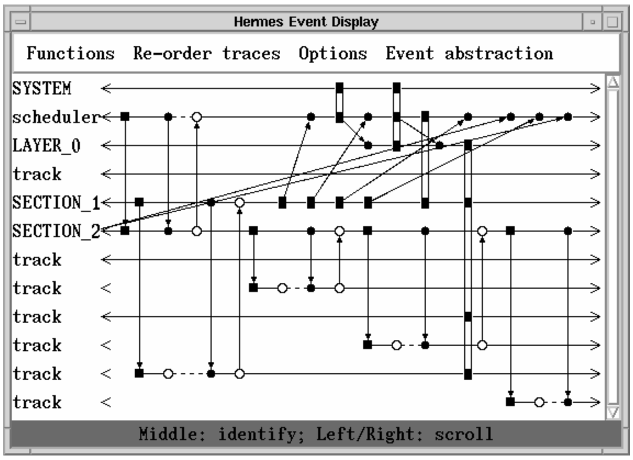

FIGURE 1. Part of the execution of an airport shuttle simulation

In a large execution, automatic assistance will be required for both process and event abstraction: clearly, the interactive developer will be unimpressed with having to specify that potentially thousands of primitive events should belong to an abstract event, or even how a few tens of processes should be clustered hierarchically.

Furthermore, even in the presence of both process and event abstraction, there may be more traces and events than will fit on a single screen. To be usable, Poet must be able to scroll through the event traces in both the trace and time dimensions, with a response time acceptable to the interactive developer.

Last but not least, many different environments now exist for distributed and parallel computing. While each environment has its own specific features and problems, developing, debugging, and maintaining applications in all these environments is a non-trivial task. Thus, a tool intended to assist in understanding and debugging applications in distributed and parallel environments will be most useful if it is not tied to a particular environment, but is useful across many such environments. Poet was designed with target-system independence in mind, and this paper will discuss some of the ways in which we achieved our goal.

Packaging the ideas described above into a usable tool with acceptable performance is not just a simple implementation exercise. It requires attention to a number of conflicting design requirements, ranging from the probe effect on the target program, to multiple data representations in a heterogeneous environment, to the careful design of optimisations that can lead to dramatic improvements in performance while maintaining the theoretical and algorithmic elegance underlying the implementation.

The paper is organized as follows. Section 2 compares Poet to other distributed-debugging tools and points out the unique aspects of our work. Section 3 discusses how we model distributed executions across all target environments and Section 4 describes the basic Poet display. The next two sections describe the two abstraction facilities supported by Poet: clustering and event abstraction. Each section addresses the issues of underlying theory, automatic detection or derivation of these abstractions, and Poet's user interface. Section 7 discusses our approach to achieving target-system independence and Section 8 examines the performance of Poet, comparing it to a popular visualisation tool for PVM executions, XPVM 23. Section 9 concludes the paper with a summary of the implementation status of Poet and a brief discussion of future work.

2. RELATED WORK

A number of tools to facilitate event-based distributed debugging have been developed in recent years. An early, non-graphical approach is described in 9, where event-based behavioural models are compared against an event trace. To cope with the large amount of trace data, distributed debuggers soon provided graphical visualisations of this trace data 14, 23, 24, 25, 41.

The majority of these visualisation tools provide process-time diagrams that are superficially similar to Poet's displays. However, event placement in these diagrams is usually based on the real-time of event occurrence only. Such real-time displays are useful to examine performance issues. However, they tend to obscure the logical structure of the execution because of irrelevant timing details 18, 25. While Poet supports real-time displays as well, this paper concentrates on displays that are built based on the partial order of events 30, emphasizing the logical structure of the execution and supporting fault debugging.

Poet differs from other tools also in its target-system independence and its unique abstraction facilities to view the execution behaviour at high abstraction levels. Some tools are either tied to a particular programming environment (for example 23) or a particular language (for example 14). Poet, on the other hand, is actively being used to visualise parallel and distributed executions in a number of different target environments. We make it available to students working on assignments in an object-oriented distributed programming language, use it within our research group to develop parallel applications in PVM 16, and Poet is used internally by developers at the IBM Toronto lab, working on distributed programming environments. Each target environment has its own programming paradigm and programming language.

Finally, the more recent visualisation tools all use abstraction to reduce the apparent complexity of distributed executions. Some abstraction facilities are based on zooming operations along the time axis 14, 23. Such an approach fails for displays based on the partial order of event occurrences, with its implied lack of a global time. Zooming along the time axis also does not address abstraction in the trace dimension, which is essential for dealing with applications that consist of many sequential entities (threads, processes, etc.). Zernik et al. 41 suggest an event-abstraction operation based on the partial order. This abstraction operation, however, cannot be applied recursively, limiting the number of possible abstraction levels. Kranzlmüller et al. 24 describe multi-level abstraction facilities in both the trace and event domain. However, since their tool is not based on the partial order of event occurrence, no attempt is made to correctly reflect the underlying partial order at higher abstraction levels. As discussed in 31, this might potentially mislead the user. Poet provides multi-level abstraction facilities based on the partial order in both the trace and event domain, as discussed in the next section.

3. THEORETICAL FOUNDATIONS

For the sake of simplicity and generality, our model of computation makes few assumptions about the programs being debugged. Our model contains two basic concepts: traces and events. A trace represents some entity with sequential behaviour and events represent activities of interest. Each event is assumed to be associated with a single trace. In addition, events may be connected to each other, as event pairs, and pairings may be synchronous or asynchronous. In some sense, that is the complete model and it is simply a matter of mapping the behaviour of any particular system onto it. One frequently encountered communication primitive in distributed system is the *remote procedure call* or RPC, for example in OSF DCE 33. One process (the client) invokes a procedure in another process (the server) by sending a synchronous message to the server, identifying the procedure to execute as well as the necessary parameters. The client then blocks, waiting for the server to reply with another synchronous message, indicating that the remote procedure terminated and returning the return status and values. In such an environment, each trace represents a process and events represent RPC interactions as well as "local" activities such as process initiation and termination. Specifically, an RPC will be represented by four events: call, call-receive, reply, and reply-receive, with the first two being a synchronous pair (and representing the first message exchange) and the last two also being a synchronous pair (representing the second message exchange, in the opposite direction).

In this paper, we often use the term *process* to indicate a component of a distributed application. It should be understood, however, that this includes as well a thread, a semaphore, or any other component within a distributed application with sequential behaviour. Similarly, we will refer to events that initiate an event pairing as *send* events and events at the terminating side as *receive* events. In the RPC example above, call and reply are the send events, call-receive and reply-receive the receive events. However, a send event could equally well be a process-creation event (with the matching receive event being the startup of the newly created process) or the entrance of a process into a monitor (with the corresponding receive event indicating the assignment of a given monitor to the acquiring process).

Distributed applications can be executed on different computers that may be physically separate. Since clock-synchronization algorithms depend on bounded communication latency 30, which we do not assume, we do not assume the existence of a global clock either. Traces can be created and destroyed during execution, and so their number is not known *a priori*.

The order in which events happen is important to debugging. Indeed, presenting the order of events in traces faithfully is a primary goal of Poet, which is based on the *partial order* model of event occurrence introduced by Lamport 30.

One problem with using partial orders is that it is more difficult to answer the question "how are these two events ordered?" If a total ordering of events is assumed, the question can be answered by comparing two numbers—the global times associated with the two events. Vector timestamps 15, 34 provide a way to determine the order of two events in a partial order quickly. The desired property is that event e precedes event f if and only if $T_e < T_f$, where T_e denotes the timestamp of event e. Two events g and h are concurrent if neither $T_g < T_h$ nor $T_h < T_g$.

The timestamping algorithm in Poet is based on one by Fidge 15, modified for synchronous communication by Cheung 13. The timestamp of an event is a vector of integers, one component for each process in the computation. Element-wise comparison of timestamps is sufficient to determine the order of events: e in trace i precedes f in trace j if and only if $T_e[i] < T_f[i]$ (13, 15, 34).

Timestamps can be calculated by carrying with each message a timestamp whose size is in the order of the number of processes. If the exact number of processes or an upper bound for this number is known a priori, vectors can be used to represent this timestamp 13, 15, 34. This is necessary if the logical timestamps are needed by some on-line algorithm that is part of the application itself. In our case, however, the timestamps are only required in the debugger (not the application), and only when the execution is being visualised. At execution time, it is sufficient to record a minimal amount of information for each event: given an event record, it must be possible to identify uniquely a single partner event (send or receive) in the other process. Typically, this involves adding a small, constant-sized quantity of information to each message (sender identification, if no other unique message identifier is available, and a local event counter). At debug time, the timestamp code simulates the recorded message exchanges to calculate full vector timestamps.

4. BASIC POET FACILITIES

The original intent for the Poet display was to draw a process-time diagram, which is a common representation of distributed executions. That is, to draw a set of vertical lines, one for each process in the application, placing a symbol on the appropriate line for each event, with a connecting arrow between events that pair (using a horizontal arrow for synchronous pairing and a sloping arrow for asynchronous pairing). The events would be positioned so that if event e preceded event f, then e would be positioned higher than f. That is, the top-to-bottom ordering of events would be consistent with the partial order. The displays were to be built making heavy use of timestamps to determine event precedences. As the event display has evolved and been refined, this basic nature has not changed, but many details were added to make the display easy to use and informative. As well, it became clear that using horizontal, rather than vertical, lines to represent processes would be useful, in particular making possible the display of process names for all processes. The discussion below assumes this horizontal orientation.

Figure 2 shows part of an execution of the Hermes 37 helloworld program. Because of the use of processes in Hermes where other languages would use procedures, even helloworld provides a large set of events. Each line represents a Hermes process. Time flows from left to right; different event types are shown with different round or square symbols; and blocked processes are shown with a dotted line. The displayed part of this program involves only synchronous communication, so all event pairings are vertical. The middle mouse button has been pressed on event 2 of process helloworld, and so additional information is displayed, including the event type, event number within the process, and predecessor and successor events are highlighted by using different colours (not obvious in this black-and-white rendering).

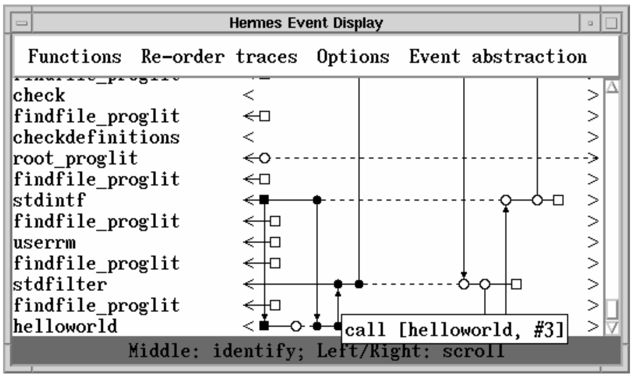

FIGURE 2. Part of the execution of helloworld

An obvious practical problem is how to display the complete execution history of an application, which is usually larger than the available display-screen space. If there are more processes than will fit, a standard scrollbar is used. If there are more events than will fit, a scrollbar is not an appropriate solution. A scrollbar requires some function that can associate scrollbar positions with data to be displayed. For event displays based on real time, an obvious function exists. Since the displays here are based on the partial order of events, some alternative means is required to move backward and forward in the execution history.

The technique adopted is based on selecting an event and specifying that it should be dragged to the left or right edge of the display area. An appropriate event display is then constructed, where "appropriate" means that it must be consistent with the partial order, and events in the previous display should be removed only if necessary. The underlying idea is that, instead of imagining that events are in fixed positions relative to each other, the user should imagine that events are beads on wires, with some of the beads connected to each other. Scrolling through the event history involves sliding a bead along its wire, with other beads moving as necessary. An event-scrolling algorithm based on this principle provides a simple, intuitive interface for the user, but is quite complex. In fact, the above is not even an adequate description of the requirement for the algorithm. The key issue not addressed is the meaning of "[remove] if necessary." The obvious interpretation, "if it is impossible to build a display consistent with the partial order," proved to be inappropriate. The constraint needed to be refined so that when an event was dragged to a new position, events related to it in the partial order would be given precedence for placement in the display over events that were concurrent with it and present in the previous display. The basic principle of the algorithm, for the case that event e is moved to the left, is to put all successors of e into the display, then predecessors of those events, successors of the most recently added events, and so on. In all cases, of course, the set of events used is constrained by the horizontal size of the display window.

If a user performs an extensive examination of an event history, it is likely that certain events will be noted as interesting and ones that potentially need to be returned to later. Thus, the user might create a list of these interesting events, with entries like "event 65 in `root`". To eliminate such tedious off-line record keeping, we added a facility for event tagging. That is, the user can associate an arbitrary label with an event and can request that the display be scrolled to an event with a specified label.

It was clear that a detailed description of each event should be available on request, but initial use of the debugger showed that some information also needed to be present statically in the display. Thus, the debugger was modified to show call and call-receive events as filled circles, reply and reply-receive events as open circles, and so on. The use of a few different symbol types greatly improved the display, making most inquiries for event details unnecessary.

It is possible to determine the partial order from the event display by noting the sequencing of events within a process and the connections between processes. For convenience, the debugger can also explicitly mark the displayed predecessors and successors of a specified event. In complex traces, this information can be a significant convenience for the user.

To allow users to tailor the display to their needs (or expectations), we also added facilities for changing the order of the displayed traces. The simplest of these facilities moves a single trace (or set of traces representing a single cluster) to a designated position. Although this "move one" capability is clearly sufficient to implement any change, various other facilities are provided to make major rearrangements convenient. For example, it is possible to request that the lengths of the connecting lines that join communicating events be minimized. (Since this problem is computationally intractable, a heuristic is used. In practice, its results are generally quite good.)

Although the debugger is built around the notion of a partial order that relates the events to each other, the real time at which events occurred is also often of interest. If performance problems, rather than functional problems, are being studied, real-time information is of obvious value. The significant problem in providing real-time information to the user is that we did not wish to eliminate the partial-order relationship from the real-time display. If clocks were perfectly synchronized, real times would always be consistent with the partial order, but real times obtained from actual clocks may not have this property. Since we do not, in general, have any control over the clocks used in the target environment, we decided to adjust the real times assigned to events after they were delivered to Poet. The adjustment is performed using an extension of Lamport's algorithm 30. Lamport's algorithm can greatly distort the apparent lengths of intervals within individual processes, so modifications were included to minimize such distortions. Once the modified Lamport algorithm is used, a real-time display is simple to construct. In our implementation, the user can choose the scale of the time axis (within certain limits), and large gaps are replaced with "cut" marks.

5. PROCESS ABSTRACTION

Process abstraction allows groups of processes to be combined into *clusters*, eliding the interaction among processes in the cluster. Combining clusters into higher-level clusters leads to an abstraction hierarchy. Problems that must be addressed when clustering processes include: how to represent a cluster, what processes and/or subclusters should be combined, and how to provide an intuitive user interface to manage and navigate through a cluster hierarchy.

5.1. Cluster Representation

The visual display of events entering or leaving a cluster should resemble that for processes, so that the developer sees essentially the same type of display at different levels of abstraction. However, if the behaviour of the cluster involves visible concurrency, it is important not to distort the actual precedences by, for example, placing concurrent events as if one preceded the other across the cluster interface. Representing a cluster by *one* totally ordered event trace is therefore, in general, not possible. Instead, a *set* of totally ordered traces is needed, with the number of traces being equal to the dimension of the partial order of the events in the cluster interface. Discussions of some of the problems in constructing cluster-interface traces can be found in 13. Poet takes a simple approach 38: as concurrency is encountered during display drawing, the number of trace lines associated with a cluster is increased as necessary.

5.2. Cluster Membership

A second problem is the identification of the appropriate processes and subclusters to be combined. For large distributed applications, the number of processes will be large enough to make any manual clustering both tedious and error-prone. We implemented a number of heuristics to derive cluster hierarchies automatically. In general, we expect these heuristics to provide good initial clusters that can then be modified and fine-tuned by hand.

A first algorithm creates a very simple cluster hierarchy consisting of two clusters, *SYSTEM* and *APPLICATION.* The first cluster contains all processes known to belong to the runtime system, the second cluster contains all other processes. The definition of the *SYSTEM* cluster depends on the specific target environment in which Poet is executing.

Clustering by stream is another simple tool. It clusters those processes together that report events on the same TCP stream. Depending on the target environment, a stream might correspond to all processes on a single host, to all threads in a single Unix (heavy-weight) process, all Hermes "processes" implemented by a single Hermes interpreter (which is a single Unix process), and so on. The motivation behind this tool is that distributed applications often exploit locality, and so processes that report events on the same stream "belong" together in a cluster.

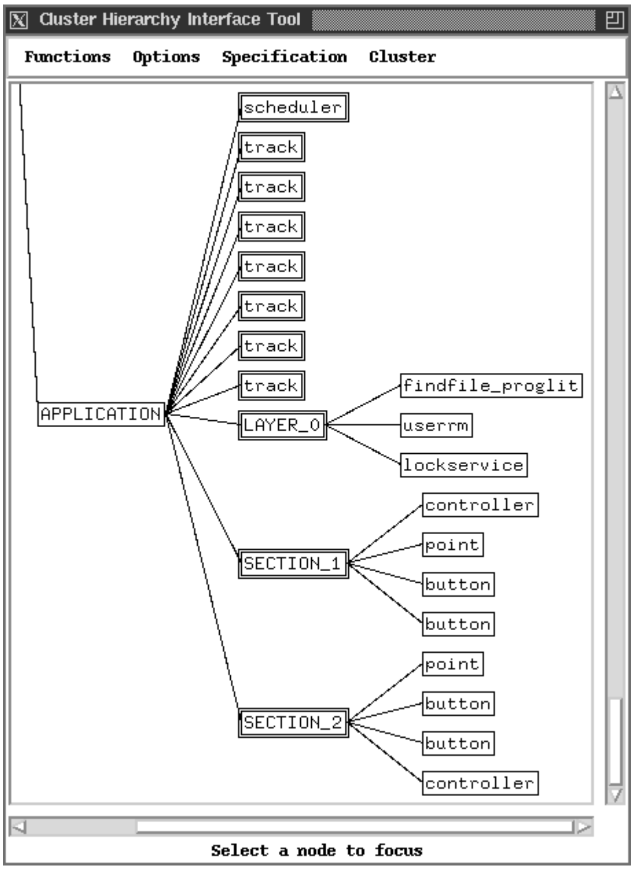

FIGURE 3. CHIT interface

These two tools derive simple (one-level) abstraction hierarchies, exploiting target-system-specific semantics. Our third tool is a more general automatic clustering tool. Initially, we compared two approaches: one clustering processes solely on the basis of their interprocess communication (IPC) frequency during runtime, and another that makes use of additional information. This additional information consists of a characterization of the application processes derived by a static source analysis, and specific clustering rules that combine this semantic data with IPC information. Experimental results from a number of Hermes applications show the latter approach to result in better clusters 27.

5.3. Cluster Interface

Figure 1 showed a display that contains four clusters derived by the automatic clustering tool. Without clustering, the lowest-level visualisation of this particular execution requires 8.5 windows in the process dimension to display all 139 processes. Using clustering, all process and cluster traces fit into the display, so no scrolling in the process dimension is necessary.

Poet contains a component called CHIT, the Cluster Hierarchy Interface Tool, to visualise the cluster structure as a tree: the internal nodes are clusters and the leaf nodes are traces or empty clusters. Figure 3 shows a CHIT display, containing the relevant part of the application processes shown in Figure 1. All application processes (i.e., all processes that are not grouped into SYSTEM as part of the runtime system) are part of the high-level cluster `APPLICATION`, which contains three non-empty subclusters and eight traces. With the mouse, the user can modify the cluster structure by modifying the tree, using simple operations such as "create new cluster below node X", and "move A, B, C, D into cluster Y". The user can also save the current cluster hierarchy in a cluster-description file, or read a cluster-description file created earlier by manual or automatic means 27.

Given such a cluster structure, there is also the problem of selecting what clustering is to be used for the event display. That is, although a cluster has been defined to hide a set of traces and other clusters, a user may still want to see the cluster "opened up" on occasion. The notion of a "debugging focus" captures this requirement. Formally, it is a cut across the cluster-hierarchy tree, including exactly one node on each root-to-leaf path. The user can modify the focus by simply selecting a node that should be placed in it. A minimal set of other changes is then made to create a legitimate focus containing that node 38. In Figure 3, traces and clusters in the current cut are surrounded by a double box. This focus is reflected in the display in Figure 1.

6. EVENT ABSTRACTION

Event abstraction allows groups of primitive events to be combined into "abstract events." As with clustering, the motivation is to reduce complexity by eliding unwanted details. Event abstraction may also allow "reconstruction" of the high-level viewpoint seen by a programmer; for example, an RPC could be recognized and displayed as a single event.

6.1. Theory

Like process abstraction, event abstraction should not introduce spurious precedences or hide real ones. Ideally, the precedence relation on abstract events should "act like" the precedence relation on primitive events, that is, it should be a partial order. However, this issue is surprisingly more complex than it might appear.

The most general potential definition of an abstract event is an arbitrary subset of the events in a computation. Unfortunately, with this definition, abstract events do not have the same properties as primitive events. Primitive events are atomic, and a partial order is sufficient to describe the causal ordering of primitive events. Abstract events are not necessarily atomic, and, in general, partial orders are not sufficient to describe all the possible causal relations between them. For example, two abstract events can overlap in time and interact causally; one cannot be said to precede the other in the usual sense of the term.

Based on similar observations, Lamport 31 identifies two meaningful precedence relations on abstract events. The first is to say that an abstract event A precedes an abstract event B if *every* primitive event in A precedes *every* primitive event in B. The second is to say that an abstract event A precedes an abstract event B if *at least one* primitive event in A precedes *at least one* primitive event in B. The former is often referred to as the *strong* precedence relation, while the latter is called the *weak* precedence relation.

The strong precedence relation has several desirable properties, including asymmetry and transitivity. Thus, this relation is a partial order. Unfortunately, strong precedence does not yield much information about the causality structure of an abstract view of an execution. Most abstract events will be unrelated.

A useful property of the weak precedence relation is that abstract events that are unrelated by the relation are truly concurrent: there is no causal relation between them. A disadvantage is that this relation is neither asymmetric nor transitive, so it does not determine a partial order. This is a big drawback: a pictorial representation of this relation can be counter-intuitive, especially if abstract and primitive events are displayed simultaneously. Nevertheless, we have chosen to retain this second definition, where only a pair of primitive events need to be related for the abstract events to be related.

A further complication is that two timestamps are required if the abstract events and precedence relation do not form a partial order 6, 7. The two timestamps indicate, roughly, the beginning and end of the abstract event. Thus, using the general definition of an abstract event and the weak precedence relation implies that two timestamps are necessary.

These considerations have led in two directions. One approach is to attempt to preserve the partial-order structure of causality between abstract events. The other is to cope with a precedence relationship that is not a partial order. The first approach suggests that a more restricted definition of "abstract event" would be more useful. Such restricted abstract-event structures are explored in more depth in 1, 11, 13, 32.

While such abstract event structures are attractive because they preserve the partial order, it appears that they are not general enough to be useful. Many apparently intuitive "abstract events" do not satisfy these constraints. A simple example is a scenario where one client successively calls multiple servers to achieve one task. This occurs in a Kerberos 22 environment, for example, when a client first has to contact a ticket-granting server before sending a print request. None of the restricted abstract-event structures identified in the papers referenced above would allow us to model the sequence of RPCs as a single abstract event. Thus, we have recently taken a second approach, which considers very weak structural requirements on abstract events, recognizing that it will not be possible to maintain a partial order on them.

We have investigated the use of weak precedence between convex abstract events 7. In a convex abstract event, any primitive event on a precedence path between two events in the abstract event is also contained in the abstract event. Such convex abstract events generalize a number of abstract-event structures presented in the literature, such as those in 1, 32. For a convex abstract event A, there is no primitive event a, not a constituent of A, such that a depends on the completion of part of A while at the same time the completion of A depends on a. In other words, there is no direct outside interference. Note that any non-convex abstract event A violates the anti-symmetry requirement of a partial order.

Convex abstract events appear promising for a number of reasons. First, they are easier to detect than arbitrary event sets because it is not necessary to filter out interfering events. Second, they are more general than most of the structures guaranteeing the preservation of the partial order for the precedence relation, and hence allow for more flexibility and expressiveness in modeling a distributed execution. Third, because of the absence of interfering events, convex abstract events are considerably easier to display than arbitrary event sets. Finally, convexity also simplifies timestamping. If all convex abstract events are disjoint, a single vector timestamp is sufficient 7. A more detailed discussion of convex abstract events can be found in 29.

6.2. Specification and Recognition

There are several possibilities for specifying abstract events, ranging from mouse-driven pictorial representations to a notation similar to regular expressions. Several algorithms have been devised to recognize abstract events based on such specifications. These include shuffle automata 8, 10 and predecessor automata 19. Both these techniques have theoretical and practical problems (see 4, 34). The languages used to describe abstract events are restrictive, and the recognition algorithms are complicated and slow.

As a first step toward a less restrictive framework, we have investigated the specification of abstract events by concurrent regular expressions 4. These can be transformed to *context-free pomset grammars* (CFPGs) that describe pomset languages. A CFPG is a generalization of a context-free grammar that generates partially ordered multisets *(pomsets)* instead of strings, which are totally ordered multisets. CFPGs can be recognized by *Pomset-LR* (PLR) parsers 5. Although this approach has a firm theoretical foundation and appears promising, it is not yet clear whether this is a practical method for specifying and recognizing abstract events. For example, concurrent regular expressions generate only series-parallel pomsets, a result that appears to be a major drawback since we believe many intuitively meaningful abstract events to be outside this class.

As an alternative to specifying abstract events, we are developing tools to derive convex abstract events automatically, similar to the clustering tools discussed earlier. In particular, we have developed a tool that combines runtime information with a static source analysis, described in 26.

Another tool derives abstract events by applying simple pattern matching ideas 36. The user identifies a given event pattern on the display and asks Poet to identify identical patterns in the event stream. Each occurrence of such a pattern automatically results in the creation of an abstract event. While this tool is more restrictive than the previous one, it provides a simple initial exploratory tool to examine the event stream for repeatedly occurring patterns.

In some target environments, specific classes of abstract events can be determined automatically, and corresponding tools have been developed. ABC++ 3, for example, is a concurrent object-oriented programming language. In this environment, Poet offers the possibility to abstract method invocations, based on the receiving object, the method/class combination, or manual selection with the mouse (including potentially nested invocations of other methods) 35. This tool proved valuable in reducing the apparent visual complexity by "hiding" the interactions via method invocations that have been examined and found correct, allowing a user to concentrate on the unexplored interactions. In PVM, a message-passing library for parallel computing, some communication primitives result in one-to-many and many-to-many pairings, such as broadcasting a message or synchronizing via a barrier. To overcome Poet's limitation of one-to-one event pairings, these primitives are modeled as abstract events, with the one/many send or receive events forming the constituent primitive events 28.

6.3. Displaying Abstract Events

In the case of abstract events that form a partial order, there is, in principle, no reason for the abstract display to be very different from the primitive display. One potential for differences concerns the representation of the relationship between processes and events, in that an abstract event now can involve more than one process. If there were no trace lines in the display, an abstract event could be reduced to a single point and appear almost identical to a primitive event. However, we feel that the "location" of an abstract event in several traces is useful information that should be immediately apparent to the user. For that reason, Poet draws an abstract event as a rectangle placed across a number of trace lines. The manner in which the intersection with a trace is drawn indicates whether the process or cluster represented by the trace participates in the abstract event (see Figure 1).

Convex abstract events do not guarantee a partial-order relationship, so this design is potentially deceiving, since the apparent visual transitivity through abstract events may not exist in the underlying execution. We need to gain more experience with this aspect of abstract-event displays. Since Poet also contains functionality to explicitly highlight predecessors and successors of an event on the display, we hope that this feature can be used to make the actual precedences clearer when requested by the user.

To indicate the resulting reductions in display complexity, let us return to Figure 1 once again. The traced execution generated a total of 3404 primitive events. For a complete display of the execution, 73 windows (or the corresponding number of scroll operations) in the time dimension are necessary. Using both event abstraction and clustering, the complete execution history occupies only 4.5 windows in the time dimension. Obviously, such reductions in the required display space reduce the mental effort required to understand the execution only when the abstract entities represent "meaningful" units.

6.4. User Interface

Poet has been expanded to handle convex abstract events. Great care has been taken to integrate abstract events seamlessly. For all practical purposes, they can be treated just like primitive events: they can be dragged to the left or right edge of the display area to scroll in the time dimension, they can be tagged, will be highlighted as predecessor or successor of a specified event, etc. Unlike clusters, however, it is more difficult to imagine a user-friendly interface to manage the complete event-abstraction hierarchy. Because of the huge number of primitive events and the size of the event-abstraction hierarchies, a straightforward adaptation of the cluster hierarchy interface appears not very promising. Displaying the complete hierarchy, even using scrollbars, immediately raises a new complexity problem: how to find or determine the part of the abstraction tree in which a user is currently interested. Another problem is that abstract events are related by the precedence relation. This relation might also need to appear in such a display, potentially adding unwelcome complexity to the interface.

For the time being, we implemented a simpler interface, based on the main Poet display. A user can manually create and/or modify the abstract-event hierarchy using mouse operations. Poet ensures that these manual modifications do not violate the convexity requirements. The user can also navigate the event-abstraction hierarchy (i.e., change the "debugging focus") by "opening" a displayed abstract event or "closing" it by selecting one of its displayed constituents. Finally, abstract-event hierarchies can be read in from an external file or saved for later reuse.

7. TARGET-SYSTEM INDEPENDENCE

One design goal was to make Poet independent of the specific target system. A tangible proof that we achieved this goal is that to date we use Poet to visualise executions in several different target environments, including ABC++ 3, μC++ 12, Concert/C 39, OSF DCE 33, Hermes 37, PVM 16, and SR 2. These environments differ widely in their programming paradigms (message passing, shared memory) and intended computing platforms (distributed processing, parallel programming). This section explains how we achieved our goal and some of the major design decisions necessary. To understand the remainder of this section, we have to explain the general Poet architecture first.

7.1. Poet Architecture

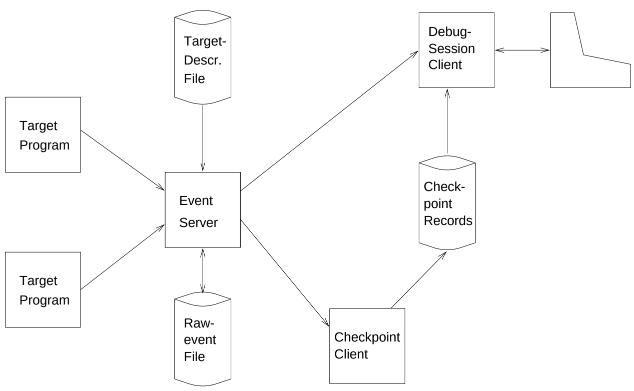

FIGURE 4. The Architecture of Poet

The architecture adopted for Poet is shown in Figure 4. In a minimal configuration, the debugger consists of an event-server process and two clients of the event server: the debug-session process and the checkpoint process.

The debug-session process is responsible for direct interaction with the user. It obtains user input via the keyboard and mouse, and generates appropriate displays in response. Most of the complex algorithms for the debugger are in this process, including those for display scrolling, clustering, and event abstraction.

The event server is responsible for communication with the processes making up the target application. The concurrency required between handling user interactions and handling the target application makes it unreasonable to deal directly with the target application from the debug-session process. As well, a separate process with a primary responsibility for accepting and storing events reduces the probability of blocking on the event stream. Any such blocking would, of course, greatly increase the probe effect in the target program. Thus, event data is received from event-collection "hooks" embedded in the target environment and placed in a disk file, with minor transformation to make it more convenient for later use. Then, clients of the event server (the other two processes) request event data from the event server as required.

The checkpoint process exists solely to improve performance. As discussed previously, vector timestamps are associated with events to determine the partial-order relationship between them. Because timestamps are $O(N)$ in size if N processes exist, storing a timestamp with each event is not feasible. Thus, timestamps must be computed as needed, but the calculation of a timestamp, in principle, requires tracing the execution history of the target application from its beginning to the point at which the specified event occurs. To make calculation of arbitrary timestamps reasonably efficient, the checkpoint process simply timestamps all available event data and periodically writes out a checkpoint containing the internal state of the timestamp algorithm. The debug-session process can then find a checkpoint preceding an event that is to be timestamped and need only run the timestamp algorithm forward from that checkpoint. Communication between the debug-session and checkpoint processes takes place only through the file of checkpoint data, whereas communication between the event server and its clients involves explicit message passing.

The design, consisting of a single server and multiple clients, allows easy addition of further processes as necessary. For example, the debug-session process can be instantiated several times to provide multiple simultaneous views of the execution. Clients have also been written to make "annotations" to the raw event data, such as marking events participating in data races, and converting run-time information such as a thread address into a more meaningful symbolic form by debug-time use of the symbol tables stored in object files 35. A "dump and load" client is also available: it dumps an event data file in text form for use either by other tools or by a loader which can feed a new instance of the event server. This allows easy migration of event data files to new versions of Poet or to versions compiled for different architectures.

7.2. Target-System Descriptions

Because Poet is intended to deal with the problems of debugging that are found in distributed and parallel environments, it has the potential for use with a variety of environments. Details unique to a particular environment can be handled by an address-space debugger. Thus, we strove to make Poet independent of the target environment, starting with the very first design. As Poet has developed (and we experimented with and learned from working with multiple target environments), the target-system independence has been increased, so that now all the main Poet executables contain no code specific to a target environment, but rely entirely on data in an external configuration file. Of course, the actual collection of events cannot be performed independently of the target environment. Each target environment must be instrumented appropriately to provide the event stream required by Poet. Different approaches have been taken for different target environments. The PVM built-in tracing facility was used to collect relevant events and a pre-processor translates event data to a format understood by Poet. OSF DCE applications are instrumented automatically by a modified version of the IDL compiler that generates client and server stubs, and for Hermes applications we added a tracing facility to the runtime system. All instrumentation produces a standard event stream that is then collected and processed by Poet, as shown in Figure 4.

The key to achieving Poet's target-environment independence is its very general model of a distributed execution, based on *traces* and *events* as discussed before. A usual interpretation is that each trace represents a process or thread, but other interpretations are also possible. For example, in the μC++ environment, some traces represent execution threads, but others represent monitors, semaphores, and other "passive" entities. Anything with sequential behaviour can be represented by a trace, and interactions between active and passive entities can be represented in the same way that the sending of a message between processes is represented. Thus, for example, in μC++, monitor entry and return are shown as interactions between the monitor and the thread entering and leaving.

Most of the information Poet requires concerning a target environment relates directly to the issues just discussed. That is, the set of possible event types must be specified, as well as their representation on the display, which event types pair with which other types, whether pairings are synchronous or asynchronous, what textual name should be used for the event, and so on. All of this information is placed in an event-description table, which forms the largest part of the target-description file. The following simplified extract from a hypothetical event-description table describes the four events involved in an RPC interaction. Each entry begins with the identity of the event being described; the remaining fields will be discussed below.

```
CALL,      -1,    F_CIRCLE, DASH,  "call";
CALL_RECV, CALL, -F_CIRCLE, SOLID, "call receive";
REPLY,      -1,    O_CIRCLE, SOLID, "reply";
REPLY_RECV,REPLY,-O_CIRCLE, SOLID, "reply receive";
```

Event pairing is one of the most important specifications. Essentially, three things must be specified: which event types pair with each other, which events contain partner data, and whether pairings are synchronous or asynchronous. The first two are specified by giving a partner-event type: if that type is zero, there is no partner; if that type is -1, there is a partner but this event type does not contain partner data; otherwise, that type is a possible partner and this event type contains partner data. For an event type that contains partner data and has multiple partner types, multiple entries are included in the table, one for each partner type. The partner event is the second field of each entry, so in the above example CALL and CALL_RECV are indicated as an event pair, with only CALL_RECV containing partner data. Asynchronous events are identified by attaching a flag to the sending event of the pair. (All events in the above example are synchronous.)

It is not reasonable to describe here all of the other information that can be provided in the event-description table; the following includes some of the more important information. A character-string name is provided, to use in identifying the event on the display, e.g., CALL will be identified as "call". A symbol type is provided, indicating whether to represent the event by a filled or unfilled square or circle, e.g., the call/receive pair of events will be shown as filled circles and the reply/reply-receive pair of events will be shown as unfilled circles. (The minus signs indicate the direction in which arrows should be drawn between events.) Finally, one field indicates what kind of line should be drawn following this event type (solid, dashed, or none). The interpretation of the three line types is dependent on the environment. The most common interpretation is that a solid line means the process or thread is active, a dashed line means it is blocked, and no line means it has not yet started or has terminated. In the above example, CALL blocks the thread and so has linetype DASH, whereas all other line types are SOLID. (Of course, the line types can only reflect information available in the set of events. Unless a receive-block event is added to the above example, blocking at the receiving side of a call will not be reflected.) Because such an interpretation is not embedded in the description technique, a quite different interpretation can be used if appropriate. As described previously, in μC++, when a thread is executing in a monitor, the trace line for the thread is not drawn. Combined with the connecting lines for the event pairs representing monitor entry and exit, this creates a visual effect suggesting that the thread of control leaves its trace line, enters the monitor, and finally returns to its own trace line.

As indicated in Figure 4, the target-description file is read by only the event server. Clients can then obtain any information they need about the target environment by sending appropriate requests to the event server.

Other data in the target-description file includes a few simple things such as the title to use in the event-display window and windows created for the execution of target programs. As well, a code value is specified that identifies the target environment. Then, when a target program provides event data or an old file of event data is accessed, it is possible to determine the target environment based on the code value. In general, it is reasonable to configure Poet with several target-description files specified. Then, as soon as event data becomes available, the appropriate target-description file is selected and loaded.

Some data about the target environment is not included in the target-description file, notably the field lengths used in the event stream to identify streams and traces, and the length of text information in the event stream. Such data is included in a "header" event that is sent first on each event stream. A major reason for placing this information in the event stream itself is that the consequences of an incorrect value are complete misinterpretation of the event stream. That is, if the target-program instrumentation and the event server differ in their view of the length of a certain field, then the event server will lose track of the boundaries between events and be unable to extract anything useful from the event stream. On the other hand, if the target-description file fails to describe a particular event type or describes it incorrectly, events of that type will not be handled properly, but following events can be processed.

Thus, an attempt has been made to encode a description of the target environment in a few variables and the event-description table. In practice, it is not possible to handle everything in this way. One example is the "default clustering" capability, in which a set of clusters is built corresponding to features of the target environment. Such default clustering only makes sense for some environments. For the two environments in which it has been implemented (Hermes and ABC++), the algorithms are complicated and also quite different. It might be possible to reduce them to a single, table-driven algorithm, but there would be little reason to believe that the needs of some third target environment could be handled by the algorithm. A second example is the name translation performed in ABC++. In that environment, meaningful trace names cannot be reasonably obtained by the target-program instrumentation. Instead, internal representations are obtained which must be translated by making reference to object-module symbol tables. Again, generalizing this across target environments appears unreasonable.

Such problems have been dealt with through the client-server structure of Poet. In addition to the clients discussed in the previous section, which are independent of the target environment, it is possible to include clients that belong to a specific target environment. Thus, default clustering is implemented by a Hermes client and an ABC++ client. Other environments do not provide such a client and default clustering is then not available in those environments. Similarly, name translation is provided by a client specific to ABC++.

These two problems also illustrate two forms of using target-specific clients. In the default-clustering case, there is a general problem that requires a target-specific solution, so code in a target-independent part of Poet will attempt to access the target-specific client. In the name-translation case, even the existence of the problem is target-specific. Thus, the target-description file for ABC++ identifies the name-translation client and specifies that it should be started automatically when the ABC++ target-description file is loaded. The client then performs name translation, using standard event-server requests to fetch and store event-server data, but the other clients are essentially unaware of its existence.

8. PERFORMANCE

Visualising parallel and distributed executions can be very resource intensive. While it is relatively simple to write graphical user interfaces to depict the execution of toy applications, building tools that work efficiently and with adequate response time for large executions, i.e., executions with many process and/or many events, is a major challenge. The first publicly available version of XPVM, version 1.0.3, for example, was based on the Tcl/Tk toolkit to rapidly prototype a visualisation tool. As reported in 23, however, the response time for executions with many events was abysmally long. Consequently, the improved version 1.1.0 re-implemented many of the core drawing routines in C to speed up execution. Because Poet is intended to be useful when there are many processes and many events, both functional behaviour and performance were considered carefully during the design. This section briefly describes these issues and compares Poet's performance to XPVM.

8.1. Designing for Performance

A potential problem with the present debugger is that the complete set of events must be stored in a disk file, which does not allow it to be used with programs that generate extremely large sets of events. In practice, at approximately 40 bytes per event, it is possible to handle quite large event histories before disk space becomes a problem. Also, all other visualisation tools we reviewed store the complete execution histories, frequently employing more bytes per event than Poet.

The aspect of performance we addressed is maintaining reasonable response time when dealing with large execution histories. The use of a checkpoint file to avoid extremely high timestamp-computation costs has already been described. In addition, two different caches of timestamped events are maintained. One is simply the set of events appearing in the current display. The other is the set of events that have recently been processed by the timestamp algorithm. Several such sets of events are maintained, along with the corresponding timestamp-algorithm state, so that the timestamper can be reset to a recent state as well as a state obtained from the checkpoint file.

A cache of "raw" (non-timestamped) events is also maintained in each client program. The obvious technique is used, requesting a block of events from the event server rather than a single event, and then saving the block for possible use in satisfying future requests. At present, one block of raw events is cached for each trace.

8.2. Comparison with XPVM

Most of the performance enhancements within Poet are obtained by some form of caching. To quantify the performance and to put the achieved results into perspective, we compared Poet to XPVM 1.1.0. We measured the CPU time consumed when visualising the execution of a simple but heavily communicating parallel application, `cholesky`. The application consists of one driver task, `cholhost`, and n node tasks, `cholnode`, to perform column Cholesky factorization of a symmetric matrix partitioned by vertical strips. We traced ten different executions of this application, varying the matrix dimension from 10 to 100 in steps of 10. The parallel virtual machine consists of 5 IBM RS6000/25T workstations, connected by 10 Mbps Ethernet. For each execution, the driver starts up one `cholnode` task on each workstation and distributes the matrix rows in a round-robin fashion. The traced executions generate from 490 to 3,738 events, exchange from 132 to 1,212 messages, and execute in less than 4 seconds wall-clock time for a matrix of dimension 100.

To avoid the dependence of the measurements on variations of the execution time (and therefore the state of the parallel virtual machine), the event traces are stored in a trace file. These trace files, in turn, drive the visualisations in both XPVM and Poet. The CPU times reported here were obtained by starting up the respective visualisation tool, building the initial display from the trace files, and quitting the visualiser. Each entry in the following graph represents the average value over three identical runs.

XPVM provides a set of different views, but for comparison purposes we restrict ourselves to the space-time view, XPVM's equivalent of a space-time diagram. Despite the differences in the display-building principles, the resulting visualisations are fairly similar. While this experiment does not exercise all options available with each tool, it is a fairly comprehensive and intense task: all events stored in the trace file must be processed, a complex display must be built, and eventually, for longer execution histories, the display must be scrolled. In addition, we set a number of Poet options to increase similarity in both appearance and functionality with XPVM.

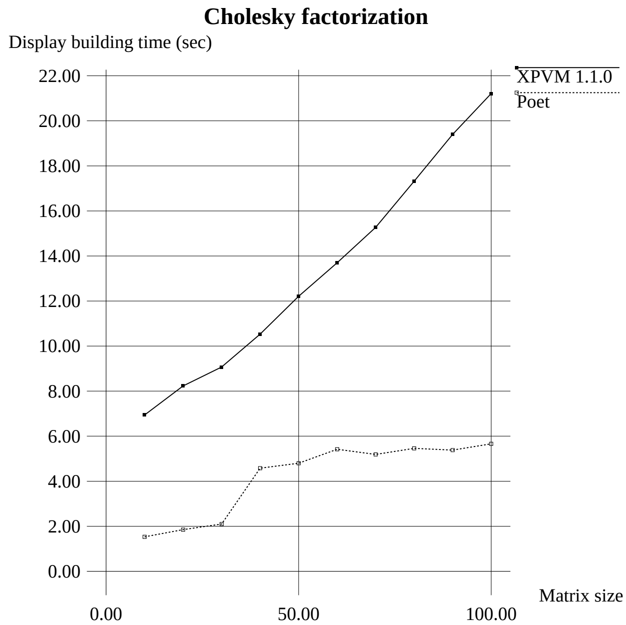

FIGURE 5. CPU time consumed

Figure 5 plots the CPU time consumed by the two tools. Comparing the reported times to the elapsed execution times of a few seconds shows that visualisation is an expensive operation. The CPU time is about an order of magnitude longer than the (wall-clock) execution time. This strongly supports designs of event visualisation tools that separate event collection and presentation. Poet is designed in such a way. Only the event collection component of Poet needs to execute on a machine that is part of PVM's virtual machine. The event information can then be forwarded via TCP connections to a separate machine, where the resource-intensive event processing and visualisation takes place. XPVM, on the other hand, does not support such a split operation. XPVM is itself a PVM task and has to execute on one of the machines comprising the virtual machine. When running our application under XPVM, with `cholhost` and one instance of `cholnode` competing for the CPU with XPVM, we noticed a considerable decrease in the execution speed of the target application.

Poet is significantly faster than even the improved version of XPVM, building the initial display in a fraction of the time. Additionally, while the CPU time increases monotonically for larger event sets for XPVM, Poet reaches a plateau from approximately a problem size of 40. Starting from this problem size, the execution history fills more than one window, so both tools fill the initial drawing canvas and have to employ scrolling to fit the end of the execution history into the window. Scrolling is implemented differently in the two tools. Once Poet recognizes that the event history extends beyond the current window size, it locates the end of the display and scrolls directly there. (However, Poet still needs to process any events that might be skipped, to timestamp later events correctly, for example.) XPVM scrolls the display piece by piece, just enough each time to provide room for the new information to be displayed. This results not only in more scrolling operations, but also draws parts of the execution history that will be scrolled off the display subsequently.

9. CONCLUSIONS

The facilities included in Poet sketched above can be summarized briefly as an event display with intelligent scrolling plus general, hierarchical clustering and event-abstraction facilities with support for automatic derivation of abstractions. Other useful facilities have recently been implemented in prototype form. One of the most important of these is the ability to link with an "address-space" debugger so that conventional debugging facilities are available in conjunction with the event display. (By an address-space debugger, we mean one that can be used to inspect and affect one or more threads of control sharing a single virtual address space.) We have recently modified Poet for use with the AIX Workbench 20 ("encapsulated it," in Workbench terms), and established communication with an encapsulated address-space debugger (dbx). Another important addition is a deterministic replay facility, allowing an execution to be repeated with the partial order of events constrained to be the same as in the initial execution. This extension, including simple control of distributed breakpoints, is described in 40.

Poet was designed to be largely independent of the target environment, obtaining target-system specific information at debug time from a configuration file. As a result, the executable is completely independent of the target environment. Poet has successfully been adapted to a number of different target environments, such as ABC++ 3, μC++ 12, OSF DCE 33, PVM 16, and the debugger itself. (That is, one instance of the debugger can be used to examine the client-server interactions in another instance of the debugger.) The differences among these environments are substantial, which suggests that the debugger can indeed be adapted to a wide variety of target environments by changing only an event-description file. In particular, although the above description refers to "processes" on the display, the trace lines on the display can also be used to represent other objects, for example, threads within a process in an OSF DCE environment, or monitors and semaphores in the μC++ environment. Work is also under way to adapt Poet to additional target environments, most notably Java 17.

Our debugger is at a usable stage, in use in a number of target environments, and helpful in testing and validating our ideas in distributed debugging. The clustering features are used widely, and are very helpful in reducing the apparent complexity of the display: hiding a number of processes inside a cluster not only reduces the number of trace lines, but also hides events internal to the cluster, which tends to reduce the "length" of the display in the time dimension.

Completion of our prototype facilities for event abstraction is an important watershed. Trying to get any sort of "feel" for abstract events present in real executions is nearly impossible without some automatic support. We are now in a position to gain more experience and to build on facilities that might, in future, permit more advanced automatic use of event abstraction, including formal specifications and automatic recognition of abstract events, as well as the more general use of abstract events as a basis for distributed breakpoints.

Generally, there appear to be many open problems relating to the specification and recognition of abstract events. We feel we have made some progress in our attempt to link formal specification techniques based on partial-order models and pomset grammars to the practical difficulties in recognizing and displaying abstract events. However, for the moment, the links are more tantalizing than real. In the future, we will be exploring the use of Poet for visualising distributed executions in support of application management and performance monitoring, including dealing reasonably with very large execution histories.

The main contribution of our work is the blending of firm theoretical foundations with practical tools intended for real applications. As our research evolves, we expect to continue to exploit the synergy we have achieved between theory and practice.

Acknowledgments

The work described here began in the Shoshin project at the University of Waterloo and has benefited from numerous discussions with members of the Centre for Advanced Studies, IBM Toronto Lab. The work was supported by NSERC, by the Information Technology Research Centre, and by IBM.

REFERENCES

[1] M. Ahuja and S. Mishra. Units of computation in fault-tolerant distributed systems. In *Proc. 14th Int. Conf. on Distr. Comp. Systems*, pages 626–633, Poznan, Poland, June 1994.

[2] G.R. Andrews et al. An overview of the SR language and implementation. *ACM Trans. on Programming Languages and Systems*, 10(1):51–86, Jan. 1988.

[3] E. Arjomandi et al. ABC++: Concurrency by inheritance in C++. *IBM Sys. Journal*, 34(1):120–137, 1995.

[4] T. Basten. Hierarchical event-based behavioral abstraction in interactive distributed debugging: A theoretical approach. M.Sc. thesis, Eindhoven University of Technology, The Netherlands, Aug. 1993.

[5] T. Basten. Parsing partially ordered multisets. To appear in *Int. Journal of Found. of Comp. Science*, 1997.

[6] T. Basten, T. Kunz, J.P. Black, M.H. Coffin, and D.J. Taylor. Time and the order of abstract events in distributed computations. Computing Science Note 94/06, Eindhoven University of Technology, Feb. 1994.

[7] T. Basten, T. Kunz, J.P. Black, M.H. Coffin, and D.J. Taylor. Vector time and causality among abstract events in distributed computations. To appear in *Distributed Computing*.

[8] P.C. Bates. Shuffle automata: A formal model for behavior recognition in distributed systems. COINS Tech. Rep. 87-27, U. of Massachusetts, Jan. 1987.

[9] P.C. Bates and J.C. Wileden. An approach to high–level debugging of distributed systems. In *Proc. of the ACM SIGSOFT/SIGPLAN Soft. Eng. Symp. on High–Level Debugging*, pages 107–111, Pacific Grove, CA, March 1983.

[10] P.C. Bates. Debugging heterogeneous distributed systems using event-based models of behavior. *ACM Trans. on Comp. Systems*, 13(1):1–31, Feb. 1995.

[11] E. Best and B. Randell. A formal model of atomicity in asynchronous systems. *Acta Informatica*, 16:93–124.

[12] P. A. Buhr and R. A. Stroobosscher. The μSystem: Providing light-weight concurrency on shared-memory multiprocessor computers running Unix. *Software — Practice and Experience*, 20(9):929–963, Sept. 1991.

[13] W.-H. Cheung. *Process and Event Abstraction for Debugging Distributed Programs*. PhD thesis, Dept. of Computer Science, University of Waterloo, Oct. 1989.

[14] S.G. Eick and A. Ward. An interactive visualization for message sequence charts. In *Proc. of the Fourth Workshop on Program Comprehension*, pages 2–8, Berlin, Germany, March 1996. ISBN 0-8186-7283-8.

[15] C.J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, Aug. 1991.

[16] Al Geist et al. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994. ISBN 0-262-57108-0.

[17] J. Gosling and H. McGilton. *The Java Language Environment*. Javasoft, CA, May 1996. `http://www.javasoft.com/doc/language_environment/`.

[18] A.A. Hough and J.E. Cuny. Initial experiences with a pattern–oriented parallel debugger. In *Proc. of the ACM SIGPLAN/SIGOPS Workshop on Par. and Distr. Debugging*, pages 195–205, Madison, WI, May 1988.

[19] W. Hseush and G.E. Kaiser. Modelling concurrency in parallel debugging. In *Proc. 2nd ACM/SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 11–20, Seattle, WA, March 1990.

[20] IBM. *IBM AIX SDE Workbench/6000 User's Guide and Reference*. IBM, September 1992. SC09–1453–01.

[21] M. Khouzam and T. Kunz. Single stepping in event-visualization tools. In *Proc. of the 1996 CAS Conf.*, pages 103–114, Toronto, Canada, Nov. 1996.

[22] J.A. Kohl. The Kerberos Network Authentication Service (V5). Internet Draft, Internet Activities Board, March 1993.

[23] J.A. Kohl and G.A. Geist. The PVM 3.4 tracing facility and XPVM 1.1. Technical report, Comp. Science & Math. Division, Oak Ridge Nat. Lab., TN, USA, 1995.

[24] D. Kranzlmüller, S. Grabner, and J. Volkert. Event graph visualization for debugging large applications. In *Proc. of SPDT 96: SIGMETRICS Symp. on Parallel and Distributed Tools*, pages 108–116, Philadelphia, PA, USA, May 1996.

[25] J. Kundu and J.E. Cuny. A scalable, visual interface for debugging with event-based behavioral abstraction. In *Frontiers '95. The Fifth Symp. on the Frontiers of Massively Parallel Computation*, pages 472–479.

[26] T. Kunz. Reverse engineering distributed applications: An event abstraction tool. *Int. Journal of Soft. Eng. and Knowledge Eng.*, 4(3):303–323, Sept. 1994.

[27] T. Kunz and J.P. Black. Using automatic process clustering for design recovery and distributed debugging. *IEEE Trans. on Soft. Eng.*, 21(6):515–527, June 1995.

[28] T. Kunz and D.J. Taylor. Visualizing PVM executions. In *Proc. of the 3rd PVM User's Group Meeting*, Pittsburgh, PA, USA, May 1995. 13 pages, URL: `http://www.cs.cmu.edu/Web/Groups/pvmug95.html`.

[29] T. Kunz. High-level views of distributed executions: Convex abstract events. *Automated Soft. Eng.*, 4(2):179–197, April 1997.

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

[31] L. Lamport. On interprocess communication. *Distributed Computing*, 1:77–85, 1986.

[32] M.J. Manthey. Hierarchy in sequential and concurrent systems. In L.H. Jamieson et al, editors, *The Characteristics of Parallel Algorithms*, pages 139–164. The MIT Press, Cambridge, MA, 1987.

[33] Open Software Foundation. *Introduction to OSF/DCE*. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[34] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7:149–174, Sept. 1994.

[35] I.R. Seelemann. Application of event-based debugging techniques to object-oriented executions. M.Math. thesis, University of Waterloo, June 1995.

[36] M. Seuren. Design and implementation of an automatic event abstraction tool. M.Sc. thesis, Eindhoven University of Technology, The Netherlands, Aug. 1996.

[37] R.E. Strom et al. *Hermes: A Language for Distributed Computing*. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[38] D.J. Taylor. The use of process clustering in distributed–system event displays. In *Proc. of the 1993 CAS Conf.*, pages 505–512, Toronto, Canada, Oct. 1993.

[39] S. Yemini et al. CONCERT: A high-level-language approach to heterogeneous distributed systems. In *Proc. 9th Int. Conf. on Distr. Comp. Systems*, pages 162–171, June 1989.

[40] Y.M. Yong and D.J. Taylor. Performing replay in an OSF DCE environment. In *Proc. of the 1995 CAS Conf.*, pages 52–62, Toronto, Canada, Nov. 1995.

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