

Linking Specification, Abstraction, and Debugging

J. P. Black, M. H. Coffin, D. J. Taylor

Dept. of Computer Science

University of Waterloo

Canada

T. Kunz

Dept. of Computer Science

Technical University of Darmstadt

Germany

A. A. Basten

Dept. of Mathematics and Computing Science

Eindhoven University of Technology

The Netherlands

November 4, 1993

Abstract

Designing and implementing a visual debugger for distributed programs is a significant challenge. Data about executing programs must be gathered and collated without unduly modifying the behaviour of the program under study. The data must be displayed in ways that are understandable to the user. One aspect of this problem is that human users are easily overwhelmed by huge amounts of detail, so the debugger must be capable of providing a variety of displays, ranging from high-level views of the entire execution to low-level views of primitive events. If the debugger is to be used interactively, all this must happen fairly quickly.

This paper discusses the basic theoretical issues involved in the design of visual debuggers for distributed programs and then describes the practical issues involved in the implementation of such debuggers, including a brief description of a prototype implementation.

1 Introduction

One of the barriers to distributed-application development is the lack of good debugging tools to reduce the development time and effort. With distribution come concurrency, scale, and complexity. Instead of a single process, the developer is faced with several. Instead of an execution being a totally ordered set of events, it is only a partially ordered set. Instead of the usual deterministic execution of sequential programs, most distributed applications are inherently non-deterministic. Instead of a single process on a single machine, the application involves a potentially large number of machines, processes, threads, and events. An important approach to coping with this complexity is to apply abstraction in the process and event domains, so that the developer can view the execution in terms of larger units than threads and primitive events.

A distributed debugger should scale well to large numbers of events and processes. This implies that a single level of abstraction is not, in general, 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. By starting with a high-level abstraction, 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.

Unlike sequential applications, simply re-executing a distributed application with the same input data does not guarantee identical program behaviour, because of inherent non-determinism in the program execution. A replay mechanism [15] ensures deterministic repetition of program execution. Such a mechanism also allows minimization of the probe effect: collecting and presenting behavioural data influences the (non-deterministic) program execution, potentially masking errors. Separating event collection from behaviour visualization allows this effect to be minimized. In a first run, an application is executed and only the information essential for a deterministic replay is collected, minimizing the perturbation. Later replays will then be guaranteed to execute identically, even in the presence of massive user intervention, such as stopping processes or modifying the state space of individual processes (assuming, of course, that such modifications

still admit the same partial order of event execution). Separating event collection from presentation has the additional advantage that it provides a good opportunity to develop a debugger that is independent of the target system. While the event-collection mechanism clearly must be adapted to each new target system, the debugger itself should be kept independent to conserve the effort invested.

Typically, the lowest level of observable behaviour for distributed debugging is process creation, termination, and interprocess communication. However, to really debug an application, more detailed information about processes might be necessary. We therefore propose to incorporate adequate address-space debuggers for each target environment into the debugger. In the general case of multiple threads executing in a single shared address space, the “sequential” debugger is responsible for such tasks as stack-frame and variable display, breakpoint insertion, and variable modification for all the threads in the address space. How to handle the data and control flow between our distributed debugger and the address-space debuggers is one of the problems currently under investigation.

Presenting abstract views of behaviour requires some transformation of the primitive behaviour captured by debugging probes in the target system under development. This implies a need for both a formal approach to process and event abstraction, and for specifications of the abstract processes and events to be presented. The need for such specifications related to the *actual* behaviour in turn raises the question of specifications for the *expected* behaviour: much work has already been done on hierarchical specification models for distributed and concurrent systems.

While we have not ignored issues of replay or conventional debugging technology, we have chosen to concentrate on questions related to abstraction, and on developing a prototype which is effective in displaying “large” executions consisting of many processes and events. Section 2 describes the theoretical basis of our approach to distributed debugging. Then, Section 3 describes the prototype we have developed, which implements many of the theoretical concepts. Finally Section 4 describes some open problems and Section 5 provides a summary and some concluding remarks.

2 Theoretical Foundations

For the sake of simplicity and generality, our model of computation makes few assumptions about the programs being debugged.

Following Lamport [14], we view a process as an ordered sequence of atomic events. An event can be thought of as the instant at which some computation is completed. Our model of computation addresses only the events and their relation to each other. The computation itself is not addressed by the model.

Our model of computation does not assume any bound on communication latency. Thus the results are applicable to systems with or without such bounds. Our model allows lost messages, since a lost message is not essentially different from a message with a very large latency.

Processes can be executed on different computers that may be physically separate. Since clock-synchronization algorithms depend on bounded communication latency [14], which we do not assume, we do not assume the existence of a global clock either. Our model includes both synchronous and asynchronous communication; a message is modeled as a send event in one process and a receive event in the other. Processes can be created and destroyed during execution, and so their number is not known *a priori*.

2.1 Ordering Events

The order in which events happen is important to debugging. Not only do we want to know the order in which events occur in a particular execution, but we would like to specify allowable event orders and have the debugger notify us if the specification is violated.

Two models of event ordering are in common use. The *interleaving* model begins with the observation that, in any particular execution, events are executed in some particular temporal order. (There is a theoretical possibility that two events could happen simultaneously; this is not only unlikely, but also indistinguishable from happening at slightly different times.) However, since concurrency usually produces nondeterministic behaviour, it is usually not possible to specify a single order of events for a program. Thus an ordering

specification in the interleaving model consists of a set of potential interleavings.

The *partial-order* model does not assume any total order; instead Lamport's precedence relation ("happens before") is used [14]. For asynchronous systems, the precedes relation is defined to be the transitive closure of two primitive orders: the order between two events in a single process and the order between a send event and the corresponding receive event. To cope with synchronous messages, additional primitive orders have to be introduced. (See also the discussion in [19, p. 17].)

More formally, an event e precedes an event g ($e \prec g$) if and only if one of the following is true:

- e and g occur in the same process and e occurs before g ,
- e is an asynchronous send event and g is the corresponding receive event,
- e is a synchronous event, f is the corresponding partner event, and $f \prec g$,
- g is a synchronous event, f is the corresponding partner event, and $e \prec f$, or
- there exists an event f , distinct from e and g , such that $e \prec f$ and $f \prec g$.

Notice that a synchronous send event does not precede the corresponding receive event.

For our purposes, this partial-order model has several advantages. First, saying that there is a total ordering of events implies, contrary to our model, that a global clock exists. This is not a serious theoretical problem, but it is inelegant.

More seriously, a global order on events contains both more and less information than is desirable. Since the actual execution of processes is concurrent, many relations in a total order are accidental and irrelevant. Two concurrent events can happen in either order; their happening in some particular order during a particular execution cannot affect the outcome of the computation. In addition, information about which events are truly concurrent is lost in a total order; only a single instance from a set of possible interleavings remains.

A concrete example of this is the use of event orders to specify expected behaviour: a set of allowed orderings on events is given; the actual order is compared with those expected, and discrepancies are reported to the user. Suppose the order is specified as a set of allowable interleavings, and suppose that a particular computation satisfies the specification. Two kinds of ordering errors will not be detected. First, the program under consideration may erroneously disallow certain interleavings that should be possible; for example, excessive serialization may be taking place. Second, the program may satisfy the specification only because of fortuitous timing; errors involving unlikely race conditions will probably not be found.

Both kinds of errors described above would be detected by a debugger based on partial orders. Excessive serialization would be detected if events specified to be concurrent were not. Timing errors have no effect since the wall-clock time of events is ignored in constructing the partial order.

A drawback of 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. Timestamps [10, 16, 19] provide a way to quickly determine the order of two events in a partial order. The desired property is that event e precedes 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 $T_h < T_g$.

Our timestamping algorithm is based on one by Fidge [10], modified for synchronous communication by Cheung [9]. The timestamp of an event is a vector of integers, one component for each process in the computation. (For efficiency, our prototype implements timestamps with fixed-length arrays; other, more flexible implementations are also possible [10].) The timestamp of the initial event in process i is a vector consisting of all zeros. Let C_i and C_j be the local timestamps being maintained by processes i and j , and T_e and T_f the timestamps of events e and f occurring in i and j respectively. For a timestamp T , let T^{+i} be T with its i th component incremented by one. Other timestamps are then calculated as follows (*max* is element-wise maximum).

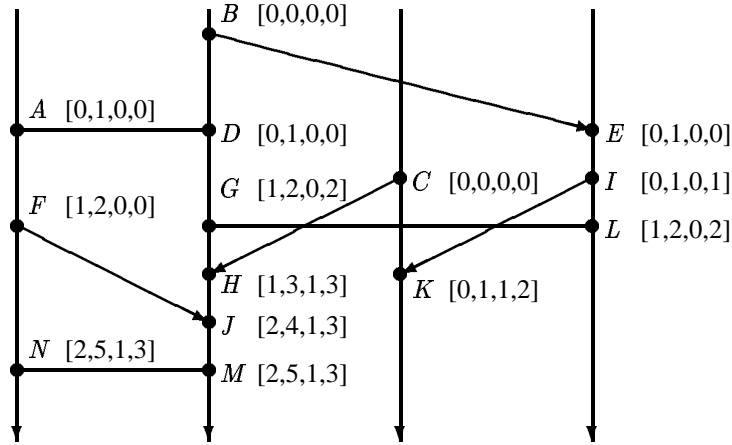


Figure 1: Timestamps of Events

1. For a local event e in process i , $T_e \leftarrow C_i$; $C_i \leftarrow C_i^{+i}$.
2. For an asynchronous send event e in process i , $T_e \leftarrow C_i$; append T_e to the message sent; $C_i \leftarrow C_i^{+i}$.
3. For an asynchronous receive event f in process j , corresponding to send event e in i , $T_f \leftarrow C_j \leftarrow \max(T_e^{+i}, C_j)$; $C_j \leftarrow C_j^{+j}$.
4. For a synchronous event e in i , corresponding to f in j , send C_i to j and receive C_j from j ; $T_e \leftarrow C_i \leftarrow \max(C_i, C_j)$; $C_i \leftarrow (C_i^{+j})^{+i}$. (Each event thus receives the same timestamp.)

It can be shown [9] that element-wise comparison of timestamps has the desired property: e in i precedes f in j if and only if $T_e[i] < T_f[j]$. Figure 1 gives an example of the timestamps associated with the events in a simple execution involving four processes.

The timestamp algorithm above is expressed as if each message carried with it a timestamp vector of size N . This is necessary if the vector timestamps are needed by some on-line algorithm which is part of the application itself. In our case, however, this is completely unnecessary: the timestamps are only required in the debugger (not the application), and only when the execution is being visualized. This means that 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 which observed the message exchange. Typically, this involves adding at most one integer (a local event counter) to each message, if no other unique message identifier is available. At debug time, the timestamp code simulates the actual message exchanges as recorded in order to calculate full vector timestamps. As an aside, it is easy to prove that a vector of size N is sufficient and sometimes necessary in order to correctly represent the partial order [19, Ch. 2] [8].

There have been a number of variations on vector timestamps with subtly different properties [2, 8, 10, 16]; the one chosen here is convenient for our prototype because it avoids collapsing synchronous pairs of events into a single event, and because the timestamp test involves comparison of only one pair of integers. It is not the only variation with these properties.

2.2 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. Two problems must be addressed when clustering processes:

- how to represent a process cluster, and
- what processes and/or subclusters should be combined.

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 some 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 [9] and [19]. Our prototype takes an even simpler approach [20]: as concurrency is encountered during display drawing, the number of trace lines associated with a cluster is increased as necessary.

Automatic clustering. 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 high enough to make any manual clustering both tedious and error-prone. Our group has therefore worked on tools to derive process-cluster hierarchies automatically. Two approaches were compared, one clustering processes solely on the basis of their interprocess communication frequency during run-time 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 process clustering rules that combine this semantic data with IPC information. Experimental results with a number of Hermes [18] applications showed the latter approach to result in better clusters [13].

2.3 Event Abstraction

Event abstraction allows groups of basic events to be combined into “abstract events.” As with process 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, e.g., an RPC could be recognized and displayed as a single event.

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, i.e., 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 com-

putation. Unfortunately, with this definition, abstract events do not have the same properties as primitive events. Elementary 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.

There are two obvious ways to define a relation over arbitrary 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 first definition has several desirable properties, including asymmetry and transitivity. Thus, this relation is a partial order. Unfortunately, in most cases, the relation it defines is too small to be useful: most abstract events will be unrelated (hence formally “concurrent”).

An advantage of the second definition 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 the abstract events and primitive events are displayed simultaneously. In general, two timestamps are required to correctly determine the precedence between a pair of arbitrary sets of events. The two timestamps indicate, roughly, the beginning and end of the abstract event. Worse still, one of these must be calculated by processing the event traces in the reverse direction [2]. 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.

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. There are a number of possibilities, called *transactions*, *central event abstractions*, *complete precedence abstractions*, and *contractions*.

First, define an *input event* (respectively *output event*) to be a member of an abstract event which has an immediate predecessor (successor) outside the abstract event.

Transactions have a single input event and a single output event. They correspond to the normal notion of atomic transactions. They are a subclass of central event abstractions [19], which contain an event which is preceded by all input events and followed by all output events. This guarantees transitivity, and the timestamp of this central event can be used for the timestamp of the abstract event.

Complete precedence abstractions [19] have all input events preceding all output events. This also guarantees transitivity, and an appropriate timestamp is the element-wise minimum of the timestamps of the output events.

Contractions are the most general structure that has the same partial-order structure as primitive events [6, 9]. They have the property that there is a path inside the contraction for each pair of preceding and succeeding events outside it. Unfortunately, the definition of a contraction and the rules for interconnection of such abstract events are both restrictive and complicated. In addition, timestamping does not work properly with contractions. Timestamps can be defined so that $e \prec f$ implies that $T_e < T_f$. However, what is usually needed is the converse, which unfortunately does not hold.

While these possibilities are attractive because they have fairly strong structural properties, it appears that they are not general enough to be useful: many apparently intuitive “abstract events” do not satisfy these constraints. Thus, we have recently taken a second approach, which considers even weaker structural requirements on abstract events, recognizing that it will not be possible to maintain a partial order on them.

We are currently investigating combining the use of this definition of precedence between abstract events with “convex” event sets [2, 3]: these sets are disjoint and do not interfere with each other. More formally,

any path between two events in the convex set contains only other events in the set. Convexity also simplifies timestamping. For this special case, it is possible not only to calculate the two timestamps required for abstract-event precedence testing in a single pass over the event traces, but it is also possible to restrict timestamp-element comparisons to processes participating in the relevant convex event. They also appear to be easier to recognize and display than arbitrary abstract event sets would be. In addition, if all convex abstract events are disjoint, a single vector timestamp is sufficient [3].

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 [4, 5] and predecessor automata [12]. Both these techniques have theoretical and practical problems. The languages used to describe abstract events are restrictive, and the recognition algorithms are complicated and slow.

As a first step towards a less restrictive framework, our group is investigating the specification of abstract events by concurrent regular expressions. These can be transformed to *context-free pomset grammars* (CFPGs) which describe pomset languages [2]. A CFPG is a generalization of a context-free grammar that generates partially-ordered multisets (*pomsets*) instead of strings (totally-ordered multisets). CFPGs can be recognized by *Pomset-LR* (PLR) parsers.

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 [11], which appears to be a major drawback since we believe many intuitively meaningful abstract events to be outside this class.

Displaying Events. In the case of abstract events which 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. Our current design draws an abstract event as a rectangle placed across a number of process trace lines. The manner in which the intersection with a trace is drawn is used to indicate whether the process participates in the abstract event.

If abstract events that do not guarantee a partial-order relationship are being displayed, 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 display. Since our prototype also contains functionality to explicitly highlight predecessors and successors of an event on the display, we are hoping that this feature can be used to make the actual precedences clearer when requested by the developer.

3 The Shoshin Debugger Prototype

A partial prototype of a distributed debugger has been implemented, including many of the concepts discussed in the preceding sections. There are at least two reasons for the prototype implementation. One is to demonstrate the practical applicability of the concepts. The other is that experience with a prototype helps us to refine our concepts and frequently suggests valuable further work, both practical and theoretical.

3.1 The Event Display

The original intent for the debugger display was 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 represented the two endpoints of a communication activity (using a horizontal arrow for synchronous communication and a sloping arrow for asynchronous communication). The events would be positioned so that if event A preceded event B, then A would be positioned higher than B. 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 needed to be handled properly to make the display easy to use and informative. As well, it became clear that using horizontal lines to represent processes would be convenient, in particular making possible the display of process names for all processes. The discussion below will assume this horizontal orientation.

Figure 2 shows part of an execution of the Hermes `helloworld` program. Each line represents a Hermes process. Time flows from left to right, different event types are shown with different round or square symbols, and processes have a dotted line while blocked. This program involves only synchronous events, so all event relationships are vertical. The mouse has been pressed on event 2 of process `helloworld`, and so additional information is displayed including the event type, number within the process, and predecessors and successors (not obvious in this figure). The paragraphs below first describe the major problem of scrolling the event display and then discuss several other issues very briefly.

An obvious practical problem is that in most cases the complete execution history of an application cannot be displayed at once because it is 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. The difficulty is that a scrollbar requires that there be some unique indexing that can form the basis for associating scrollbar positions with data to be displayed. If the event displays were based on real time, then an obvious index would exist. Since the displays are based entirely 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, and that 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 and cannot be described here. 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.

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 end up creating a list of such interesting events, with entries like “event 65 in root”. To eliminate such tedious off-line record keeping, a facility was added that allows events to be tagged. 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 prototype showed that some information also needed to be present statically in the display. Thus, the prototype was modified to show call and call-receive events as filled circles, return and receive-return 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, providing event abstraction is not used, by noting the sequencing of events within a process and the communication connections between processes. For convenience, the prototype can also explicitly mark the displayed predecessors and successors of a specified event. In complex execution histories, this can be a significant convenience for the user. At present, it is also essential for determining precise precedence when abstract events are displayed.

To allow users to tailor the display to their needs (or expectations), facilities were also added to allow

the order of the displayed processes to be changed. The simplest of these facilities moves a single process 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 which join communicating events be minimized. (Since this problem is computationally intractable, a heuristic is used. In practice, its results are generally quite good.)

3.2 Hierarchical Process Clustering

The ability to collapse processes into process clusters is provided in the prototype, including the construction of clusters containing clusters as components. This gives the user a powerful facility, but requires an appropriate interface to make it conveniently usable.

The prototype provides a separate window in which the cluster structure is displayed as a tree. 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”. Of course, these operations are specified using menus and mouse-based selection of nodes, not textually.

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 processes 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 prototype allows the user to modify the focus by simply selecting a node that should be placed in the focus. A minimal set of other changes is then made to create a legitimate focus containing that node [20].

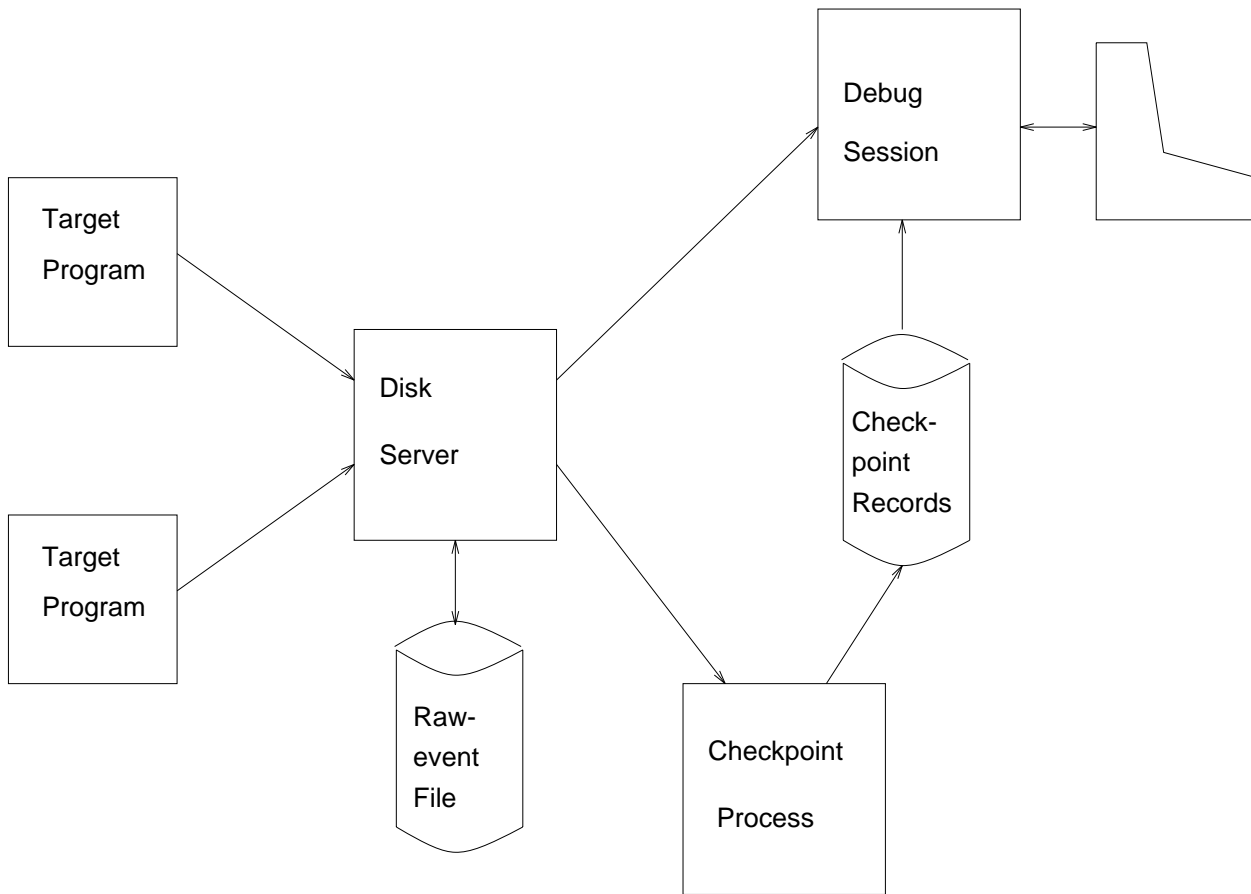


Figure 3: The Architecture of the Shoshin Debugger

3.3 Architecture of the Prototype

The architecture adopted for the prototype is shown in Figure 3. The debugger consists of three processes: the debug-session process, the disk server, 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 disk 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. Thus, event data is received from event-collection “hooks” embedded in the target environment and placed in a disk file, with only minor transformation to make it more convenient for later use. Then, clients of the disk server (the other two processes) request event data from the disk server as required.

The checkpoint process exists solely to improve performance. As discussed previously, logical timestamps are associated with events in order to determine the partial-order relationship between events. 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 which 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 disk server and its clients involves explicit message passing.

The design, consisting of a single server and multiple clients, is intended to allow easy addition of further processes if new features should make that desirable. It has already made possible a feature that might otherwise have been quite difficult. If the user would like to have multiple views into the set of events being produced by an application, it is only necessary to create additional instances of the debug-session process. These simply become additional clients of the disk server (and all rely on the single checkpoint file produced by the checkpoint process).

3.4 Performance Issues

The prototype is intended to be useful when there are many processes and many events, so both functional behaviour and performance were considered carefully when the prototype was designed.

A basic problem with the present prototype 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 20 bytes per event, it is possible to handle quite large event histories before disk space becomes a problem.

The aspect of performance that has been 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 display that is about to be replaced. 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 state it was in recently as well as a state obtained from the checkpoint file.

A cache of “raw” (untimestamped) events is also maintained in each client program. The obvious technique is used, requesting a block of events from the disk 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 process.

Thus, most of the performance enhancements are obtained by some form of caching. There is also a specific technique used to improve performance when building displays containing clusters. In that case, a basic decision is whether an event should be displayed at all. As much as possible, the prototype makes such decisions using raw-event data rather than timestamped events. Even with the significant caching of timestamped events, the average time to obtain a timestamped event is much larger than the average time to obtain a raw event, so the performance gain can be significant.

It is very hard to quantify the effects of the above performance enhancements, for two reasons. One reason is that performance is very dependent on the set of event data and the clustering, if any, being used. The other reason is that the different techniques adopted interact with each other, making a reasonable

statement about their individual effects difficult or impossible. The combined effect is clearly dramatic. A particular case involving very heavy clustering on a large set (approximately 200) of processes required over 30 minutes to generate a display even with some of the performance enhancements implemented. With all implemented, the time was reduced to less than a minute. (If the initial performance enhancements, raw-event caching and a single block of recently timestamped events, had not been implemented, it is likely that we would not have had the patience to wait for the display to appear, but it is speculative just how bad the performance would have been.)

3.5 Status of the Prototype

The facilities included in the prototype have been sketched above and can be summarized briefly as event display with intelligent scrolling plus a general, hierarchical process-clustering facility. Other facilities that are likely to be useful will probably be added as time is available. One of the most important of these is the ability to link with an “address-space” debugger so that conventional debugging facilities would be 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.) Unfortunately, at present, each of the target environments has some specific problem that makes such linkage difficult or impossible. (One obvious problem is the lack of a debugger in some cases.) Another important addition would be a “replay” facility, allowing execution to be repeated with the partial order of events constrained to be the same as in the initial execution. Such a facility existed in an earlier prototype but has not yet been re-implemented in the current one. Facilities for event abstraction are another obvious addition. A preliminary version of event abstraction has been implemented and is currently being explored.

The prototype was designed to be largely independent of the target environment. The initial implementation used Hermes [18] as the target environment, but the Hermes-dependent code was confined to a small part of the disk server. This code now amounts to approximately 150 lines of 30,000 total source lines. The

most significant part of this code is a table that defines important characteristics of the events in the target environment, such as what event types pair with each other and what symbol should be used to display an event. The prototype has now been adapted to work with four other target environments: Concert/C [21], SR [1], the μ System [7], and OSF/DCE [17]. The differences among these environments are substantial, which suggests that the prototype can indeed be adapted to a wide variety of target environments by changing only a very small part of the code. In particular, although the above description refers to “processes” on the display, the trace lines in the display can also be used to represent other objects, e.g., threads within a process in an OSF/DCE environment, or monitors and semaphores in the μ System environment.

4 Open Problems

Building the prototype debugger using our formal base for process and event abstraction led to the identification of several open problems. First, while abstraction is essential to cope with complexity, it also introduces new problems. As discussed above, the interface of a cluster does not necessarily consist of a totally ordered set of events. Consequently, a cluster differs fundamentally from a sequential process. Similarly, the adopted definition of “precedes” among abstract events does not ensure transitivity. Restricting the definition of abstract events to structures that guarantee transitivity makes abstract events both more difficult to handle and less expressive. We are currently investigating other approaches which allow for more expressive abstract events, giving up the partial-order structure.

A second problem deals with the display of abstract events in a process-time diagram as used in our prototype debugger. In general, abstract events consist of primitive events from various processes. Displaying an abstract event in the same way as a primitive event does not allow all the process traces involved to be properly reflected, so a different form of presentation must be developed. For abstract events that do not guarantee the transitivity of the precedence relation, a second display-related problem exists. As outlined above, the precedence relation among primitive events can be deduced easily from the debug display by

noting the sequencing of events within a process and the communication connections between processes. This, however, is no longer true when non-transitive abstract events are added to the display. This makes the explicit marking of precedence already in the debugger prototype all the more important. We currently have only preliminary experience with the visualization of abstract events, and so these issues need to be dealt with in more depth.

Our prototype currently has no notion of the passage of physical time—it deals only with the partial order of events captured by the event probes in the target system being debugged. For many applications, it is also useful to know the real time at which events occurred, and to be able to view time on a scale related to the passage of real time. At a low level, this requires adding a real time to each event record created by a debugging probe, and assuming that these real timestamps are at least approximately synchronized. We have only begun to explore how our display and scrolling algorithms might be modified to reflect the passage of real time more explicitly.

Most 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.

5 Conclusions

Our prototype 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 process 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 which might, in future, permit 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.

We have chosen to ignore certain features that would need to be present in any production distributed debugger. For example, while a previous prototype included a replay facility, there is none in our current software. We have also chosen to postpone the incorporation of “standard” debugging features, hoping that it would be easy to incorporate such functionality later, and/or to make use of the growing mechanisms in modern computer-aided software engineering-environments which permit communication among tools. This might allow a debugger for an address space (including a number of threads) to be used without being reimplemented, including notions of breakpointing, variable and stack display, and single stepping, in particular.

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.

References

- [1] G. R. Andrews et al. An overview of the SR language and implementation. *ACM Trans. on Programming Languages and Systems*, 10(1):51–86, January 1988.
- [2] A. A. Basten. *Hierarchical Event-Based Behavioral Abstraction in Interactive Distributed Debugging: A Theoretical Approach*. Master’s thesis, Dept. of Mathematics and Computing Science, Eindhoven University of Technology, The Netherlands, August 1993.
- [3] A. A. Basten and T. Kunz. Order and time for event sets in distributed systems. In preparation.

- [4] P. C. Bates. *Debugging Programs in a Distributed System Environment*. PhD thesis, University of Massachusetts, Dept. of Computer and Information Science, Amherst, Massachusetts, 1986.
- [5] P. C. Bates. Shuffle automata: A formal model for behavior recognition in distributed systems. COINS Tech. Rep. 87-27, University of Massachusetts, Dept. of Computer and Information Science, January 1987.
- [6] E. Best and B. Randell. A formal model of atomicity in asynchronous systems. *Acta Informatica*, 16:93–124, 1981.
- [7] P. A. Buhr and R. A. Strooboscher. The μ System: Providing light-weight concurrency on shared-memory multiprocessor computers running Unix. *Software—Practice and Experience*, 20(9):929–963, September 1991.
- [8] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39(1):11–16, July 1991.
- [9] W.-H. Cheung. *Process and Event Abstraction for Debugging Distributed Programs*. PhD thesis, Dept. of Computer Science, University of Waterloo, October 1989. Also available as CCNG Technical Report T-189.
- [10] C. J. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, August 1991.
- [11] Jay L. Gischer. The equational theory of pomsets. *Theoretical Computer Science*, 61(2,3):199–224, November 1988.
- [12] 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, Washington, March 1990. Published as *ACM SIGPLAN Notices*, **25**(3).

- [13] T. Kunz, D. P. Dueck, and J. P. Black. Automatic process clustering. Submitted to *IEEE Trans. on Software Engineering*, August 1993.
- [14] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. of the ACM*, 21(7):558–565, July 1978.
- [15] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Trans. on Computers*, C-36(4):471–482, April 1987.
- [16] F. Mattern. On the relativistic structure of logical time in distributed systems. *Bigre*, 78:3–20, March 1992. Proceedings of the workshop “Datation et Contrôle des Executions Reparties,” 4 December 1991, Rennes, France.
- [17] Open Software Foundation. *Introduction to OSF/DCE*. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.
- [18] R. E. Strom et al. *Hermes: A Language for Distributed Computing*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
- [19] J. A. Summers. *Precedence-Preserving Abstraction for Distributed Debugging*. Master’s thesis, Dept. of Computer Science, University of Waterloo, Ontario, 1992.
- [20] D. J. Taylor. The use of process clustering in distributed-system event displays. In *Proc. CASCON ’93, Vol. 1, Software Engineering*, pages 505–512, Toronto, Ontario, October 25–28 1993.
- [21] S. Yemini et al. CONCERT: A high-level-language approach to heterogeneous distributed systems. In *Proc. 9th Int. Conf. on Distributed Computing Systems*, pages 162–171, June 5–9 1989.