

High-Level Views of Distributed Executions

Thomas Kunz

Department of Computer Science
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1
e-mail: tkunz@uwaterloo.ca

Abstract. Due to 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 discusses some of the problems encountered when displaying executions at abstract levels and presents a graphical representation for *convex* abstract events. Using convex events as building block for abstract execution visualizations avoids the identified problems. The proposed representation can easily be included in the process-time diagrams frequently employed to depict the behaviour of distributed applications. Such visualizations, in turn, are helpful during the construction, debugging, and monitoring of distributed applications as well as in trying to understand old “legacy” code in a program-understanding task. We enhanced the visualization component of a prototype distributed debugger with the facility to depict executions at various abstraction levels. Examples of the resulting abstract visualization for the execution of a non-trivial distributed application are discussed. These abstract visualizations are essential to minimize the complexity of the understanding process, and support top-down debugging.

1 Introduction

Empirical evidence shows that programmers spend the majority of their time on program maintenance tasks such as debugging [6]. 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 [3]. *Distributed applications* are one class of applications with a high degree of complexity [10, 14]. 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. 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.

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 suc-

cessively moves to the next lower level. In the context of program understanding and debugging, a hierarchy of abstract behaviour visualizations can be used to minimize the complexity involved. Typically, a programmer starts the debugging task by trying to gain a global overview [15]. This can be 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 the bug is tracked down. 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.

Suitable 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 research focuses on the development of tools to derive suitable abstraction hierarchies automatically, as described in [16, 18, 20]. This paper discusses some of the problems and proposed solutions for *visualizing* the behaviour at various abstraction levels.

The sample application discussed in this paper is written in Hermes, a high-level, process-oriented language for distributed computing [26]. Hermes was chosen from the large number of distributed programming languages for a number of reasons. First, following [12], a complete programming model for distributed applications consists of two separate pieces: the computation model, which allows programmers to build a single computational activity, and the coordination model, which binds separate activities into an ensemble. In Hermes, this coordination model is its process model. Hermes processes are created and terminated dynamically and communicate and synchronize by synchronous and asynchronous message passing. In other words, 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.

The paper is organized as follows. Next, a visualization frequently employed to depict the execution of a distributed application is presented. We will discuss some related work and point out some of the corresponding problems. We then introduce the concept of *convex* abstract events and justify the graphical representation chosen for them. Non-overlapping convex abstract events are the basic building block of the automatic event abstraction tool described in [18]. Abstract visualizations build from convex events avoid the identified problems. Using a non-trivial distributed application, an example for abstract execution visualizations is given. The paper finishes with a summary of our work and suggestions for possible future work.

2 Visualizing Distributed Executions

An *event-based* approach is typically employed for debugging distributed applications. Following [21], the execution of 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. The distributed application behaviour is frequently depicted using *process-time* diagrams [9, 25].

Figure 1 shows the visualization provided by the tool described in [27, 28]. This tool draws a set of horizontal lines, one for each process, placing a symbol on the appropriate line for each event. Time flows from left to right, and a scrollbar allows for scrolling in the vertical (process) dimension. (Scrolling in the time dimension is more complex since it depends on the actual partially-ordered execution; it has also been implemented by the tool.) The lines are preceded by the process name. Events that represent the two endpoints of a communication activity are connected by an arrow, using 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, while open circles are used for 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 invisible. 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 active, displayed by a solid line.

The basic relation between primitive events is the *happened before* relation introduced by Lamport[21], 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

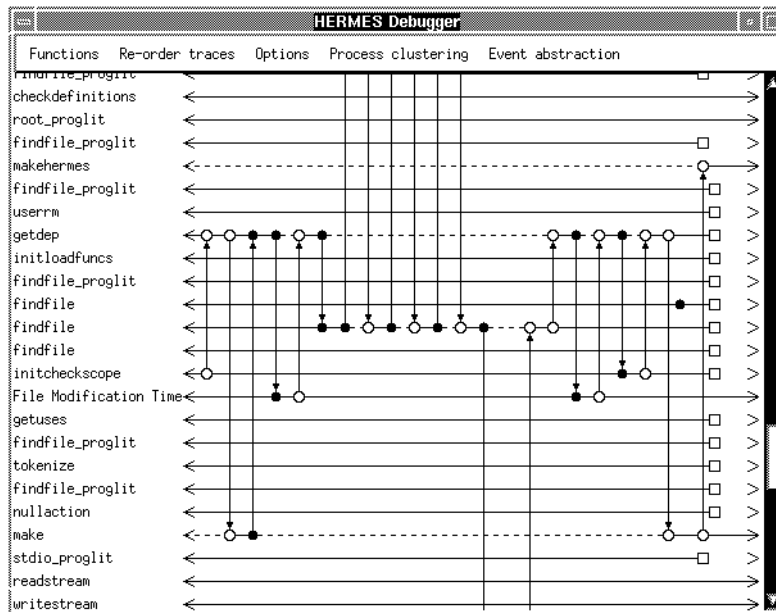


Fig. 1. A typical execution visualization

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 causality graph. Two events cannot influence each other when they are concurrent, that is, unrelated by the \rightarrow relation. In Fig. 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 [21, 25]. 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 Related Work

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.

Event abstraction is often approached in an ad-hoc, implementation-driven manner without firm theoretical foundation. Examples of early systems that support event abstraction are [4, 13]. However, as already discussed in [25], these systems allow for precedence relations to be observed at higher levels that might not hold at lower levels.

Applying more theoretic event abstraction approaches to the task of behaviour visualization can result in problematic abstract visualizations as well. In [1], the notion of molecules and atoms (sets of primitive events with certain properties) is introduced and a precedence relation on such abstract events is defined. Figure 2 gives a simple example, containing the three atoms E_1 , E_2 , and E_3 .

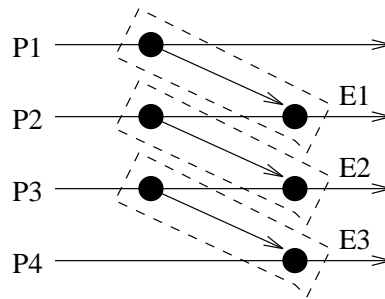


Fig. 2. Adding spurious precedence relations

According to the definition of precedence given, $E_1 \rightarrow E_2$ and $E_2 \rightarrow E_3$. This precedence relation is proven to be a partial order and therefore transitive, so $E_1 \rightarrow E_3$. However, there is not a single primitive event in E_1 that precedes any primitive event in E_3 . At the lowest observable level of behaviour, these two event sets are completely unrelated. Following [22], the system execution at the primitive level is not a valid implementation of the system execution at the abstract level, or, vice versa, the system execution at the abstract level is not a valid specification of the system execution at the primitive level. The usefulness of the resulting abstract views when trying to debug a distributed execution is therefore questionable.

Zernik et al. [29] discuss abstraction on a causality graph, representing the execution of a distributed application. This causality graph is constructed from a few basic node and edge types. The basic model of concurrency supported is the fork-join model of parallelism, threads communicate by message passing using send and receive primitives. Figure 3 gives an example.

The paper presents an abstraction algorithm to collapse nodes (e.g., events) into supernodes (e.g., abstract events). To avoid cycles, a supernode is not represented by a single node. Instead, a set of message-send and message-receive nodes is chosen by the algorithm as a minimal representation of the supernode. Furthermore, the abstraction algorithm assumes that the “natural” scope for abstraction is a branch that includes the node. In Fig. 3, for example, the abstraction scope for node X is the subgraph delimited by the fork node A and the join node B , consisting of

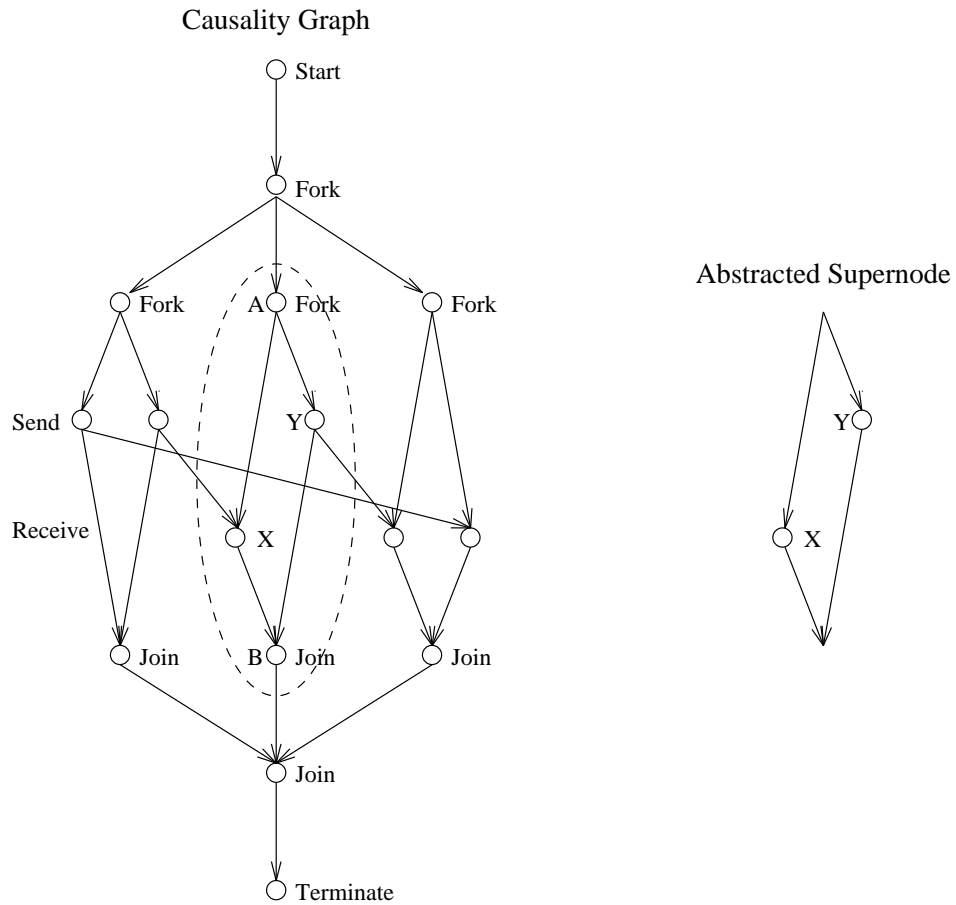


Fig. 3. Abstraction in a causality graph

all nodes contained in the dashed ellipses. However, when applying the abstraction algorithm given to this subgraph, no complexity reduction is achieved at all.

Our approach allows to collapse all nodes in this specific subgraph into *one* supernode and to represent this supernode by *one* entity. Furthermore, we do not restrict the abstraction operation to the “natural” scope. The computation and communication in a subgraph delimited by fork and join can become rather complex. A human user (or automatic tool) can derive subabstractions within such a subgraph, supporting incremental understanding of complex executions. In addition, for distributed applications not based on the fork-join paradigm, a “natural” abstraction scope might be hard to define.

4 Convex Abstract Events

We now introduce the notion of *convex* abstract events as basic building block for abstract visualizations. 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 relation between them. A disadvantage is that this relation is neither anti-symmetric nor transitive, so its reflexive closure does not determine a partial order.

Another alternative would be to define a precedence relation based on universal quantification (e.g. event set A_1 precedes event set A_2 if and only if all primitive events in A_1 precede all primitive events in A_2). Our choice is motivated by the following example, outlined in Fig. 4. Two clients want to access two different application servers. In this system, a Kerberos-like security scheme is implemented, so both clients first have to obtain tickets from the central ticket granting server TGS. From the client’s point of view, obtaining the ticket is part of the task “send request to a server”, so the corresponding abstract events E_1 and E_2 contain both the communication with the ticket granting server and the desired server. While implementing this application, the following debugging scenario might occur: the initial version of the ticket granting service is faulty. The first request overwrites some internal data while still returning a valid ticket. This bug will show up by the inability of the second client to access the FileServer because it was issued a wrong ticket. To track down the bug, we examine all abstract events that precede (and hence potentially causally influence) E_2 . With universal quantification, E_1 does not precede E_2 and will therefore not be examined, making debugging a lot harder.

Weak precedence relations, such as the one defined in Def. 1, cannot occur in systems where every activity involving multiple processes is encapsulated in a transaction, for example. However, distributed systems with such strong semantics are not in widespread use. More frequently, distributed system are build based on message passing packages with much weaker semantics, such as the Remote Procedure Call package provided by OSF’s DCE (distributed computing environment) [24] or the newly standardized MPI (message passing interface) [8].

Primitive events occur instantaneously in time, but arbitrary abstract events do not. We tried to preserve some of the essential characteristics of instantaneity for event sets by introducing the notion of *convexity*. Convex abstract events, can be imagined to occur at one instant in time.

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.

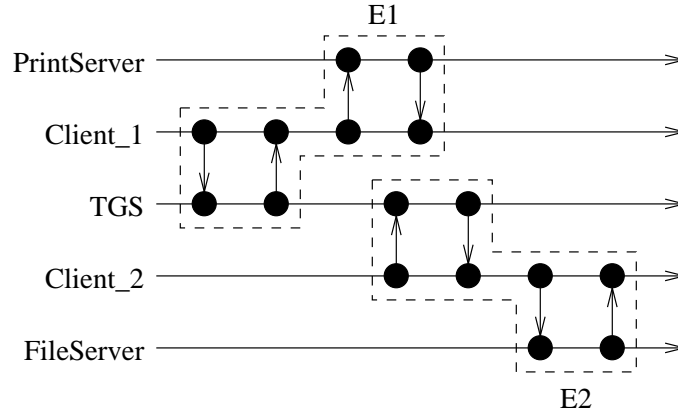


Fig. 4. Motivating the chosen precedence relation on abstract events

So far, the \rightarrow relation is defined on primitive events (Lamport[21]) 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.

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 [5]. 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 in a process-time diagram than arbitrary event sets. Finally, timestamping convex event sets can be done more efficiently than timestamping arbitrary event sets. Kunz [17] 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 no easy task. Black et al. [5] 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 [18]. 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.

Abstract visualizations based on convex abstract events avoid the two problems outlined above. The precedence relation defined does not introduce spurious prece-

dence relations. According to our definition of precedence, the two atoms E_1 and E_3 in Fig. 2 are unrelated, reflecting the fact that at the primitive level no event in E_1 precedes an event in E_3 or vice versa. Similarly, convex abstract events allow for more reduction in complexity than supernodes while at the same time supporting a more flexible modeling of the application execution. In Fig. 3, all four events A , B , X , and Y can be abstracted into one convex abstract event, whose graphical representation will be discussed next. Unlike [29], we also provide a formal definition for a precedence relation on abstract events and [17] describes timestamping algorithms and tests to deduce this precedence relation for abstract events efficiently.

5 Displaying Convex Abstract Events

The prototype distributed debugger mentioned above 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. 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, explicitly indicates the process traces involved, and can easily be integrated into the standard process-time diagrams that are displayed by the debugger. Figure 5 contains an example of the representation chosen. A discussion of various display alternatives can be found in [19].

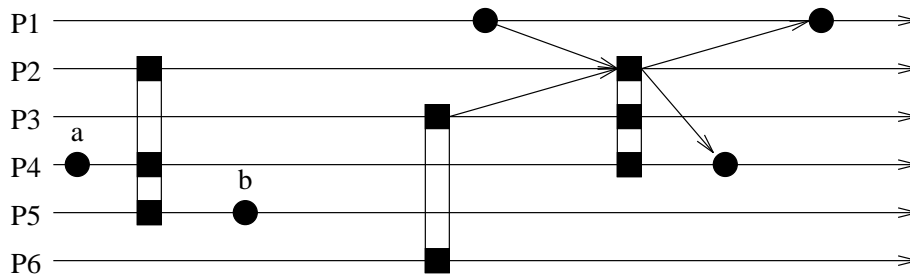


Fig. 5. A graphical representation for abstract events

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. Note that synchronous events are always either both part of a convex abstract event or not, therefore no synchronous messages will enter or leave abstract events.

Abstract events occupy the same horizontal display space as primitive events. 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 and fits into the standard debugger display, fulfilling three of the four 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 Fig. 5. 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.

6 Example

This section presents visualizations at various abstraction levels for a non-trivial distributed application. These examples indicate how the complexity of the understanding process can be reduced by the use of abstract behaviour visualizations. The process clusters and abstract events used in these visualizations are all derived automatically with the tools described in [16, 18, 20].

The execution visualized here is an execution of the `makehermes` application, Hermes' version of the Unix `make` tool. A Hermes application consists of separately compiled process and definition modules which are imported by "linking" and "usage" lists respectively. Parsing these lists in the source reveals all dependencies, and so a separate `makefile` is not necessary. `Makehermes` builds a graph structure representing the dependencies, and checks, starting from the leaves, whether a source module must be recompiled. To limit the extent of this recursive dependency check, an environment variable restricts its scope as follows: only sources in the current directory and in directories specified by the environment variable are scanned for further dependencies. In the traced execution, the dependencies for a single definition module are checked. The source modules included by this definition module are outside the current scope, that is, in directories that are neither the local directory nor in the list specified by the appropriate environment variable. `Makehermes` determines that an object file for this definition module exists and is more recent than all the source modules on which it depends. Therefore, no compiler is invoked. This execution creates a total of 175 processes, and the event trace contains 2534 primitive events.

Figure 1 depicts a segment of the execution history at the lowest abstraction level. The execution segment occurs at the end of the `makehermes` execution. For this

window size, not all relevant processes fit on the screen, as indicated by the arrows leaving the window at the top and bottom boundaries. A higher-level visualization of the same execution segment is shown in Fig. 6.

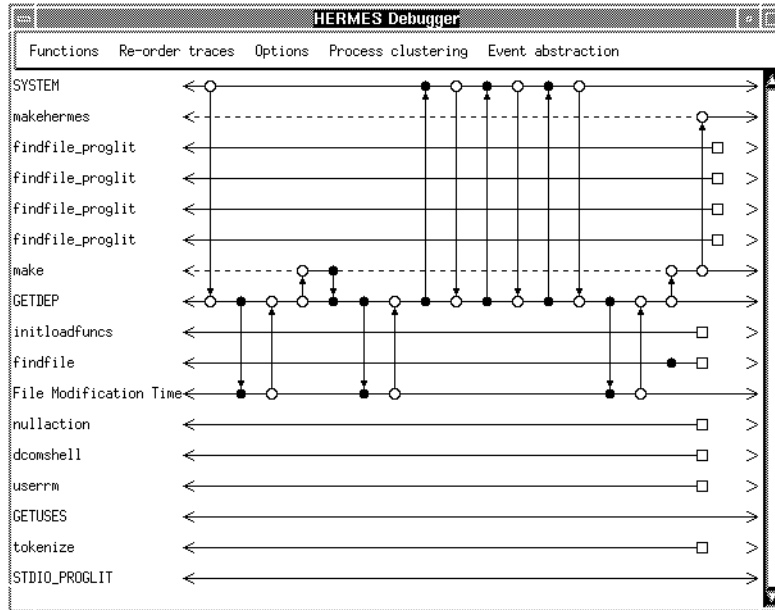


Fig. 6. Process cluster view of the `makehermes` execution

In Fig. 6, lines preceded by a name in capital letters indicate process clusters, e.g., groups of processes. On a colour display, clusters are highlighted compared to processes. Only events with a partner outside the cluster are displayed, while events purely internal to a cluster are ignored. Process clusters therefore not only reduce the display space in the process dimension, but also in the time dimension. The cluster is always represented by one or more solid lines, the display details are discussed in [28]. In this figure, all processes are shown, either by displaying them individually or as part of a process cluster.

Figure 6 shows the following sequence of actions. After scanning the source of the definition module, process `make` knows all sources this module imports. It repeatedly invokes `GETDEP` to detect the dependencies of these modules. Cluster `GETDEP` uses the process `File Modification Time` and processes within the standard Hermes runtime system to locate these sources. The sources are not within the current scope, so `GETUSES` is not invoked to scan a source file. After checking all dependencies, `make` determines that the existing object file is more recent than any of the source files on which it depends and returns to `makehermes` without invoking the Hermes definition module compiler (via process `dcomshell`).

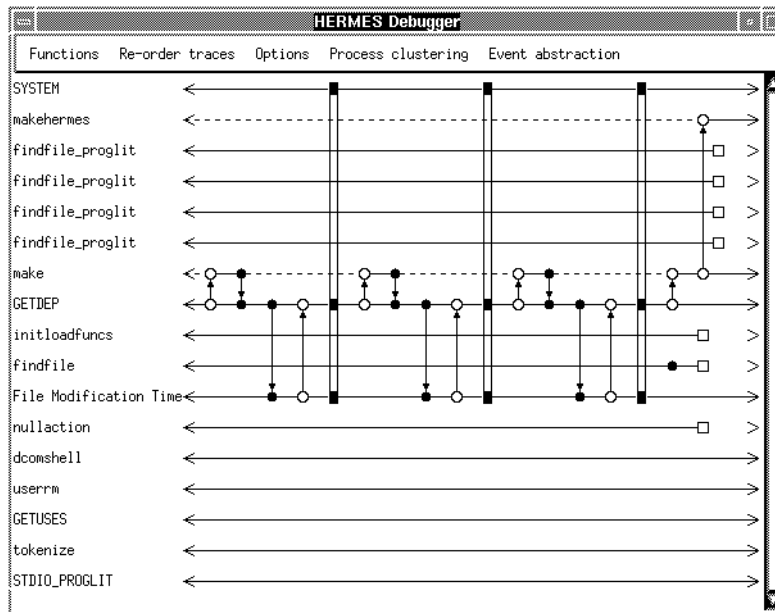


Fig. 7. Abstract view of the `makehermes` execution

Figure 7 displays the end of the execution, similar to Fig. 6, utilizing abstract events. Due to the display space saved by event abstraction, a larger piece of the execution fits on the screen. The same execution history segment without event abstraction occupies approximately two and a half windows in the horizontal (time) dimension. Figure 7 shows that `make` invokes `GETDEP` multiple times to determine dependencies for a specific source file. In each case, `GETDEP` checks whether the source file is in the current directory (the first call and associated reply to `File Modification Time`). Since this is not the case, `GETDEP` then proceeds to find the source file, obtains its timestamp, and checks whether it is in the current scope. The three abstract events in Fig. 7 group all primitive events involved in this activity. Because none of the sources imported by the target module are in the current scope, `GETUSES` is never invoked. Figure 7 displays three such iterations before `make` has exhausted all dependencies derived from the source scan. It then determines that the existing object file is up-to-date and returns to `makehermes` without invoking a compiler.

7 Conclusions and Future Work

This paper discusses some of the obstacles encountered when developing abstract visualizations of distributed executions. These higher-level views are based on the idea of event abstraction. We motivate our definition of a precedence relation on

abstract events and introduce the *convex* abstract events. A graphical representation for these abstract events is developed and added to the standard process-time diagrams frequently employed to visualize distributed executions. To depict the behaviour of distributed applications at various abstraction levels, we modified an existing prototype debugger. The execution of a non-trivial application is visualized at various abstraction levels. These abstract visualizations indeed reduce the display complexity. The resulting hierarchy of abstract views of an application execution can then be used in a top-down manner to minimize the complexity of the program-understanding and debugging task.

The abstract visualizations are firmly integrated with the other functionality provided by our distributed debugger. However, we still lack a convenient user interface for the display of complete event abstraction hierarchies. Such an interface would allow a user to work directly on the event abstraction hierarchy, to modify it manually or to navigate through the hierarchy of abstract views. 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. One of the potential solutions we are investigating is the use of fisheye views [23].

Our work to date has mostly been using Hermes as target environment. To explore the feasibility of our general approach to the visualization of complex distributed executions, we have ported the debugger to present behaviour in a number of different, widely-used distributed programming environments such as DCE. We intend to apply the abstraction tools to more realistic distributed applications and to use the results to improve our automatic abstraction tools. In particular, we are investigating how to use program slicing in a distributed program [2, 7] as a basis for automatic event abstraction.

Event abstraction will also play a dominant role in our current effort to port the debugger to PVM [11]. The basic model of a distributed execution outlined above supports only point-to-point interactions. Advanced middleware layers for distributed programming frequently offer one-to-many communication mechanisms, such as broadcasts and multicasts. To visualize these advanced features, they are modeled as a set of point-to-point communications at a lower layer and visualized as abstract events at higher layers. In this case, the tracing mechanism that collects the primitive events has to provide the data necessary to allow the automatic derivation of abstract events.

Another current project applies our previous results to the problem of network and distributed systems management. Monitoring large distributed systems produces a huge quantity of trace data. We aim to reduce the complexity when visualizing the continuing operation of such systems with our techniques.

References

1. Mohan Ahuja and Shivakant Mishra. Units of Computation in Fault-Tolerant Distributed Systems. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 626–633, Poznan, Poland, June 1994.
2. Thomas Ball and Susan Horwitz. Slicing Programs with Arbitrary Control-flow. In Peter A. Fritzson, editor, *Proceedings of the First International Workshop on Auto-*

- mated and Algorithmic Debugging*, number 749 in Lecture Notes in Computer Science, pages 206–222. Springer–Verlag, Linköping, Sweden, May 1993.
3. 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.
 4. Peter Bates. Distributed Debugging Tools for Heterogeneous Distributed Systems. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 308–315, San Jose, California, June 1988.
 5. J. P. Black, M. H. Coffin, D. J. Taylor, T. Kunz, and T. Basten. Linking Specification, Abstraction, and Debugging. CCNG Technical Report E-232, Computer Communications and Networks Group, University of Waterloo, November 1993.
 6. 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.
 7. Jingde Cheng. Slicing Concurrent Programs – A Graph-Theoretical Approach. In Peter A. Fritzson, editor, *Proceedings of the First International Workshop on Automated and Algorithmic Debugging*, number 749 in Lecture Notes in Computer Science, pages 223–240. Springer–Verlag, Linköping, Sweden, May 1993.
 8. Lyndon Clarke, Ian Glendinning, and Rolf Hempel. The MPI Message Passing Interface Standard. In Karsten M. Decker and René M. Rehmman, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 213 – 218. Birkhäuser Verlag, Basel, July 1994. ISBN 3-7643-5090-3.
 9. Colin Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):28–33, August 1991.
 10. 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.
 11. Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994. ISBN 0-262-57108-0.
 12. David Gelernter and Nicholas Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.
 13. Dieter Haban and Dieter Wybraniec. A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems. *IEEE Transactions on Software Engineering*, 16(2):197–211, February 1990.
 14. Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger. Monitoring Distributed Systems. *ACM Transactions on Computer Systems*, 5(2):121–150, May 1987.
 15. 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.
 16. 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.
 17. Thomas Kunz. *Abstract Behaviour of Distributed Executions with Applications to Visualization*. PhD thesis, Technische Hochschule Darmstadt, Darmstadt, Germany, May 1994.
 18. Thomas Kunz. Reverse Engineering Distributed Applications: An Event Abstraction Tool. *International Journal of Software Engineering and Knowledge Engineering*, 4(3):303–323, September 1994.

19. Thomas Kunz. Visualizing Abstract Events. In *Proceedings of the 1994 CAS Conference*, pages 334–343, Toronto, Ontario, Canada, November 1994. IBM Canada Ltd. Laboratory, Centre for Advanced Studies.
20. Thomas Kunz and James P. Black. Using Automatic Process Clustering for Design Recovery and Distributed Debugging, 1995. To appear in *IEEE Transactions on Software Engineering*.
21. Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.
22. Leslie Lamport. On interprocess communication. *Distributed Computing*, 1:77–85, 1986.
23. Manojit Sarkar and Marc H. Brown. Graphical Fisheye Views. *Communications of the ACM*, 37(12):73–84, December 1994.
24. Alexander Schill, editor. *Proceedings of the International DCE Workshop: DCE – The OSF Distributed Computing Environment*. Number 731 in Lecture Notes in Computer Science. Springer-Verlag, Karlsruhe, Germany, October 1993.
25. Reinhard Schwarz and Friedemann Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, 7:149–174, September 1994.
26. Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, Daniel M. Yellin, and Shaula Alexander Yemini. *HERMES: A Language for Distributed Computing*. Prentice Hall, Inc., Englewood Cliffs, New Jersey, 1991.
27. 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.
28. 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.
29. Dror Zernik, Marc Snir, and Dalia Malki. Using Visualization Tools to Understand Concurrency. *IEEE Software*, 9(3):87–92, May 1992.