Distributed Debugging in the Large

Zhonghua Yang
CRC for Distributed Systems Technology
University of Queensland, QLD 4072, Australia
Email: yang@dstc.edu.au

T. A. Marsland
Department of Computing Science
University of Alberta, Edmonton, Canada T6G 2H1
tony@cs.ualberta.ca

Abstract

A notion of distributed debugging based on partially ordered events is presented. Both the execution model of a distributed program and an order-preserving mechanism supporting this notion are formally examined. In this paper, we discuss the fundamentals of distributed computation for debugging and identify the atomic process interaction group as a higher level view of the partial ordering model of program execution. This atomic process interaction group exists inherently in a computation and has the distinct properties of indivisibility and independence, with no communication taking place between groups. We claim that the atomic process interaction group is a unifying mechanism for event and process abstraction and advocate it as a basic debugging unit. Finally, an algorithm to detect the atomic groups as they occur during a computation is outlined.

1 Introduction

The difficulties in debugging distributed programs (distributed debugging for short) are well recognized, and include:

1. The complexity of the control flow of a distributed program. Accordingly, the interactions between component processes of the program become very intricate.
2. Communication with an unpredictable delay. This makes it difficult to obtain and maintain a global state and time of the computation.
3. Nondeterminism, an inherent feature of distributed systems, asynchronous systems in particular. For example, the order of event occurrence in a computation cannot easily be determined. A program that works correctly one time may fail later if the duration of events changes. From a debugging viewpoint, this means that reproduction of the execution results requires serious effort.
4. The probe effect. Any attempt to trace the execution of a program may interfere with the program's behavior, which may further contribute to the difficulties of obtaining reproducible execution results. How much this intrusiveness affects the computation in progress is the "Uncertainty Principle" as applied to debugging: there is no precise answer.
5. Information explosion. Any kind of trace for a distributed program will potentially generate volumes of information that must be conveyed to a programmer in a comprehensible manner.

These difficulties provide convincing evidence that new techniques and tools are required for distributed debugging. In providing such facilities and tools, four main approaches have been taken.

- Use of a collection of sequential debuggers. Although not able to handle time-dependent errors, this scheme works well at a lower level, i.e., at the instruction level or at the procedure level. Most available commercial debuggers fall into this category.
- Centralized debugging. When developing a distributed program, all the processes are first run on a single processor. Once the program has been debugged, the processes can then be run on the separate processors. Although this approach is sometimes convenient, many problems may not be easily revealed because of differences between debugging and actual execution run-time environments.
- Use of event-based debugging. A distributed debugger views the program execution as a sequence of events. During the execution, an event history

(or trace) containing all the events generated by the program is recorded for browsing or replay. Most research projects in distributed debugging take this approach.

- By static analysis. This differs from a formal proof of program correctness, which is explicitly excluded when dealing with distributed debugging. The approach uses dataflow analysis to detect synchronization errors and race problems.

The design and implementation of a distributed debugger is divided into two categories:

- Operating system service-based design and implementation. The distributed debugger explicitly invokes a set of operating system and networking services. In other words, the support for debugging is provided by the operating system and networking services. This design and implementation can support several languages, but provides a low level description of program behavior.

- Language-based design and implementation. Language based debuggers use high-level application programming language facilities to support debugging. It is, therefore, system independent.

Although the debugging of distributed programs has been gaining research interest in recent years, as witnessed by several special workshops [10, 9], the works have focused mainly on the practical aspects of providing a user with special functions and tools. Rarely have the fundamental issues underlying the design and implementation of these facilities and tools been given a penetrating exposition. These issues include the program execution model, the partial ordering view of events in a computation, and partial order preserving mechanisms. This paper devotes itself to the some of them. However, since the partial ordering view of events in a computation is the cornerstone of our discussion, we first give a brief justification of *why partial orders*? Its full account and arguments can be found elsewhere in the literature [12].

1.1 Why partial orders?

In recent years, modeling concurrency with partial orders has been advocated by several researchers. Just as linear ordering naturally models sequential computations, partial orders of events are a natural and normal model for distributed computation. As stated eloquently by Pratt,

> "The undefinedness of 'global simultaneity' is not merely a relativistic curiosity, it is a practical engineering problem. Given two microprocessors on two ocean liners 100 miles apart, the relative temporal order of execution of individual instructions on these two machines is for all practical purpose undefined. As another example, two fragmented packets arriving at the same ports overlapped in time have their relative order undefined" [11].

Obviously, from a debugging viewpoint, the problem of observability makes reliance on a total order undesirable and inconvenient. If two unrelated events occur on two separate computers, a total ordering of these events is unnatural and artificial because the exact order is simply undetermined. Thus, if an execution of the distributed program exhibits an abnormal behavior, and problems are suspected to exist, then a total ordering of event occurrence in the execution is of little help in locating the problems. Rather, we need to determine the causal relation between an event which manifests the erroneous behavior (the effect) and an event which causes the problem (the bug), and this causal relation precisely defines partial orderings of a computation. We believe that this understanding captures the fundamental nature for distributed debugging and is a main theme of this paper.

Here we will examine the idea of partial orders of events from a distributed debugging standpoint. While doing so, we consider the problem from two different levels of abstractions. At the higher level, we identify a possible problem area where the bugs might reside. However, the precise place of the bugs remains unknown, and is left to a lower level debuger. These areas are called (atomic) process interaction groups, and will be defined in Section 3. This kind of debugging is called *distributed debugging in the large*, as opposed to the distributed debugging in the small (at a lower level), where the detailed debugging task is carried out. In this paper, we only discuss the former.

In the remainder of this paper, we give an overview of the program execution model based on a partial order relation, then provide a comprehensive treatment of logical clocks and a virtual time mechanism for maintaining this partial order. Finally, partial ordering based debugging is presented, by analyzing the distinct behaviour feature of a distributed program and conducting debugging at different levels of detail.

2 The execution model of distributed programs

A distributed program is modeled by $D = \{P, C\}$, where P consists of a finite set of processes, $P = \{p_1, p_2, \cdots, p_n\}$, that run at one or more nodes connected by a communication network. These processes have a disjoint address space and communicate with each other solely by message passing via a finite set of communication channels, C. These processes may also perform operations independently and concurrently, and cooperate with each other to achieve a common goal. The dynamics of each process p_i is characterized by a finite set of three kinds of events on that process: *a send event, a receive event* and *an internal event.* A send event occurs when the process transmits a message to another process, and is the cause of an occurrence of the receive event on the destination process. The communications between processes are one-to-one, and the send and receive events are corresponding *communications* within a system. An internal event causes a local state change in the process on which it occurs. A (possibly infinite) sequence of all the events in a process constitutes the *local history* of the process, denoted as $h_i = e_i^1 e_i^2 e_i^3 \cdots$, and a *global history* of the computation is the union of the local histories of all the processes containing all of its events, $H = h_1 \cup h_2 \cup h_3 \cup \cdots \cup h_n$. Since an event generally changes the state of a process (and a computation), we will use the term event and state interchangeably wherever there is no confusion. These events and the partial order relationship between them (see Definition 1, later) characterizes the behavior of distributed programs, and have two distinct attributes:

- Locality, that is, events occur in a small area and over a short period of time.
- Atomicity, that is, an event has either occurred or has not. In other words, it has either left some *effects* or nothing during the computation.

In monitoring the execution of a distributed program, we ignore those processes that have no events in any computation, and also those processes that have only internal events without communications, since they are considered to be pure sequential programs. We assume that the facilities and tools for debugging sequential programs are adequate. We denote E_i as an event set on a process p_i, E a set of events within a system, and $e_{i,k}$ is the kth event on the process p_i. It is assumed that there are n internal events called the initial event, $e_{i,0}$, one on each process. For simplicity, we sometimes also use e_k to represent an event in the system when the process on which the event occurred is irrelevant. For convenience, each process is assumed to have a unique identifier, as we have already done above.

Note that this model forms a high level of abstraction of a distributed program. It abstracts away the physical organization of the program (e.g., process – processor mapping) and the particular details of the supporting communication network, it even abstracts the details of the distributed programming environment. It can be further noted that the whole state information about program executions consists of the following two parts [14]:

- Process states — which are similar to state information about the sequential programs.
- Channel states — the sequence of messages in transit on the channel when the state information is taken.

It is believed that in a highly distributed computation, relations between events become important. What is required in debugging such programs is to catch the significant events and understand how the occurrence of an event causally depends on the previous occurrence of others. Intuitively, our objective is to find out the *causes* of the bugs, once a program execution exhibits misbehavior (a fault). For example, a send event of a process would depend on it first performing either an internal computation event or a receive event, which in turn would be *caused* by some other event, including a send event by another process. This view of a distributed computation as an event occurrence together with the *causality* relation is along the lines suggested by Lamport [6], and leads to a definition of program execution as a partial ordering of an event occurrence.

Definition 1 *The execution of a distributed program is represented by* $(E, \longrightarrow)$*, where* E *is a finite set of events, and* $\longrightarrow \subseteq E \times E$ *is the "happen before" relation [6], defined to be the smallest relation for which* $e_{i,s} \longrightarrow e_{j,t}$ *if*

1. $i = j$ *and* $s < t$ ($e_{i,s}$ *comes before* $e_{j,t}$)*,*
2. $i \neq j$ *and* $e_{i,s}$ *is a send event and* $e_{j,t}$ *is the corresponding receive event.*
3. $\exists e_{k,u} \in E, e_{i,s} \longrightarrow e_{k,u} \wedge e_{k,u} \longrightarrow e_{j,t}$*. That is, the "happen before" relation is transitive.*

Certain events in the execution of a distributed program may be causally unrelated. These events are said to be *concurrent* (or incomparable). Formally, for two distinct events e_1 and e_2, if $e_1 \not\longrightarrow e_2$ and $e_2 \not\longrightarrow e_1$, then $e_1 \parallel e_2$.

For convenience, and stressing the causal relation between communicating events, we define the following mapping:

Definition 2 *For a send/receive event e_i, and its corresponding receive/send event e_j, the mapping $M(e_i) : E \longrightarrow E$ is defined as $M(e_i) = e_j$ and $M(e_j) = e_i$.*

Obviously, the observed behavior of the execution of a computation must be "meaningful". A form of generalization of "meaningful" is the notion of *consistent cut*. A cut of a distributed computation is a subset C of its global history H and contains an initial prefix of each local history, denoted by $\{e_i | e_i$ is the last event in the prefix for each process$\}$. Clearly, a consistent cut is one obtained by an omniscient external observer who can observe simultaneously the events in each process. In a system where there is no global time, the consistent cut is defined to be one that respects the causal relation, in other words, a cut C is consistent if for all events e and e',

$$(e \in C) \wedge (e' \in C) \Rightarrow e' \in C$$

that is, a consistent cut is left closed under the causal relation.

Note that this view of computation at the event level is fundamental to "meaningfully" capture its behavior. However, in the analysis, design, and debugging of the distributed system, it is often required to create multiple viewpoints on a system, and to be able to view the system at different levels of abstraction. To make this requirement more convincing, we introduce the following definitions, taken from Lamport's development [7]:

Definition 3 *A higher level view of a distributed program execution $(E, \longrightarrow)$ is $(E^*, \stackrel{*}{\longrightarrow})$ such that:*

H1. *E^* partitions E, and is nonempty.*

H2. *$G_1, G_2 \subset E^*$: $G_1 \stackrel{*}{\longrightarrow} G_2 \Longleftrightarrow \forall a \in G_1, b \in G_2 : a \longrightarrow b$.*

From a distributed debugging perspective, if E is taken to be the execution of the high level language statements, and E^* is treated as the execution of the steps of a particular algorithm implemented by the code of the statements, then E^* forms a higher level view of E. In Section 3, we identify the partitions E^* and its subset G on which a higher level view about the computation can be based, and in this way, debugging can be safely carried out at the different level of abstraction.

The mechanism we use for maintaining the event ordering is the *vector clock* [8, 4] using Lamport's notion of causality. A vector clock, V, is represented by an n-component vector $V(e_i)[1..n]$. As it occurs, each event e_i is timestamped by the value of $V(e_i)$ (also called *vector time*). The vector clock ticks based on the causal relationship between the events, and thus the components $V(e_i)$ have the following value:

- $V(e_i^t)[i] = t$, where t is the number of events that process p_i has executed up to and including e_i.
- $V(e_i^t)[j], j \neq i$ is the number of events p_j has executed that causally precede e_i^t.

A straightforward implementation of the vector clock has each process p_i maintaining a vector of n-components, V_i, and advance the component as follows:

- If e_i is a local or a send event, then $V(e_i) = V(e_i) + 1$.
- For a send event, process p_i tags the message with its *timestamp*, $V(e_i)$.
- If e_j is the receive event, the process p_j takes a pair-wise maximum of its own vector and the message timestamp.

Vector time has several interesting properties which adequately characterize causality [13, 2]. In particular, we have:

1. $e \rightarrow e'$ if and only if $V(e) < V(e')$,
2. $e \parallel e'$ if and only if $\neg V(e) < V(e')$ and $\neg V(e') < V(e)$,

We require that the timestamp tagged with each message also contains the origin and destination of the message. Thus, the problem of determining the casual precedence can be further simplified. That is, for two events $e \in E_i$ and $e' \in E_j$, we have:

1. $e \rightarrow e'$ if and only if $V(e)[i] \leq V(e')[i]$,
2. $e \parallel e'$ if and only if $V(e)[i] > V(e')[i]$ and $V(e')[j] > V(e)[j]$,

Now we are ready to examine the behavior of the program execution from a distributed debugging perspective.

3 A higher level debugging unit of distributed computation

Earlier, we stated that there is no reason to interleave the instruction streams of processes in a computation to form one stream. Instead, from a debugging viewpoint, it is much more convenient, both intuitively and conceptually, to juxtapose these streams and to add interprocess communication in between. In particular, a distributed debugger has the following view of a computation:

- Each process carries out its computation sequentially;
- When necessary, a process communicates with other processes via the interprocess communication primitives, e.g., those provided in high level languages;
- Because of the locality attribute of events, a process does not communicates with all the processes in any reasonably short period of time. This communication pattern of process interactions can be generally identified in a distributed computation.

This partial ordering view of a computation suggests that we establish our strategy for building the debugger on the basis of the differences between distributed and sequential computations. This also is the principle of separation of concerns. A distributed debugger need only concentrate on the interaction of processes, leaving debugging of the sequential phase to a standard debugger. To stress this insight, we introduce the concept of a Process Dependent Diagram. A Process Dependent Diagram (PDD) is an abstraction of the dynamic behavior of a distributed program, and shows the interactions between processes while hiding the detail of internal events within a process.

In a large scale computation, the PDD can become very big and intricate, so we must restrict it to a manageable size. This can be done by a so-called *restriction of the computation*, denoted by Z/(P', E'). It is the result of hiding away all processes and events not in $P' \subseteq P$ and $E' \subseteq E$. We regard the restriction as a means of managing the complexity and intricacy of a distributed program's behavior. Thus, we can define a higher level view of process interaction — the Process Interaction Group (PIG) — as a restriction unit, where a pair of send and receive events is a primary process interaction unit. Informally, a Process Interaction Group (PIG) G is a nonempty set of events which have occurred between any two consistent cuts. In addition, if a send/receive event e_i belongs to G, then the corresponding event $M(e_i)$ also belongs to G.

Thus, the PIG begins with a consistent cut. However, requiring that a pair of send and receive events be in the same group makes the notion of a PIG stronger than a consistent cut. Groups are formed when a computation is quiet, and process interactions only take place within groups (hence the name). We believe that methodologically the PIG can be used as a debugging unit to provide a higher level starting point for debugging. In fact, it provides another way of viewing the behavior of the execution in the sense that the only way the rest of the computation affects the group is in determining the process state at the beginning of the group execution, and conversely, the only way the group affects the rest of the computation is in determining the process state at the end of the group execution.

To further keep the PIG size manageable, we define the *atomic process interaction group* as the smallest process interaction group which cannot be further divided. An atomic *PIG* possesses indivisibility in the sense that it is any process interaction group that does not have another group as its proper subset. Obviously, each atomic PIG is a process interaction group and inherits the general properties of a process interaction group, but not vice versa. Furthermore, an atomic *PIG* enjoys great computational independence, and can be considered as an *isolated* unit, since no message passes between groups and every message sent must be received in the same group. In addition, they partition a computation, and never overlap.

For convenience, we assume that there is an initial atomic PIG which is the set of initial events on all processes in the system, that is,

$$g_0 = \{e_{i,0} \mid \forall i \in \{1, 2, ..., n\} \text{ and } e_{i,0} \textit{ is an initial event on } p_i \}$$

The independence of a group and the indivisibility of an atomic group qualify them as a suitable computational unit for debugging purposes. As mentioned above, a restriction of a computation may be imposed in terms of the group. The group is the unity of the process abstraction and event abstraction in the following sense:

- Grouping communicating processes as a unit abstracts away how the processes communicate, hiding all communication activities among themselves from the outside, and conducting no interaction at all with processes outside the group.
- A process can be executing within at most one atomic process group at any one time. This fea-

ture is what we want when we are debugging: one group, one process, at one time.

- The whole group, as an *isolated* unit, behaves as a single event and abstracts away the internal structure of the group. Group is truly *an event* in the computation. Progress of a computation is from one group to the next and, as defined below, the partial order relation is maintained among groups.

This unifying abstraction of processes and events is necessarily based on our program's execution model. Recall that only events and their relation appear in this model, nothing else, and thus process abstraction must be expressed (embodied) in terms of event abstraction. We believe that our view of process and event abstraction, i.e., *quiet* external to the group and *clamor* internal to the group, is simpler and neater conceptually, and is more practical, since a simple and well-defined interface would be considered as one of the criteria for modularization.

Since the group is formed along the consistent cut, there exists a "*precedence*" relation between groups. Formally, for two atomic process groups g_1 and g_2, the g_1 "precede" the g_2, denoted as $g_1 \longrightarrow g_2$, if and only if $\exists e_i \in g_1, e_j \in g_2, e_i \longrightarrow e_j$. Two atomic groups are concurrent if $g_1 \not\longrightarrow g_2$ and $g_2 \not\longrightarrow g_1$.

Having defined the *precede* relation between atomic groups, we can establish the following theorem, presented without proof.

Theorem 1 *The "precede" relation between atomic groups is a strict partial ordering.*

The fact that the "precede" relation between atomic groups is a strict partial ordering ensures that as the program executes, the atomic groups generated, plus the relation between them, form a directed acyclic graph (DAG). This DAG of atomic groups depicts intuitively the behavior of the execution of a computation, and captures the manner in which the computation progresses.

With the abstraction power of the process interaction groups, we propose the two debugging concepts: qualitative debugging and quantitative debugging.

- *Qualitative debugging.* Debugging is done at a higher atomic group level. At this level, the relationships between groups are inspected. In addition, one can see what a group is affected by or affects another group's behavior. The exact details of how processes interact within a group is of no concern. Qualitative debugging is also called debugging-in-the-large.
- *Quantitative debugging.* If something is found to be wrong in the qualitative debugging (which is very coarse), and one or more groups are suspected to be the offenders, we can make a closer examination of the particular groups, one group at time, ignoring the rest of the execution. Quantitative debugging is also called debugging-in-the-small.

The whole debugging process can start with the qualitative debugging at the level of the groups, and then quantitative debugging, progressing from a higher level to a lower level with more and more details of computation exposed until the bugs are located.

4 Detecting atomic groups

In the preceding section, we identified *atomic process interaction groups* as a unifying abstraction of events and processes, which inherently exist in a distributed computation and serve as a debugging unit. This section outlines how we can detect the atomic groups as they occur during a computation based on the properties the atomic group possesses.

We assume that there exists a *debugger* process in the system, which could be either one of processes (designated for debugging support) of the distributed program being debugged, or a specialized process solely serving the debugging purpose. The debugger process maintains a queue of events for each process. As the program is executing, the process sends the event information to the debugger process. These events are enqueued in ascending vector time order in the corresponding process queue.

With this data structure the protocol of detecting an atomic group, g, by the debugger process is outlined as follows:

(a) Periodically, find the consistent cut in the event queues maintained by the debugger process, i.e., find a set of events $C_m = \{e_1, e_2, \cdots, e_n\}$ satisfying the following condition:

$$\forall i, j, V(e_i)[i] \geq V(e_j)[i].$$

(b) By comparing the vector time of the events in the queues, obtain a smallest set g of those events that have occurred between any two consistent cuts, C_m.

(c) For any event $e_t \in g$, include $M(e_t)$ into g.

Detecting atomic groups is often done in conjunction with the detection of *breakpoints*. A breakpoint in the

computation is a *predicate* ϕ defined on the consistent global state. Thus,

(d) While the debugger process detects an atomic group, g, it also reaches the breakpoint ϕ, if ϕ holds on the group g, that is, $g \models \phi$.

The soundness of this protocol is straightforward — the event set found by our protocol forms an atomic group; the global state reflected in the group is consistent, and communication event pairs are in the same group. The smallness ensures the atomicity of the group.

We now argue that the protocol is complete as well; that is, if the group comes to its existence during the computation, then our protocol will detect it. Suppose that as the computation makes progress and an atomic group has formed (we just argued that an atomic group inherently exists in a computation), the events which constitute the group will be eventually sent to the debugger to be enqueued. Then, by executing our protocol, the debugger process will detect them as an atomic group.

Breakpointing in distributed computation is a constant issue. Problems may well exist if a predicate requires the evaluation of a global state covering several atomic group. If this is the case, we may have to relax the definition of the atomic group — rather than the smallest (atomic) group, A larger unit — a *molecule* group may be defined. Nevertheless, a detailed elaboration is beyond the gist of this paper, and will be left to a future work.

5 Related works

The idea of observing the behavior of a distributed computation along the line that no communication is taking place between the processes is closely related to the works by other researchers, but from different perspective and settings. Fischer *et al.*[5] consider the global state of a distributed transaction system in which a global state is *consistent* if no transactions are in progress. Following a similar line of thinking, Ahuja *et al.*[1] propose as the basic computational unit in distributed systems states where there are no messages in transit, while Elrad and Francez[3] suggest decomposing the distributed program into communication-closed layers, to simplify the program analysis and verification. Our work, while based on the similar idea, is developed in the context of distributed debugging, and provides a different treatment and algorithm.

6 Conclusion

We have presented the notion of distributed debugging based on the partially ordered events, which is supported by the execution model of distributed programs and the partial order preserving mechanism (vector time). We have shown that the program execution model, represented by a set of events and a partial order relation between events, describes the inherent nature of a program's behavior. We stress that the notion of partial-ordering-based distributed debugging derived from this model captures the behavioral characteristics of a distributed program, and is therefore more natural and convenient.

Any concept or idea, if it fails to deal with the complexity of the problems to be solved, would have little use in practice. From distributed debugging perspective, the partial ordering based notion must be developed in such a way that it can support multiple views of the program execution. To this end, we have identified the atomic process interaction group as a higher level view of a partial ordering model of the program execution, which inherently exists in a computation and serves as a unifying mechanism for event and process abstraction. We advocate that the debugging unit be an atomic group with a clear interface that requires no communication taking place between them.

The protocol outlined for detecting atomic groups is straightforward. The technical tool used is vector time as the timestamps of an event. Combining the notion of process interaction groups with breakpointing may facilitate the task of distributed debugging. However, how to specify breakpoints in the framework as presented in this paper requires further research.

Acknowledgements

Much of the research by the first author as reported in this paper was conducted while he was visiting the University of Alberta, and was supported by a Canadian NSERC Grant held by the second author. The work by the first author has also been funded in part by the Cooperative Research Centers Program through the department of the Prime Minister and Cabinet of the Commonwealth Government of Australia.

References

[1] Mohan Ahuja, Ajay D. Kshemkalyani, and Timothy Carlson. A Basic Unit of Computation in Distributed Systems. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 12–19, 1990.

[2] Ö. Babaoğlu and Keith Marzullo. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In S.J. Mullender, editor, *Distributed Systems*. chapter 4. ACM Press, 1993.

[3] T. Elrad and N. Francez. Decomposition of Distributed Program into Communication-Closed Layer. *Science of Computer Programming*, 2(3):155–163, December 1982.

[4] C. J. Fidge. Timestamps in Message-Passing Systems that Preserve Partial Ordering. In *Proceedings of 11th Australian Computer Science Conference*, pages 56–66, February 1988.

[5] Michael J. Fischer, Nancy D. Griffeth, and Nancy A. Lynch. Global States of a Distributed System. *IEEE Transactions on Software Engineering*, SE-8(3):198–202, May 1982.

[6] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[7] Leslie Lamport. On Interprocess Communication: Part I Basic formalism and Part II Algorithms. *Distributed Computing*, 1(1):77–101, 1986.

[8] Friedemann Mattern. Virtual Time and Global States of Distributed Systems. In M. Cosnard and P. Quinton, editors, *Proceedings of International Workshop on Parallel and Distributed Algorithms (Chateau de Bonas, France, October 1988)*, pages 215–226, Amsterdam, 1989. Elsevier Science Publishers B. V.

[9] Barton Miller and Thomas LeBlanc, editors. *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, University of Wisconsin, Madison, May 5-6 1988.

[10] Barton P. Miller and Charles McDowell, editors. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging, Published in ACM SIGPLAN NOTICES Volume 26 Number 12, 1991*, Santa Cruz, California, May 21-22 1991. ACM Press.

[11] V. R. Pratt. On the Composition of Processes. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Language*, pages 213–223, January 25-27 1982.

[12] Vaughan Pratt. Modeling Concurrency with Partial Orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[13] R. Schwarz and F. Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. Technical Report SFB 124-15/92, Department of Computer Science, University of Kaiserslautern, December 1992.

[14] Z. Yang and T. A. Marsland. Global Snapshots for Distributed Debugging. In *Proceedings of 4th International Conference of Computers and Information (ICCI'92)*, pages 436–440, Toronto, Canada, May 28-30 1992. IEEE Computer Society Press.