Achieving Target-System Independence in Event Visualisation

David J. Taylor
Thomas Kunz
James P. Black

Abstract

A process-time diagram showing the execution history of individual processes and the interactions between processes can be a very useful tool in understanding the behaviour of a distributed or concurrent application. Because of the variety of environments for writing distributed and concurrent applications, a tool that produces such diagrams is most useful if it is not tied to a particular environment. This paper[1] describes techniques that can be used to obtain such target-environment independence. Because target environments can differ in their conceptual model of process interactions, e.g., message passing versus remote-method invocation, it is important to provide a very general structure for such specifications, making as few assumptions as possible about characteristics that must be possessed by all target environments. By making reference to our experience with the implementation of the Partial-Order Event Tracer, we will suggest principles for effectively using table-driven implementations and appropriate means for dealing with odd situations that cannot reasonably be coded into a table or control file. We claim that appropriate use of such techniques need not compromise execution efficiency, relative to a tool designed for only one environment, and also allows easy implementation for a new target environment.

[1] The IBM contact for this paper is Pat Finnigan, Distributed AD Tools, IBM Canada Ltd., Dept. 583, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7

1 Introduction

Debugging applications in a distributed or parallel environment presents difficulties not found when debugging a sequential application running on a single processor. These difficulties include the lack of a well-defined total order for the events taking place, non-deterministic behaviour, and the lack of a global clock (in a distributed environment). Although many environments now exist for distributed and parallel computing, these difficulties are largely independent of the properties of specific environments. 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.

In the Shoshin research project at the University of Waterloo, we have been endeavouring to produce prototype software for debugging distributed applications, using the concepts we have developed for displays based on the partial order of events and for abstraction in large-scale applications. Achieving target-system independence in such prototype implementations was always one of our goals, but achieving it was sometimes a difficult task. In this paper, we describe the techniques we used to obtain executable programs that are completely independent of the target environment and the mechanisms that allow a broad range of target environments to be described.

Of course, there are limits to what can be achieved in a manner independent of the target environment. Our tools are based on the notion of collecting certain events from a running application, displaying them, analysing them, and possibly re-executing the application while coercing it to follow the partial order of the stored events. The actual collection of events can hardly be performed independently of the target environment. Thus, in the context of this work, target-system independence means that a particular target environment must be instrumented for event collection, the characteristics of the environment must be appropriately described for use by a debugging tool, and then the target-system-independent tool may be used with applications running in that environment.

The remainder of this paper is organised as follows. Section 2 provides a brief summary of our distributed-debugging tool, as background for the remainder of the paper. Section 3 describes the general trace-and-event model used to represent the behaviour of applications in a wide variety of environments. Section 4 then describes the event-description table, which is the key structure used to represent a target environment. Section 5 discusses the choice between encoding information in description files and providing information dynamically from a running application. It also describes some of the information about an environment that is not contained in the event-description table. Section 6 contains some conclusions and some lessons learned.

2 Prototype Tools for Debugging Distributed Applications

Our objective was to develop techniques and tools for effective debugging of distributed and parallel applications. In doing this, we were especially concerned with the need to deal with large applications, containing many processes and generating many events. Thus, much of our work concerns techniques for abstraction that will allow a user to work effectively with large and complex execution histories.

Our prototype initially provided a visualisation of the execution of a distributed application, displaying processes and events, with events related by the fundamental underlying partial order. (In a partial order, event A may precede event B, event B may precede event A, or neither of these may be true, in which case A and B are concurrent.) This was extended to provide abstraction in the process and event dimensions and to provide displays based on real time as well as the original partial-order displays.

Other extensions provided linkage with a "standard" debugger, using facilities of the AIX Workbench [4] and facilities for replay. In this context, "replay" means re-execution of the application, coercing it to follow the same partial order as during the initial execution.

The tool was used initially with programs written in Hermes [7], but use with other environments was contemplated from the beginning and extension to other environments, such as Concert/C [10] and OSF DCE [6], began once the prototype software became reasonably stable. As the software was modified, continuing efforts were taken to retain and enhance its target-system independence.

The tool is known as POET (for "Partial-Order Event Tracer") and has a multi-process structure, as indicated in Figure 1. Events flow from the target application to the disk-server process, where they are placed in the raw-event file. The remaining processes (other than the application processes) are clients of the disk server. The debug-session process provides direct user interaction, including the drawing of partial-order and real-time displays, fetching event data from the disk server as required. The checkpoint process is essentially a performance optimisation, which is not important for the present discussion.

POET may be configured, at installation or by an individual user, for one or multiple target environments, through a specification in an X Toolkit resource file. If it is configured for multiple target environments, it determines the environment to be used on this execution when the user starts the execution of a program or attaches an existing file of event data. The resulting displays can have significant differences, particularly in the semantic significance of the

lines and symbols used. Figures 2 and 3 show examples of displays obtained in the OSF DCE and μC++ [2] environments, respectively. It is not possible to describe the displays fully, but a description of a few features will illustrate the differences that can exist because of differences in target environments. In the OSF DCE display, all trace lines represent threads (in clients or servers); in the μC++ display, most trace lines represent threads of execution, but the trace line "buffer(0x30590)" represents a monitor [3]. Thus, all interactions between traces in the OSF DCE display represent remote-procedure calls, whereas the particular interactions shown in this μC++ display represent monitor entries and returns. The use of symbols, e.g., circles versus squares, also has different meanings as does the use of a solid versus dashed versus blank trace line. For the latter, in the OSF DCE display, a blank trace line means that the thread has either not started or has terminated. In the μC++ display, the trace line is blank in these cases, but is also blank when the thread of control has entered a monitor.

This particular screen dump was obtained from an older version of POET, but as of the current version, all these differences are encoded in the target-description file which is loaded at the point POET determines the target environment being used on this execution.

3 Distributed-System Model

To achieve target-system independence across a broad range of systems, it is necessary to begin with a very general model of the behaviour of applications. Our model contains two basic concepts: traces and events. A trace represents some entity with sequential behaviour and events represent actions 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, so it is simply a matter of mapping the behaviour of any particular system onto it. For example, in a standard RPC-type system, 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 the last two also being a synchronous pair.

For an environment such as Hermes, which provides asynchronous message passing as well as synchronous RPC, it is sufficient to provide additional events representing the asynchronous message passing. Most of the environments currently supported are based on one or both of RPC and asynchronous message passing, and can be mapped onto the model in obvious ways. The generality of the model can be illustrated by considering the μC++ environment, used as an example in the preceding section. It is based on shared-memory parallelism and provides facilities such as semaphores and monitors to control access to the shared memory. Thus, some traces represent processes but other traces represent monitors, semaphores, etc. Event pairs then can represent entry of a process into a monitor, blocking of a process on a semaphore, and so forth. Because monitors and semaphores also have sequential behaviour and their interactions with processes can be represented by event pairs, the model can easily describe such a system as well as a system based on some form of message passing.

At a somewhat lower level, the model must also specify how the event pairing is accomplished. Again, we have attempted to adopt a mechanism that is both simple and general. In each trace, events are numbered sequentially, beginning at zero, and it is the responsibility of the event-collection logic to generate the event numbers. To specify an event pairing, it is sufficient to identify the trace containing the other event and the event number within that trace for the partner event. It is only necessary for one of the two events in a pair to identify the other event, although specifying the pairing in both events, when possible, adds useful redundancy. (When events represent sending and receiving of messages, it is often difficult or impossible for the sending event to identify the receiving event. By passing an event identification along with the message, it is easy for the receiving event to identify the sending event.)

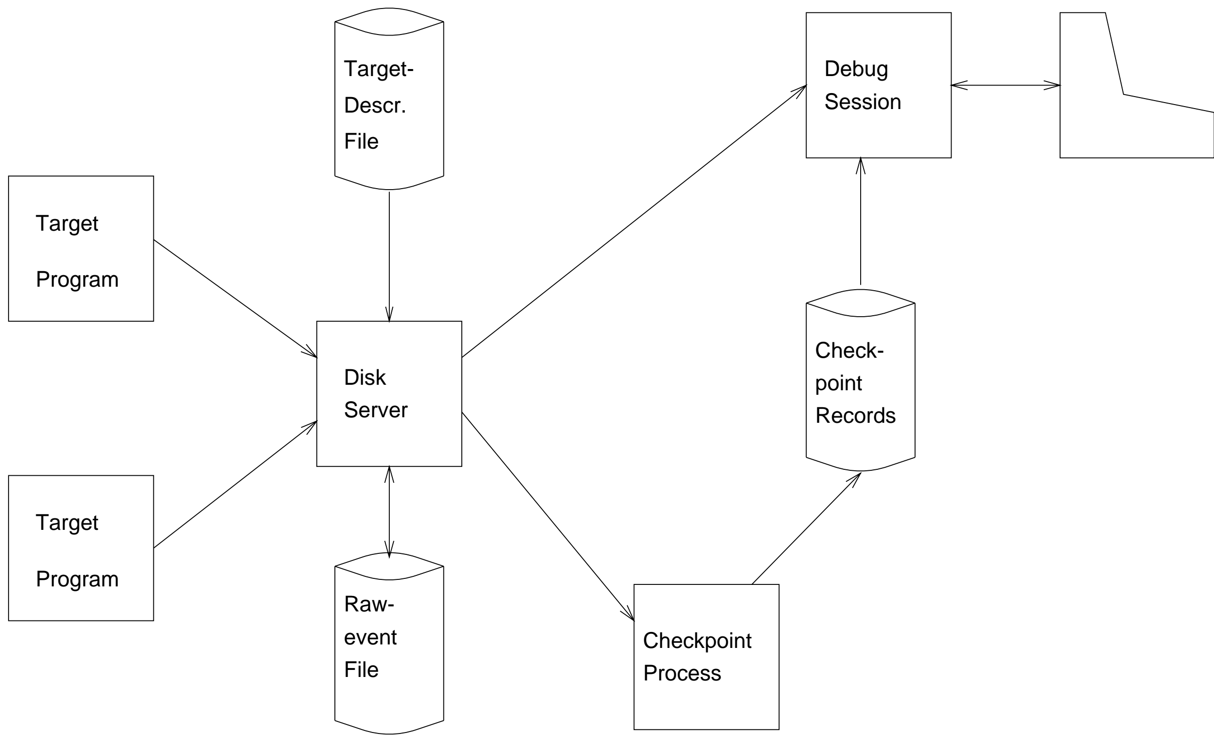

Figure 1: Process structure of POET

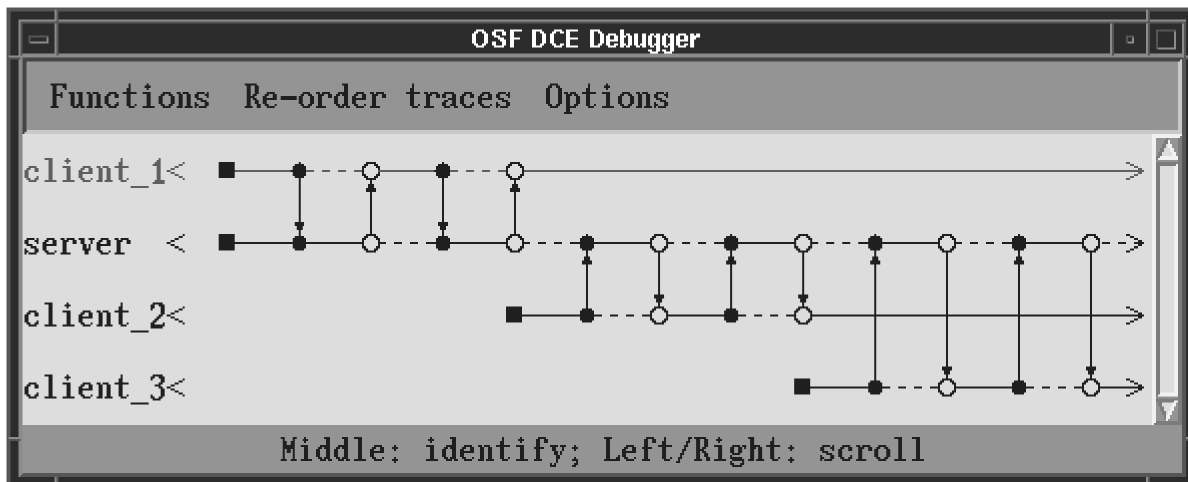

Figure 2: Display of OSF DCE execution

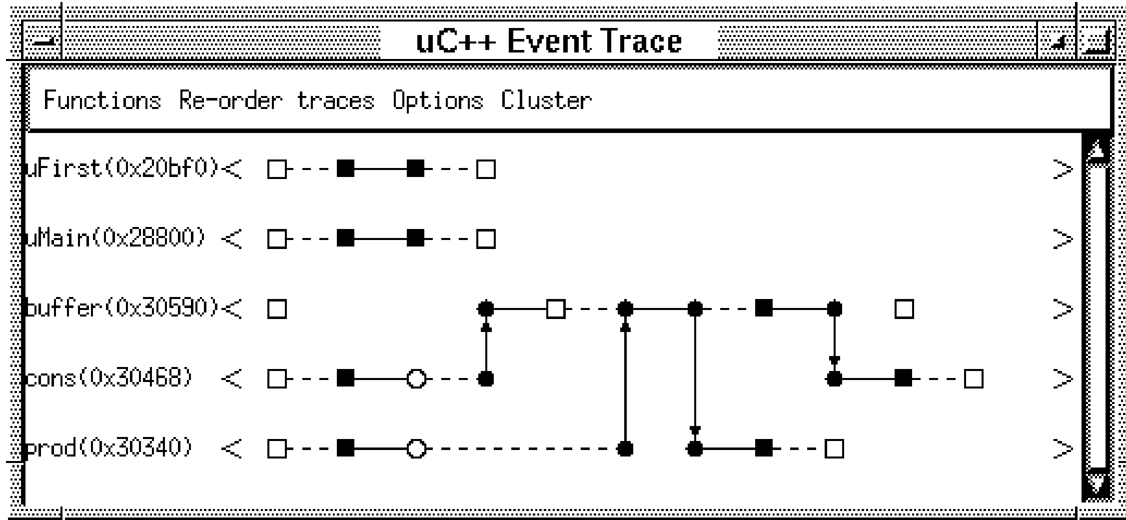

Figure 3: Display of μC++ execution

An additional practical issue needs to be taken into account in the model. Many traces may be associated with a single process, e.g., threads in an OSF DCE server. For efficiency (and conservation of file descriptors), it is necessary to multiplex such a set of traces onto a single event stream. Thus, an event is identified by a stream identification, a trace identification (which may be unique only relative to the stream), and an event number. In our software, event numbers are required to be 4-byte integers but, to allow for varying needs of target environments, both stream and trace identifications are arbitrary-length byte sequences. Because it is inconvenient to use a pair of arbitrary-length byte sequences as a unique identifier, these are immediately mapped internally into a single integer trace number (unique across all streams).

4 Event-Description Table

Given the modeling of an application as a collection of traces and associated events, the key issue in describing an environment is the description of its event structure. It is necessary to describe the types of events that exist and to indicate their relationships. The event-description table provides information about event pairings, allowable event ordering within a trace, and additional miscellaneous information. The format of the table is intended to make it reasonably simple for a developer to create it; various transformations are applied internally to create a form that is simple to access. In the table, each entry describes a single event type, with multiple entries used for one event type if necessary, as described below.

Figure 4 shows an extract from a hypothetical event-description table, describing 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.

Specifying event sequence within a trace is primarily useful as a means of providing a degree of error checking on the incoming streams of events. Three specifications can be made. A flag is used to indicate whether an event can legitimately be the first event of a trace. A next-event field is used to indicate that a specific event type must follow this event type. In the above example, the next-event field (the second field in the entry) is used to specify that a DBG_CALL must be followed by a DBG_REPLY_RECV, since the call event blocks the thread until a reply is received. A particular pseudo-event-type can also be placed in the next-event field to indicate that there is no next event, i.e., that this event type must be the last event of the trace.

```
DBG_CALL,       DBG_REPLY_RECV,
                   -1,         0,     F_CIRCLE, DASH,  "call";
DBG_CALL_RECV,  0, DBG_CALL,   TERM, -F_CIRCLE, SOLID, "receive";
DBG_REPLY,      0, -1,         0,     O_CIRCLE, SOLID, "reply";
DBG_REPLY_RECV, 0, DBG_REPLY,  TERM, -O_CIRCLE, SOLID, "reply receive";
```

Figure 4: Extract from an event-description table

Event pairing is, of course, 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 0, 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 third field of each entry, so in the above example DBG_CALL and DBG_CALL_RECV are indicated as an event pair, with only DBG_CALL_RECV containing partner data.

The synchronous/asynchronous property is specified by attaching a flag to the sending event of the pair. This reflects a subtlety of the kind that is likely to be discovered only by considering the properties of several environments. In most cases, an environment will contain disjoint sets of synchronous and asynchronous events. However, it is possible for a single type of receive operation to receive messages sent either synchronously or asynchronously. Requiring each event type, rather than just sending events, to be marked synchronous or asynchronous would not properly reflect the properties of such an environment.

All events in the above example are synchronous, so suppose that DBG_SEND and DBG_RECEIVE are an asynchronous pair of events. For DBG_SEND, the partner type will be specified as -1 and the asynchronous flag will be specified. For DBG_RECEIVE, the partner type will be specified as DBG_SEND and no flags will be specified.

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., DBG_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. The TERM flag indicates almost the same thing, namely which of the events in a pair "terminates" the interaction, but does not directly control the visual display.) A flag value indicates whether a text annotation should be expected following this event type and a second flag value indicates whether such a text annotation should be taken as the name of the trace. (None of the events in the RPC example have text annotations.)

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, DBG_CALL blocks the thread and so has linetype DASH, whereas all other line types are SOLID. (Of course, the line types cannot reflect information missing from 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 indicated.) 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.

One other issue cannot be fully described here, for space reasons, but should be mentioned. To build a display that properly reflects both synchronous event pairing and the real time of event occurrences, it is necessary to introduce fictitious events into the display [9]. The problem is that the description needs to indicate both what the real-time display should look like, including the fictitious events, and what the event stream from the application should look like, excluding the fictitious events. The problem is solved using a flag to mark the fictitious events, combined with the usual mechanism for indicating required sequences of event types.

5 Other Descriptive Information

The event-description table contains most of the critical information about a target environment. A compiled form of the table is placed in a file that can be loaded at execution time to customise POET for that environment. Other information about the target environment can be supplied in two ways: by adding it to that file or by having it passed directly from the application program, at the beginning of the event stream. Some additional information is included in the file, such as titles to use for the event-display window and program-execution windows.

Information is provided at the beginning of the event stream that is crucial to its interpretation, notably the lengths of the stream identification, trace identification, and text annotations. If some event types are omitted from the event-description table or are improperly described, the event stream will not be interpreted properly, but errors will occur only with respect to the events in question. If some of the lengths are improperly specified, the code receiving the event stream will lose track of event boundaries and will be unable to extract anything useful from the event stream. Because of the major consequences of errors in these lengths, it is prudent to place them directly in the event stream rather than relying on proper correspondence between the event-collection logic and an external file.

Early versions of POET did not provide this information directly in the event stream. We discovered that someone working to make POET available in a new target environment might copy too much of the specification from an existing target environment, including the code that is supposed to uniquely identify the environment. The result was that the two environments could be confused and if relevant length parameters did not match, chaos ensued. Although one can argue that duplicating the identifying code is an error that shouldn't be excused, reacting as well as possible to such problems is a reasonable design objective.

The information discussed above (in this and the preceding section) is sufficient to describe most properties of a target environment, as needed for providing event displays, controlled replay of execution, etc. In real life, there will always be some things that do not fit neatly into such a table-driven structure. As we encounter unusual problems in various target environments, the first approach is always to find some adaptation or extension of the existing table-driven structure that will accommodate the situation. When such an approach fails, it is highly desirable to have a solution that does not involve adding code of the form "if (target_environment == X) ..." to programs that are otherwise target-environment independent.

We have used the client-server structure of the tool to provide an escape mechanism for dealing with such problems. The disk server is not restricted to dealing with only the debug-session and checkpoint clients indicated in Figure 1, but contains a table allowing an arbitrary set of clients to be used. A mechanism has been provided for copying data into this client table from the target-description file. This mechanism has been used in two slightly different ways, to handle what might be described as known and unknown target-specific problems.

An example of a known target-specific problem is default clustering. Process clustering can be used to group traces and hide activity internal to sets of traces [8]. In some target environments, parts of the run-time system are "exposed" in the event display, but are normally of little or no value to an application developer. Thus, a default-clustering facility is provided that puts the exposed system traces into a cluster, removing them from view. The problem is that in the two current target environments that have this property, Hermes and ABC++ [1], the algorithms for determining the default clustering are radically different. Both algorithms rely on a complicated combination of data involving trace names, trace identifiers, and events. Although the Hermes default-clustering algorithm is table-driven, extending it to handle ABC++ would be unreasonable. Thus, default clustering is implemented by creating a target-environment-dependent client program to perform the clustering. Such a client can be designed using two existing specifications: the standard client-server interface of the disk server and the format of a cluster-description file, as is written out when the user specifies that a cluster hierarchy should be saved.

An example of an "unknown" target-specific problem is the handling of names in ABC++. In that environment, it is not reasonable to obtain the textual names for processes, object methods, etc., directly during program execution. Instead, information such as addresses is obtained and must be translated elsewhere. This translation is also provided by a target-environment-specific client. In this case, the client fetches the textual annotations, transforms them using information in object-module symbol tables, and then asks the disk server to rewrite the stored annotations.

Clients to address these problems are specified in the target-description file, and subsequently placed in the client table, only if relevant to the target environment. A client to solve the first type of problem is placed in a predefined location in the client table and is used under predefined circumstances. (At present, the above example is the only such client.) A client to solve the second type of problem is placed in one of two generic slots provided for the purpose and is normally started immediately. The disk server and other client processes do not have any knowledge of the function that such a client may perform, but by using the standard interface to the disk server, a useful client can be constructed.

Finally, we have avoided dealing directly with at least one problem by making appropriate use of the facilities for event abstraction [5]. As indicated above, event pairing is supported but there is no direct support for connection of larger sets of events. In an environment that provides multicast or broadcast communication, we collect a set of events that represents the communication as several consecutive sends paired with the individual receives. Abstraction is then used to combine the many elementary send events into a single abstract send. The automatic construction of such abstract events is supported using a target-environment-specific client, as just described.

6 Conclusions

When we began developing tools for debugging distributed applications, target-environment independence was an initial goal. Our first prototype used a table for some event-related information, but had two major problems. One was that a substantial quantity of target-environment information was explicitly coded into various functions. The other problem was that the event table was very complex. It was a square array, indexed in both dimensions by event type. In practice, it was very difficult to generate proper values for the (many) entries in the array and it would have been extremely difficult to document the construction of the array so that someone else could reasonably have coded one for another environment.

The first prototype provided negative experience that was very helpful in building a second prototype. In that prototype, the event-description table was kept quite simple and more care was taken to eliminate target-environment dependencies from the code. To create a working prototype reasonably quickly, a few such dependencies existed initially, but were subsequently eliminated. For some time

the level of target-system dependence was that a file containing the event-description table was linked into the disk server and some of the disk-server files were compiled using a target-system-dependent header file. Thus, the source for the disk server and the source and executables for the other processes were target-system-independent. More recently, all dependencies have been moved out of the executable code, primarily through the creation of the target-description file.

Most of the lessons learned from the experience of developing techniques to make our code target-environment-independent are reasonably obvious, at least in retrospect, but it may be useful to list some of them. (1) Put as much information as reasonable into tables and description files. These files should be simple in structure and content. Wherever appropriate, preprocessing or processing during tool initialisation should be used to reduce the complexity of the information that must be supplied by the individual coding the table or file. (2) As far as possible, assume nothing about the target environment. For example, make everything variable-length that might conceivably differ between target environments. Then, for efficiency, map the variable-length fields to a fixed-length internal representation as soon as possible. As a more specific example, do not assume that an event cannot pair with both a synchronous and an asynchronous partner event. (3) Supply highly critical information (such as field lengths) directly from the target environment, to minimise the possibility of disaster. (4) Acknowledge that some things cannot be handled by general, table-driven mechanisms and provide well-defined mechanisms for accessing environment-specific code to handle those things. Our use of a client-server structure is one possible means; dynamic loading of code, where supported by the operating system, would be another. (5) Finally, use low-level, generic system services as much as possible. For example, we use TCP streams for passing events from the target application to the disk server, because TCP is widely available. Individuals looking at only one target environment, e.g., OSF DCE, have advised us that we should use DCE facilities for transporting events. We have always ignored such advice because it would then tie the entire tool to one particular environment.

Although it is not possible to provide hard evidence, we believe that our use of a general-purpose implementation, not tied to any particular environment, has had very little impact on efficiency. One of the most critical efficiency issues is the probe effect induced by the event-collection code in the target program. Other than the requirement that an initial block of header information be provided on each event stream, there is essentially no effect on the event-collection code, so the probe effect is almost entirely independent of our implementation techniques. (In practice, the cost of collecting an event is the cost of the system call to write it to the TCP stream, plus negligible cost in the event-collection code itself.)

In the tool itself, there are some costs associated with generality. Some structures must be allocated dynamically because their lengths are not known at compile time and some data must be copied using function calls rather than assignments for the same reason. As well, there are some additional costs associated with initialisation, as data is extracted from the target-description file by the disk server, used for its own purposes and passed to the client programs. None of these costs appears significant. As well, there is the impossible-to-quantify issue of optimisation. Because the tool is being used across several target environments, considerable effort has been expended in optimising its performance. A tool with more limited applicability might not have justified such effort in optimisation.

Acknowledgments

The work described in this paper 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, under Research Grant OGP00003078 and a Senior Industrial Fellowship, by the Information Technology Research Centre, and by IBM. The authors would also like to thank Peter Buhr for providing the screen dump of the μC++ POET display.

About the Authors

David Taylor is an Associate Professor of Computer Science at the University of Waterloo, where he has been a faculty member since 1977. His research interests include distributed-systems software and software fault tolerance. During the 1991-1992 academic year, he spent a sabbatical at the Centre for Advanced Studies, IBM Canada Ltd. Laboratory.

His e-mail address is dtaylor@uwaterloo.ca.

Thomas Kunz is an Assistant Professor of Computer Science at the University of Waterloo, where he has been a faculty member since 1994. His research interests include load balancing in distributed systems, distributed debugging, management of distributed applications and systems, mobile computing, and reverse engineering/program understanding.

His e-mail address is tkunz@uwaterloo.ca.

Jay Black is an Associate Professor of Computer Science at the University of Waterloo, where he has been a faculty member since 1984. His research interests include the management of distributed applications and systems, mobile computing, and distributed-system security. He is also Director of the Mathematics Faculty Computing Facility.

His e-mail address is jpblack@uwaterloo.ca.

All authors can be reached at the address Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1. All authors are also members of the Shoshin research group; more information about the activities of that group can be found at http://ccnga.uwaterloo.ca/ .

References

[1] E. Arjomandi, W. O'Farrell, I. Kalas, G. Koblents, F. C. Eigler, and G. R. Gao. ABC++: Concurrency by inheritance in C++. *IBM Systems Journal*, 34(1):120–137, 1995.

[2] 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, September 1990.

[3] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974.

[4] International Business Machines Corporation. *IBM AIX SDE Workbench/6000 User's Guide and Reference.* SC09-1453.

[5] T. Kunz. Visualizing abstract events. In *Proceedings of CASCON '94*, pages 334–343, Toronto, Ontario, October 31–November 3 1994.

[6] Open Software Foundation. *Introduction to OSF/DCE.* Prentice-Hall, 1993.

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

[8] D. J. Taylor. The use of process clustering in distributed-system event displays. In *Proceedings of CASCON '93, Volume I*, pages 505–512, October 24–28 1993.

[9] D. J. Taylor and M. H. Coffin. Integrating real-time and partial-order information in event-data displays. In *Proceedings of CASCON '94*, pages 157–165, Toronto, Ontario, October 31–November 3 1994.

[10] S. Yemini et al. Concert: A high-level-language approach to heterogeneous distributed systems. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 162–171, June 5–9 1989.