

# Visualizing Abstract Events

Thomas Kunz

## Abstract

Because of the complexity of distributed applications, understanding their behaviour is a challenging task. The top-down use of suitable abstraction hierarchies is frequently proposed to manage this complexity. One commonly used abstraction is to group primitive events into abstract events. This paper<sup>1</sup> presents a graphical representation for *convex* abstract events. This representation can easily be included in the process-time diagrams frequently used to depict the behaviour of distributed applications. Such visualizations, in turn, are helpful during the construction, debugging, and monitoring distributed applications as well as in trying to understand old “legacy” code in a program-understanding task. We added such a graphical representation to a prototype distributed debugger. Some examples of the resulting abstract visualization of the execution behaviour are given. These abstract visualizations are essential to minimize the complexity of the understanding process, and support top-down behaviour analyses.

## 1 Introduction

Empirical evidence shows that programmers spend the majority of their time on program maintenance tasks [3]. A common first (and difficult) step in all maintenance tasks is to gain an understanding of the application at hand. In particular, highly complex applications are difficult to understand and therefore expensive to maintain [1]. *Distributed applications* are one class of applications with a high degree of complexity [4, 5]. Some of the more frequently cited reasons are parallelism, non-deterministic execution, lack of a global component (clock, memory), significant and variable communication delays, and a tendency toward large scale. These characteristics make distributed applications simultaneously highly complex and hard to understand. Graphical visualizations are used more and more frequently to present and analyze the execution of distributed applications [16]. Visualizations are used during the construction, debugging, and monitoring of distributed applications as well as in trying to understand old “legacy” code in a program-understanding task. This paper discusses visualizations at abstract levels.

A top-down approach can best address the complexity of the understanding task. A top-down approach starts with the component at the top of a hierarchy and successively moves to the next lower level. In the context of program understanding, a hierarchy of abstract behaviour visualizations can be used to minimize the complexity involved. Typically, a programmer starts the understanding task by trying to gain a global overview [6]. This can be

---

<sup>1</sup> The IBM contact for this paper is Patrick Finnigan, Client Server Technical Center, Application Development Technology Centre, IBM Canada Ltd., Mail Stop 3P, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7.

achieved by viewing an execution at a very high level of abstraction. In the course of the understanding process, more difficult or interesting parts of the execution are isolated and examined at lower levels of abstraction. In this way, more and more detailed information is collected for smaller and smaller parts of the application, keeping the overall amount of information collected manageable. The process continues until all aspects of interest are sufficiently understood. Ideally, more than one abstraction hierarchy should be employed, each hierarchy emphasizing different aspects of the program behaviour. A programmer or maintainer could then switch between these hierarchies, depending on his or her current focus of attention.

Such abstraction hierarchies, however, are difficult to construct. Programmers typically approach the understanding task by mentally applying abstraction to the application at hand, using information from the software source, related documents, and previous experience. This abstraction process is tedious and error-prone for complex applications. Our work focuses on the development of tools to derive suitable abstraction hierarchies automatically, as described in [8, 9]. This paper addresses the issue of visualizing the behaviour at various abstraction levels. Of particular interest is the display of abstract events, that is, groups of more primitive events. Another abstract visualization technique, process clusters, is discussed in [15].

The sample application discussed in this paper is written in Hermes, a high-level, process-oriented language for distributed computing [13]. Hermes was chosen from the large number of distributed programming languages for a number of reasons. First, Hermes is based on the message-passing paradigm frequently used in distributed computing. This facilitates porting our results to other environments. Second, processes in Hermes are both the unit of parallelism and the unit of modularization. Even a moderate Hermes application consists of a large number of processes. Hermes therefore is a convenient vehicle for testing our ideas in a complex environment. A last, more pragmatic reason is that we had access to the source of an existing visualization tool for Hermes applications.

This paper is organized as follows. After this introduction, process-time diagrams frequently employed to depict the execution of a distributed application graphically are presented. Basic properties of a distributed execution are discussed, and the concept of *convex* abstract events is introduced. The graphical representation chosen for these abstract events is discussed next. Using a simple distributed application, examples of abstract visualizations are given. The paper finishes with a summary of our work and suggestions for possible future work.

## 2 Process-Time Diagrams

An *event-based* approach is typically employed for reasoning about distributed applications. Following [11], each process or autonomous entity is viewed as a totally ordered sequence of atomic events. An event can be thought of as the instant at which some computation is completed. Events constitute the lowest level of observable behaviour. They could be defined according to the particular aspect a specific observer is interested in. An observer interested in the file system might define events related to the completion of file open, read, write, and close actions. Similarly, an observer interested in security issues could define events related to encryption and decryption computations. However, a more general and fixed set of events is most commonly used. The lowest level of observable behaviour in a distributed system, the *primitive* events, are events that relate processes to one another, such as sending and receiving messages or process creation and termination. It is assumed that the local (sequential) computation for each process can be dealt with using traditional approaches.

A distributed application consists of a number of autonomous and sequential processes, cooperating to achieve a common goal. Cooperation includes both communication and synchronization and is achieved by the exchange of messages. Both synchronous and asynchronous message passing is allowed. The communication channels may or may not have the FIFO property. Processes can be created and terminated dynamically. This activity is depicted in

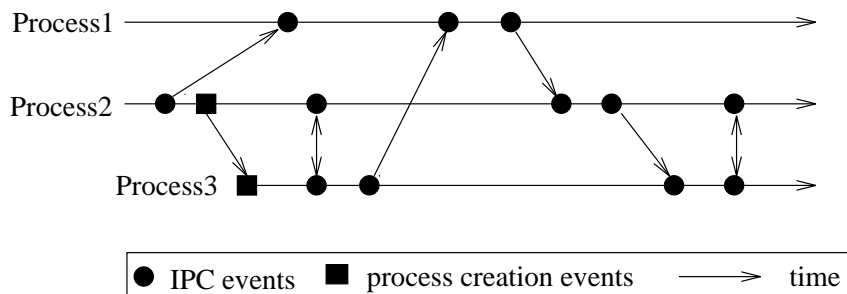


Figure 1: Model of a Distributed Application

Figure 1.

In these *process-time* diagrams, one dimension represents processes, and the other dimension represents time. In Figure 1, time flows from left to right. Messages are drawn as arrows, connecting the corresponding send and receive events. Asynchronous message passing is depicted by slanted arrows, indicating that sending a message occurs before it is received. Synchronous message passing is drawn as a vertical arrow to indicate that sending and receipt of a message occur at the same moment. Such diagrams are used widely, see, for example [11, 12].

The basic relation between primitive events is the *happened before* relation introduced by Lamport[11], and originally defined for IPC events in asynchronous systems only. This relation can easily be extended to cover process creation and termination events as well as synchronous message passing [9]. The  $\rightarrow$  relation induces a partial order on the set of events and is of particular importance for the analysis of distributed applications. An event  $a$  cannot influence another event  $b$  if it does not happen before that event. The  $\rightarrow$  relation captures the notion of potential causality, and the partial order induced by  $\rightarrow$  is sometimes referred to as causality order or a causality graph. Two events cannot influence each other when they are concurrent, that is, unrelated by the  $\rightarrow$  relation. Any display of a distributed execution should explicitly indicate this important relation. In Figure 1, the  $\rightarrow$  relation can easily be deduced. A primitive event  $a$  happens before another primitive event  $b$  if and only if there

exists a directed path from  $a$  to  $b$ , following message arrows and process lines in the time direction.

A number of logical-timestamp schemes have been proposed to determine the  $\rightarrow$  relation between any two events efficiently [11, 12]. Typically, timestamping techniques consist of two parts: a timestamping algorithm and a timestamp test. The algorithm describes how to assign timestamps to events, and the timestamp test is used to determine the  $\rightarrow$  relation from these timestamps.

### 3 Convex Abstract Events

Event abstraction allows groups of basic events to be combined into “abstract events.” The motivation is to reduce complexity by eliding unwanted details. Event abstraction may also allow some form of reconstruction of the high-level viewpoint seen by a programmer. For example, an RPC could be recognized and displayed as a single event.

A first step in event abstraction is to define a suitable precedence relation on abstract events. We have chosen the following definition:

**Definition 1 (Precedence on event sets)**

The precedence relation  $\rightarrow$  between two sets of events  $A_1$  and  $A_2$  is defined in terms of the  $\rightarrow$  relation between primitive events as follows:  $A_1 \rightarrow A_2$  if and only if there is at least one event  $a_1 \in A_1$  and at least one event  $a_2 \in A_2$ , for which  $a_1 \rightarrow a_2$ .

An advantage of this definition is that abstract events that are unrelated by the relation are truly concurrent: there is no causal rela-

tion between them. A disadvantage is that this relation is neither anti-symmetric nor transitive, so its reflexive closure does not determine a partial order.

One possible approach toward event abstraction is to restrict the abstract events to event sets with structures that preserve a partial-order structure of causality, even under Definition 1. A short survey of this work can be found in [2]. Our work follows a different approach, trading flexibility and expressiveness against the loss of this partial-order structure. This is not as heavy a price as one might expect at first sight. As long as abstract events can be timestamped in a way that reflects the precedence relation correctly, and this information can also be displayed appropriately, abstract views of the execution still present correct views of the causality among primitive events.

Primitive events are atomic, but arbitrary abstract events are, in general, not. Preserving something similar to this atomicity property can require abstract events to be *convex*, see also [7, 9]:

**Definition 2 (Convexity of event sets)**

A set  $E$  of events is *convex* if and only if:

$$\forall a, b \in E : a \rightarrow c \rightarrow b \Rightarrow c \in E$$

Note that the constituent events  $a$ ,  $b$ , and  $c$  in this definition do not necessarily represent primitive events only. The convexity property has to hold for all constituent events of an abstract event, whether they are primitive or abstract events. So far, the  $\rightarrow$  relation is defined on primitive events (Lamport[11]) and on event sets (Definition 1) only. Conceptually, however, there is no difference between a primitive event  $e$  and the event set  $\{e\}$ . Therefore, everything defined for event sets holds for primitive events as well.

Convexity is a meaningful requirement for abstract events for the following reason. For a convex abstract event  $A$ , there is no (primitive or abstract) 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.

Convexity is a useful requirement too. First, convex abstract events are easier to detect than

arbitrary event sets because it is not necessary to filter out interfering events. Second, convexity is a weaker structural requirement than most of the structures discussed in [2]. Hence, more event sets can be considered “legal” abstract events, increasing the flexibility and expressiveness in presenting a distributed execution at abstract levels. Third, because of the absence of interfering events, convex abstract events are considerably easier to display than arbitrary event sets. Finally, timestamping convex event sets can be done more efficiently than timestamping non-convex event sets [9]. Kunz [10] describes in detail the necessary timestamping algorithm and test, and proves that the timestamps accurately reflect the precedence relation of Definition 1.

The specification and recognition of abstract events is not an easy task. Black et al. [2] gives an overview of some of the work pursued in our research group. A tool for the automatic derivation of convex abstract events, following a reverse-engineering approach, is described in detail in [9]. In the following, we assume that specifications of convex abstract events are available, and we focus on the problem of how to display these abstract events.

## 4 Displaying Convex Abstract Events

A prototype distributed debugger [14] was modified and enhanced to visualize the execution of a distributed application at abstract levels. This tool provides the basic facilities to display process-time diagrams of the execution of a distributed application, combined with scrolling features in both the time and process dimensions. Scrolling is essential because the execution of a distributed application cannot be displayed completely on a single screen. However, no display facilities exist to depict abstract events. This section proposes a graphical representation of abstract events and mentions necessary modifications to the basic display-building algorithms. To allow a user to navigate through the display at various abstraction levels, we added user options to manage the abstraction level of the display to the debugger as well. These options, however, will

not be described in detail here.

Abstract events can be displayed in a number of ways. A first possibility is shown in Figure 2. Here, abstract events are represented by symbols similar to the ones chosen for primitive events in Figure 1: ideally, these would be symbols occupying little or no display area. If one event precedes another, they are connected by an arrow and the whole graph layout could be chosen in such a way that an event  $a$  that precedes another event  $b$  is always placed to the left of  $b$ . The vertical placement of each event, however, is not determined precisely.

In general, abstract events contain primitive events from different processes. Placing an abstract event on the line representing any of the processes in its location set, however, is rather arbitrary. Therefore, the notion of process lines in this type of display is given up. Events are placed anywhere along the vertical dimension, potentially subject to other display criteria such as minimizing the sum of all arrow lengths.

This simple solution has a number of disadvantages. Figure 2 indicates a basic interpretational problem with such displays. In example 1, event  $a$  seems to spawn an event  $c$  and continue in event  $b$ . In example 2, event  $a$  appears to split into the two events  $b$  and  $c$ . Both examples display the same event set with the same precedence relation, but with different intuitive interpretation. Another problem is that this type of display loses all connections to the regular debugger display. For example, two abstract events containing consecutive primitive events from the same process will not necessarily be placed adjacently in the horizontal dimension. In general, mapping events to processes (for the ensuing maintenance task) as well as switching between the primitive and abstract displays will be tedious.

To address these issues, we devised a display that integrates the visualization of abstract events into the regular debugger display. The goals pursued with this display are that it correctly reflects the  $\rightarrow$  relation, minimizes the display space needed, and explicitly indicates the process traces involved. Figure 3 contains an example of the representation chosen. Contrary to the display in Figure 2, this representation can be incorporated into a standard

process-time diagram.

In this display, an abstract event is represented by an open, vertical rectangle, stretching over the range of all processes involved. The intersection of this rectangle with a process is drawn as a filled square if primitive events from this process are constituents of the abstract event. Arrows indicating asynchronous messages enter or leave an abstract event in the vertical position that corresponds to the process generating the primitive message event. It follows that multiple arrows might enter or leave the same position.

Abstract events occupy the same horizontal display space as primitive events. In fact, each abstract event is represented in the display by its last primitive event in this trace, since this event type determines the way the process line is drawn after the abstract event. Given that, in general, an abstract event contains more than one primitive event in at least one of the processes in its location set, this display reduces the horizontal display space required for visualization. Furthermore, it explicitly indicates the process traces involved in each abstract event, fulfilling two of the three criteria listed above. However, it does not directly reflect the  $\rightarrow$  relation on the primitive events. Take for example the events  $a$  and  $b$  in Figure 3. The display does not depict whether the two events are related by the  $\rightarrow$  relation. The precedence relation depends on whether there exists a path through the first abstract event that connects these two primitive events. Adding information about all possible paths through each abstract event makes the display too cluttered, losing the major benefit of event abstraction. Rather, the user can rely on the facility provided by the debugger to explicitly highlight the successors and predecessors of a user-selected event. If, for example, the user picks event  $a$  and event  $b$  is highlighted as its successor, a path through the first abstract event exists.

Deriving a graphical representation for abstract events is only the first step toward their display. A display in the debugger prototype is build following certain principles. The display-building algorithms implicitly assume primitive events that are related by a precedence relation that is a partial order, that is, a transitive

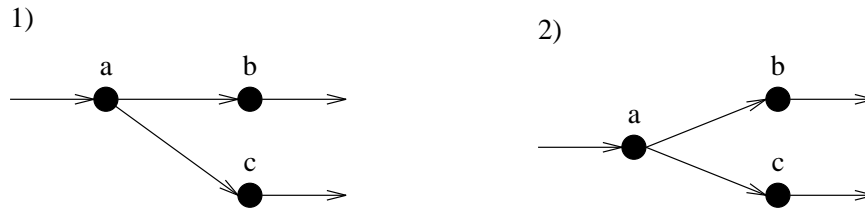


Figure 2: Displaying Abstract Events like Primitive Events

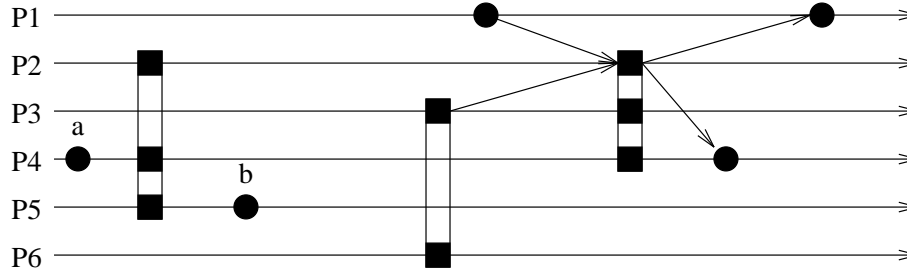


Figure 3: A Graphical Representation for Abstract Events

and anti-symmetric relation. These algorithms were modified to build displays containing abstract events with a precedence relation that is not a partial order.

## 5 Visualizations

This section presents a few visualizations at various abstraction levels for the same simple distributed application. These examples show how the complexity of the understanding process is reduced by the use of abstract behaviour visualizations. The abstract events shown in the following figures were all derived automatically with the event abstraction tool described in [9].

The execution visualized here is an execution of the `boundedbuffer` application described in the Hermes tutorial [13]. This application implements a simple bounded buffer for text strings and essentially consists of two processes. Process `boundedbuffer` implements the bounded buffer, and process `bbintf` provides a line-oriented user interface to the bounded buffer. In the traced execution, one string is

put into the buffer, fetched from the buffer and displayed, and the application terminates. During the execution, four additional processes are created: two processes `findfile_proglit` that help locate a file given its name, a process `itoa` that converts integers into strings, and a process `userrm` that presents an interface to the Hermes resource manager for application processes. The execution creates an event file containing 1874 primitive events from a total of 126 processes. (The first 120 processes form the standard Hermes runtime system.)

Figure 4 depicts part of the execution of `boundedbuffer`, using the display provided by the original Hermes debugger. This display extends the standard process-time diagrams with slightly more information about an execution. The figure shows only a subset of all processes, those relevant for the current execution segment. The scrollbar to the right of the display allows for scrolling in the vertical dimension, that is, the process dimension. The display consists of a set of horizontal lines, one for each process in the application, placing a symbol on the appropriate line for each event. These lines are preceded by the pro-

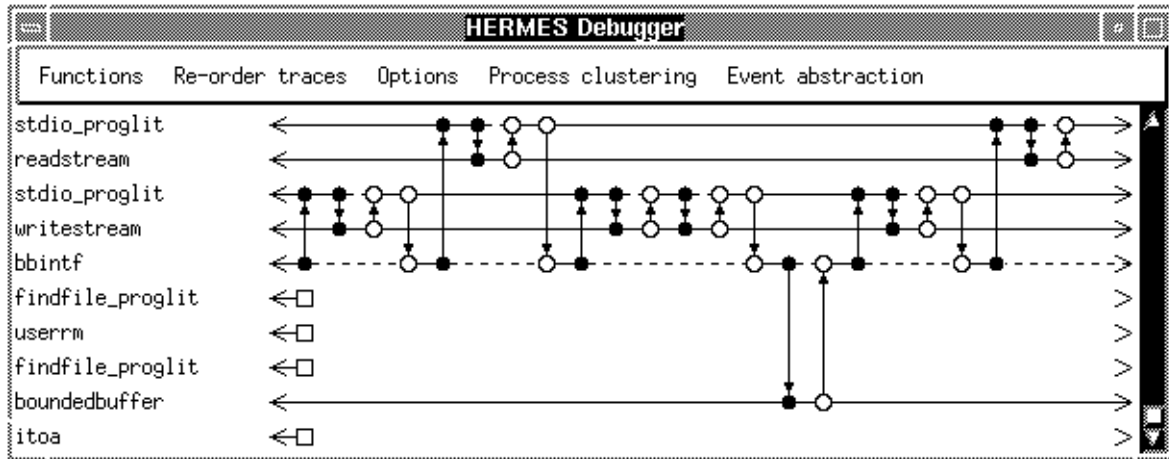


Figure 4: The boundedbuffer Execution

cess name. Events that represent the two endpoints of a communication activity are connected by an arrow, a vertical arrow for synchronous communication and a sloping arrow for asynchronous communication. The arrow points from the sending to the receiving event. Different symbols are used to distinguish different types of events. Filled squares indicate process-creation events; open squares depict process-termination events. Filled circles represent (synchronous or asynchronous) message sends and receives, and open circles the return and receive of a return for synchronous messages. The process lines are drawn in three different states, approximating the process state. Before a process starts (indicated by a filled square) and after its termination (the open square), the line is drawn in invisible mode, indicating that the process does not exist. After sending a synchronous message, a process is blocked until it receives the return message. This is represented by a dashed line. In all other cases, a process is assumed to be live and active, displayed by a solid line. Figure 4 shows the part of the execution where a string is entered and put into the buffer. Afterward, the prompt is displayed again, and the next command (to get the string from the buffer) is read in.

Figure 5 shows a visualization of the execution that starts with the same events as the one

shown in Figure 4, at an intermediate event abstraction level.

The first abstract event corresponds to the action “put a string into the buffer”. The second abstract event contains all primitive events necessary to display a prompt and enter the next command. The third abstract event represents the action “fetch a string from the buffer and display it”. The fourth abstract event again summarizes all primitive events necessary to obtain the next user input, and the fifth abstract event matches the termination activities. Because of its use of event abstraction, Figure 5 depicts a much longer sequence of the execution history than Figure 4. To indicate the amount of display space saved by a single abstract event, Figure 6 shows the same execution segment, displaying all constituent primitive events of the third abstract event.

As mentioned, the  $\rightarrow$  relation on primitive events is a partial order, but the  $\rightarrow$  relation on abstract events is not. Consequently, using an abstract display to reason about an execution might be harder. One possible remedy for this additional complexity of the abstract visualizations is to restrict the abstraction operation. Such restrictions could limit abstract events to structures that guarantee transitivity of the precedence relation; see, for example [7]. As pointed out before, this approach severely limits the expressiveness of the result-

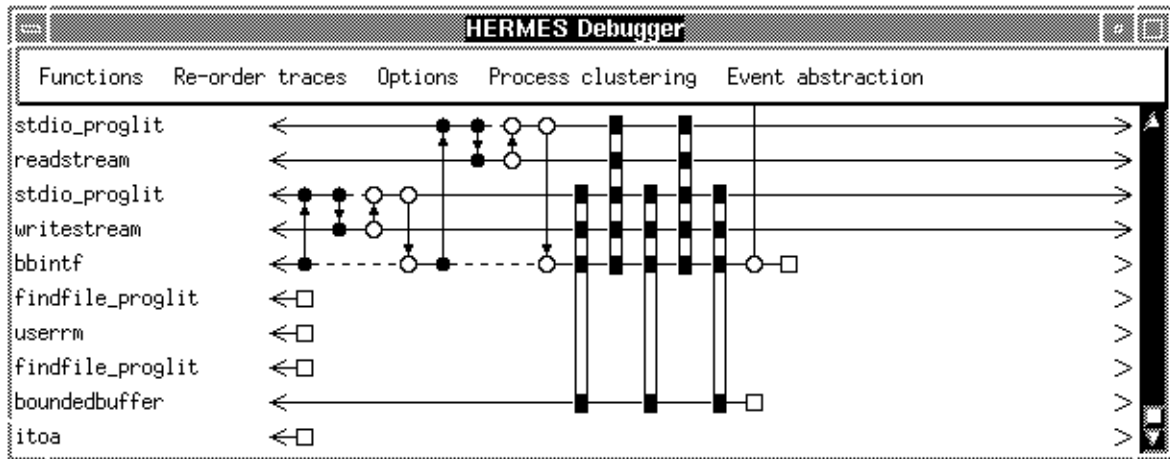


Figure 5: Intermediate Event Abstraction View of the boundedbuffer Execution

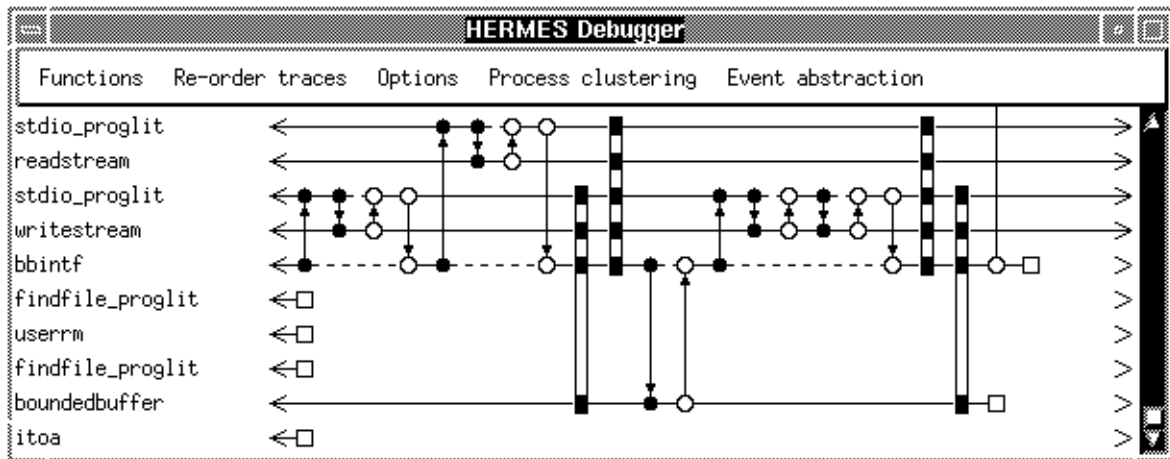


Figure 6: Display Space Reduction through Event Abstraction

ing abstractions. To retain flexibility in the derivation of abstractions, we deliberately accept a potentially higher complexity of the resulting abstract visualizations. The visualizations provided by our tool help to reduce this additional complexity. The set of predecessors and successors for each (primitive or abstract) event can be highlighted on the display, facilitating reasoning about causality in the absence of transitivity.

## 6 Conclusions and Future Work

A representation of *convex* abstract events was developed and implemented. This representation can be incorporated into the standard process-time diagrams frequently employed to visualize distributed executions. To visualize the behaviour of distributed applications at various abstraction levels, we modified an existing prototype debugger. The modifications extend the standard (low-level) visualization of the program execution provided by the debugger. The basic display algorithms were extended to support displays containing abstract events. One specific execution was visualized at various abstraction levels by using a simple distributed application. The figures indicate that such abstract visualizations indeed reduce the display complexity. The resulting abstract views of an application execution can then be used in a top-down manner to minimize the complexity of the program-understanding task.

In [15], another abstract visualization technique is discussed: process clusters. We are currently investigating the combination of both techniques in one display. Some of the resulting preliminary visualizations are discussed in [10].

A number of further improvements are planned. The abstract visualizations have to be integrated more smoothly with the other functionality provided by the distributed debugger. Furthermore, a convenient user interface for the display of complete event abstraction hierarchies is needed. Such an interface would allow a user to work directly on the event abstraction hierarchy, for example, to modify it manually. Given the huge number of primitive events in

each execution and the size of the event abstraction hierarchy, designing such an interface raises interesting problems of scale.

## Acknowledgements

The research reported here was partly done in cooperation with the Shoshin research group at the University of Waterloo, Ontario, Canada. In particular, the work benefited from numerous discussion with professors James P. Black and David J. Taylor.

## About the Author

Thomas Kunz is currently assistant professor at the University of Waterloo, Ontario, Canada. His research interests cover the areas of load balancing in distributed systems, distributed object-oriented programming languages, distributed debugging, and reverse engineering/program understanding. He recently finished his Ph.D. thesis at the Technical University Darmstadt, Federal Republic of Germany, entitled *Abstract Behaviour of Distributed Executions with Applications to Visualization*.

He can be reached at: Department of Computer Science, University of Waterloo, Waterloo, Ontario N2L 3G1. His Internet address is tkunz@neumann.uwaterloo.ca.

## References

- [1] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software Complexity and Maintenance Costs. *Communications of the ACM*, 36(11):81-94, November 1993.
- [2] J. P. Black, M. H. Coffin, D. J. Taylor, T. Kunz, and A. A. Basten. Linking Specification, Abstraction, and Debugging. CCNG Technical Report E-232, Computer Communications and Networks Group, University of Waterloo, November 1993.
- [3] Erich Buss and John Henshaw. A Software Reverse Engineering Experience. In *Proceedings of the 1991 CAS Conference*,

- pages 55–73, Toronto, Ont., Canada, October 1991. IBM Canada Ltd. Laboratory, Centre for Advanced Studies.
- [4] Hector Garcia-Molina, Frank Germano, Jr, and Walter H. Kohler. Debugging a Distributed Computing System. *IEEE Transactions on Software Engineering*, 10(3):210–219, March 1984.
- [5] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring Distributed Systems. *ACM Transactions on Computer Systems*, 5(2):121–150, May 1987.
- [6] David W. Krumme, Alva L. Couch, and George Cybenko. Debugging Support for Parallel Programs. In Jack Dongarra, Iain Duff, Patrick Gaffney, and Sean McKee, editors, *Vector and Parallel Computing: Issues in Applied Research and Development*, pages 205–214. Ellis Horwood Limited, Chichester, 1989.
- [7] Thomas Kunz. Issues in Event Abstraction. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *Proceedings of PARLE '93: Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, pages 668–671, Munich, Germany, June 1993. Springer-Verlag.
- [8] Thomas Kunz. Process Clustering for Distributed Debugging. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 75–84, San Diego, California, May 1993. Appeared as *ACM SIGPLAN Notices*, 28(12), December 1993.
- [9] Thomas Kunz. An Event Abstraction Tool: Theory, Design, and Results. Technical Report TI-1/94, Technical University Darmstadt, January 1994.
- [10] Thomas Kunz. Abstract Behaviour of Distributed Executions with Applications to Visualization. Ph.D. Thesis, Technical University Darmstadt, May 1994.
- [11] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.
- [12] Reinhard Schwarz and Friedemann Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. Technical Report SFB124–15/92, Sonderforschungsbereich 124, Universität Saarbruecken und Universität Kaiserslautern, December 1992.
- [13] Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Bill Silvermann, Daniel Yellin, Jim Russell, and Shaula Yemini. Hermes: Unix User's Guide, Version 0.8alpha. Technical report, IBM T.J.Watson Research Center, Yorktown Heights, New York, USA, March 1992.
- [14] David J. Taylor. A Prototype Debugger for Hermes. In *Proceedings of the 1992 CAS Conference, Volume I*, pages 29–42, Toronto, Ont., Canada, November 1992. IBM Canada Ltd. Laboratory, Centre for Advanced Studies.
- [15] David J. Taylor. The Use of Process Clustering in Distributed-System Event Displays. In *Proceedings of the 1993 CAS Conference*, pages 505–512, Toronto, Ont., Canada, October 1993. IBM Canada Ltd. Laboratory, Centre for Advanced Studies.
- [16] Dror Zernik, Marc Snir, and Dalia Malki. Using Visualization Tools to Understand Concurrency. *IEEE Software*, 9(3):87–92, May 1992.