CS-2001-01

ISSUES IN SCALABLE DISTRIBUTED-SYSTEM MANAGEMENT

Paul A.S. Ward
pasward@shoshin.uwaterloo.ca

Shoshin Distributed Systems Group
Department of Computer Science
University of Waterloo, Waterloo Ontario N2L 3G1, Canada
http://www.shoshin.uwaterloo.ca/˜pasward/Tech-Reports/CS-2001-01/

January 22, 2001

ABSTRACT

Distributed-system management concerns the observation of a distributed computation and then using the information gained by that observation to control the computation. This necessitates collecting the information required to determine the partial order of execution, and then reasoning about that partial order. This in turn requires a partial-order data structure and, if the reasoning is being performed by a human, a system for visualizing that partial order. Both creating such a data structure and visualizing it are hard problems.

Current partial-order data structure techniques suffer various shortcomings. Potentially scalable mechanisms, such as Ore timestamps, are static. Dynamic algorithms, on the other hard, either require a significant search operation to answer basic questions, or they require a vector of size equal to the width of the partial order for each element stored in the order.

Scalable visualization of a partial order is hard for the same reasons that drawing any large graph is hard. Any visualization that will be meaningful to a user requires appropriate abstractions on the data structure, while preserving the core meaning of the data structure. Such abstraction is difficult.

This report formalizes these problems and identifies the specific difficulties that must be solved to enable scalable distributed-system management.

Keywords: distributed-system observation, partial-order data structure, vector timestamp, data-structure visualization, scalability

CONTENTS

FIGURES

1 INTRODUCTION

We wish to build and maintain large distributed systems, where “large” should be understood to mean systems containing many concurrently-executing processes. In the course of building and maintaining such systems we will wish to observe computation executions and, based on analyzing those observations, debug code and influence, correct or steer executions. In summary, we wish to know in a precise, technical sense what is going on under the covers of a distributed system, and to use this knowledge to control the execution of said system. This is a hard problem.

One of the things that makes this problem hard is the events that form the distributed computation which executes over the distributed system are not totally ordered. Rather, they form a partial order. The ability to display, manipulate, reason about, and query this partial order is the essence of distributed observation and control. An example of a tool that provides some of these facilities is POET [44, 69]. An example of a POET display, presenting a binary-merge operation, is shown in Figure 1.1. In this particular figure time flows from left to right and each horizontal

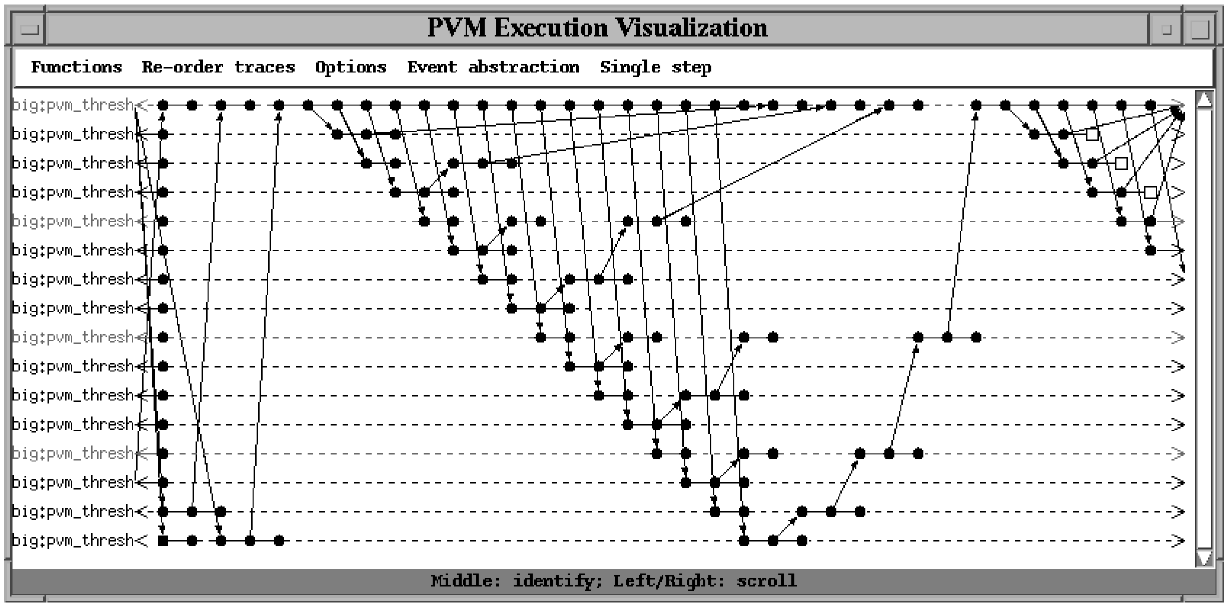


Figure 1.1: Sample POET Process-Time Diagram

line represents a process in a PVM execution. Circles represent events within processes and slanted arrows represent asynchronous message transmissions between processes.

There is a significant limitation in POET and in any similar system. Such systems are limited to reasoning about and displaying only a small portion of the execution of at most a few-hundred communicating sequential processes. It is this limitation that we propose to examine in this report. For our purposes we require a tool that can work with a far greater numbers of processes (and, by implication, a very large number of events). If the reader has difficulty imagining such a need, consider visualizing the execution of a parallel database engine over a few-hundred-processor machine. Each processor will execute a few-hundred database processes, communication processes, database-access agents, and so forth. It rapidly enters the realms of the tens or hundreds of thousands of concurrently executing processes.

In the remainder of this report, we first present a framework by which we may discuss how distributed systems are observed and controlled. We will briefly describe the function of the various components within this framework. We then define a detailed formal model of distributed systems and of distributed computations that execute over those systems. The framework, combined with the formal model, allows us to discuss existing techniques for observing and controlling computations. The significant aspects we will focus on are manipulating and querying the execution partial order, displaying it, and manipulating that display. Given this state-of-the-art, we will formalize the problems that exist when scaling such techniques to large numbers of processes. Having identified these problems we will suggest possible approaches for solving them.

2 MANAGEMENT FRAMEWORK

Our framework for observing and controlling distributed systems is presented in Figure 2.1. The various components of it are defined as follows.

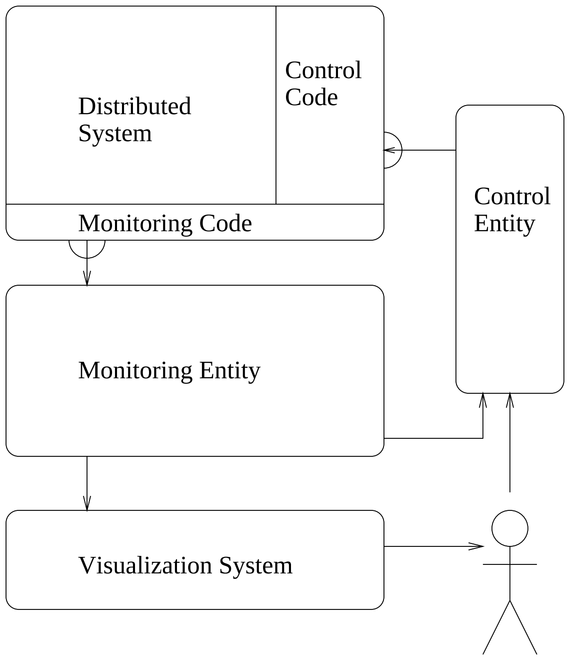


Figure 2.1: Observation and Control of Distributed Systems

The distributed system is the system under observation. It is described in detail in Section 2.1. It should be recognized that it will not be a single entity. This contrasts with the other components in the diagram (with the exception of the monitoring and control code, described below) which are single entities within the framework. Thus, the distributed computation is not expected to have a view of itself, while the monitoring entity will collect that global view.

The distributed-system code must be instrumented in two distinct ways. First, monitoring code must be present to capture relevant event information. This event information will be forwarded to some monitoring entity. Second, if the computation is to be controlled in any way, the necessary control code must be linked to the system. For example, if the application is debugging, this control code might include the necessary instrumentation to allow the attachment of a sequential debugger.

The function of the monitoring entity is to collect all of the event information that is captured by the monitoring code. This data it collects must be presentable to the visualization and control systems. In summary, it provides a data structure for storing and querying the mathematical representation of the execution of the distributed computation.

The visualization system presents the monitored information to a human. While the specific visualization presented will depend on the specific tool requirements, it ultimately is a representation of the distributed-computation execution. The visualization may be manipulable in various ways.

The control entity takes input either directly from the monitoring entity (*i.e.*, from the data structure representing the computation) or from a human (or some combination of the two). Its function is to control the computation (*via* the control code that it has access to) based on the observed behaviour of the computation.

By way of example, if we implement a distributed breakpoint tool, a human will set a breakpoint condition and then commence execution of the computation. The monitoring code will relay the various events in the computation to the monitoring entity. It builds the data structure representing the execution. The control code will analyze the data structure to determine when the breakpoint occurs. It will then cause the computation to be halted. We will discuss specific techniques for distributed breakpoints in Section 3.1.1.5

Having presented this framework, we will now describe the formal model of distributed systems to which we will apply this framework.

2.1 DISTRIBUTED-SYSTEM MODEL

Collecting together various researchers' views of what constitutes a distributed system, we arrive at the following general consensus. A distributed system is a system composed of loosely coupled machines that do not share system resources but rather are connected *via* some form of communication network. The communication channels are, relative to the processing capacity, low bandwidth and high latency [51]. Note that while the bandwidth is improving, the latency will remain relatively high because of the laws of physics. Both the machines and the network may be faulty in various ways [8]. The failure of any machine or portion of the network in any particular way does not imply the failure of the whole system. Indeed, partial failure is one of the most difficult aspects to deal with in distributed systems [79].

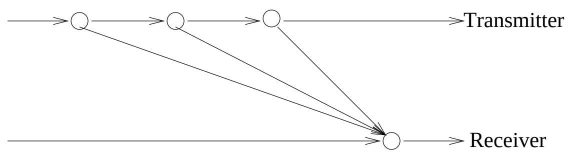
A distributed computation is any computation that executes over such a distributed system. Again, the consensus view is that the distributed computation is composed of multiple processes communicating via message passing. The processes are physically distributed [45], their number may vary over time [59] and they are sequential (that is to say, the actions that occur within any process are totally ordered). The communication may be synchronous or asynchronous, point-to-point (also referred to as unicast), multicast or broadcast. The logical network is a fully connected graph (*i.e.*, any process may communicate directly with any other process) and there is no other knowledge of the underlying physical network [58]. The underlying physical network may not be (probably is not) fully connected. This broad description is used as it encompasses the various more-specific descriptions that some researchers use and there is no generally agreed-upon restriction of this definition.

2.1.1 FORMAL MODEL

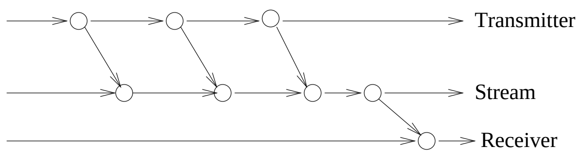
We now take this description of a distributed computation and translate it into a formal mathematical model. There are (at least) three possible techniques that might be employed here: Petri nets, a state-based approach, and an event-based approach. While Petri nets have been used to model distributed programs [53] they are not well-suited to our application. State-based approaches, on the other hand, are not so easily dismissed. They are the preferred choice for sequential debuggers, such as `dbx` or `gdb`, and are frequently employed for the formal specification and verification of distributed algorithms [45]. The method used is to represent the computation as a triple $< S, A, \Sigma >$ where S is a set of states, A a set of actions, and Σ a set of behaviours of the form

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_1} s_2 \ldots$$

where $s_i \in S$ and $\alpha_i \in A$. Some form of logic, typically temporal logic, is then used to reason about Σ. The problem with this approach is that, while it may be suitable for reasoning about distributed algorithms, it is not well adapted to debugging or observing a distributed computation. There are several aspects that make it poorly suited to our application. Perhaps the most significant limitation is that, while communication and concurrency are probably the most significant aspects of interest in a distributed debugging and observing context, they must be inferred in the state-based approach rather than being central. Furthermore, because of non-determinacy and partial failure, there is no well-defined global state in a distributed system at any given instant. In the case of sequential debugging there is a single, well-defined state at any given time and actions deterministically move the system from one such well-defined state to another. It is this state that the user of the debugger or observation system explores. In a distributed computation there is not a single deterministic movement by actions from one state to another. Instead, at any given instant the system could be in any one of a large number of states.



(a) Multi-Receive Operation Direct



(b) Multi-Receive Operation with Stream

Figure 2.2: Modeling Multi-Receive Operations

The third technique, and the one we, and most others in this field, adopt, is the event-based approach, originally developed by Lamport [46]. Rather than focusing on the state, it focuses on the *events* (or actions, to use the terminology of the state-based approach) which cause the state transitions. Information about events can be collected efficiently without regard to the current “state” of the system. Causal links between events, and thus between local states, can then be established.

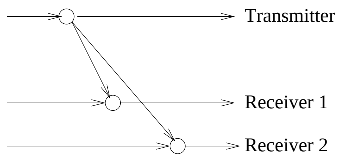
The method taken is to abstract the sequential processes as sequences of four types of events: *transmit*, *receive*, *unary*, and *synchronous*. These events are considered to be atomic. Further, they form the primitive events of the computation.

Transmit and receive events directly correspond to transmit and receive operations in the underlying distributed computation. Every transmit event has zero or more corresponding receive events in different processes. This models attempted transmission, where the message was not received, unicast (or point-to-point) transmission, and multicast transmission. An unsuccessful transmission is effectively equivalent to a unary event in terms of the mathematics.

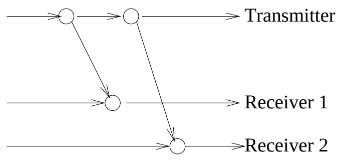
Every receive event has one or more corresponding transmit events. If there is only a single corresponding transmit event then the underlying operation is a simple unicast or multicast transmission. If there are multiple corresponding transmit events then the underlying operation is akin to a transmitter writing several blocks of data to a stream which the receiver reads in one action. Note that this can also be modeled by abstracting the stream as a process. The transmit events would be to the stream process, rather than to the receiver. The receiver would have a single receive event corresponding to a transmit event from the stream process to the receiver. This is illustrated in Figure 2.2.

We note also that a similar transformation may be performed with multicast transmissions. Figure 2.3(a) shows the initial multicast. For such a multicast there is an obvious transformation, shown in Figure 2.3(b), which is wrong. The reason it is wrong is that the transmit operation is split into two events, denying its atomicity. Further, the first of the transmit events precedes both of the receive events, while the second transmit only precedes the receive of the second receiver. It is not at all clear that this is a valid transformation. An alternate transformation, shown in Figure 2.3(c), maintains a single transmit event in transmitter process, and introduces an intermediary as did the multi-receive. This transformation maintains the events and precedences of the transmitter and receiver processes.

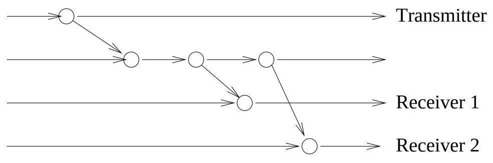
While these transformations appear to be legitimate, at this early stage we prefer to keep our model as general as possible. In particular, we are concerned that it is not clear what effect such transformations will have on the mathematics of the partial order induced by the computation. In addition, we do not have a general algorithm for such transformations. We hope to develop (or discover) formal transformations, that are valid in terms of partial-order theory, for removing multicast and multi-receive operations.



(a) Multicast Direct



(b) Point-to-point Multicast



(c) Multicast with Intermediary

Figure 2.3: Modeling Multicast Operations

Continuing with the remaining two types of primitive events, a unary event is any event of interest that does not involve information transfer between processes. Its primary purpose is to allow any additional action or state information to be recorded that the user of the formal model desires. Synchronous events correspond to synchronous operations in the underlying computation. We will omit them from our abstraction at present and discuss the difficulties involved in modeling them in Section 2.1.1.3.

We now briefly review some basic partial-order terminology before presenting the formal mathematical abstraction.

2.1.1.1 Partial-Order Theory

The following partial-order theory is due to Ore and Trotter [54, 74, 75, 76]. While we will not give detailed partial-order theory here, we will review a few of the important definitions and terminology relevant to this document.

A *partially-ordered set* (or poset, or partial order) is a pair (X, P) where X is a finite set and P is a reflexive, antisymmetric, and transitive binary relation on X. First note the restriction to finite sets. This is not strictly a requirement of all partial orders, but it is one that we impose as we are modeling distributed computations. As such all sets we deal with are finite. This restriction simplifies various theorems and proofs. Second, since the set X is often implicit, it is frequently omitted, and the partial-order is simply referred to by the relation P. This invariably causes problems. We will therefore use the convention of subscripting the relation with the set over which it forms a partial order. Indeed, we will extend this convention to all relations over the set X. Third, note that P_X is reflexive. If P_X is instead irreflexive, then (X, P) forms a *strict partial-order*. The “happened before” relation, as defined by Lamport [46], is irreflexive and corresponds to the relation $\prec_p$ that we will define in Section 2.1.1.2.

A *subposet*, $(Y, P|_Y)$, is a poset whose set Y is a subset of X, and whose relation $P|_Y$ is the restriction of P_X to that subset.

An *antichain* is any completely unordered poset. The *width* of a poset is the longest antichain contained in that poset. In the context of a distributed computation, the width must be less than or equal to the number of processes.

An *extension*, (X, Q), of a partial order (X, P) is any partial order that satisfies:

$$(x, y) \in P_X \Rightarrow (x, y) \in Q_X$$

If Q_X is a total order, then the extension is called a *linear extension* or *linearization* of the partial order. If (Y, R) is an extension of the subposet $(Y, P|_Y)$ of (X, P), then (Y, R) is said to be a *subextension* of (X, P). A *realizer* of a partial order is any set of linear extensions whose intersection forms the partial order. The *dimension* of a partial order is the cardinality of the smallest possible realizer.

A critical pair, also known as a non-forced pair, is defined as follows:

Definition 1 (Critical pair) *(x, y) is a critical pair of the partial order (X, P) if and only if $(x, y) \notin P_X$, $(y, x) \notin P_X$, and $(X, P \cup \{(x, y)\})$ is a partial order.*

Equivalently:

$$(x, y) \in \mathrm{CP}_X \iff \begin{array}{c} (x, y) \notin P_X \wedge (y, x) \notin P_X \wedge \\ \forall_{z \in X} \; (\; ((z, x) \in P_X \Rightarrow (z, y) \in P_X) \; \wedge \\ ((y, z) \in P_X \Rightarrow (x, z) \in P_X)) \end{array} \tag{2.1}$$

where CP_X is the set of all critical pairs of the partial order (X, P). The significance of critical pairs, as regards dimension, is in the following theorem [75]:

Theorem 1 *The dimension of a partial order is the cardinality of the smallest possible set of subextensions that reverses all of the critical pairs of the partial order.*

A critical pair (x, y) is said to be reversed by a set of subextensions if one of the subextensions in the set contains (y, x). Note that the subextensions need not be linear.

Finally, we say that "x is covered by y" or "y covers x" if there is no intermediate element in the partial order between x and y:

$$x <: y \iff (x, y) \in P_X \wedge \nexists_z \left((x, z) \in P_X \wedge (z, y) \in P_X\right) \tag{2.2}$$

In the context of distributed debugging, if $x <: y$ we may also say that x is an *immediate predecessor* of y or y is an *immediate successor* of x. If multicast and broadcast operations do not exist (*i.e.*, all communication is point-to-point), then any event will have at most two immediate successors or predecessors.

2.1.1.2 Mathematical Abstraction

We now present the formal mathematical abstraction of a distributed computation. First we define the finite sets[1] $\mathcal{T}$, $\mathcal{R}$, and $\mathcal{U}$ as being the sets of *transmit*, *receive*, and *unary* events respectively. These sets are pairwise disjoint. Collectively, they form the set of events for the whole computation:

$$\mathcal{E} = \mathcal{T} \cup \mathcal{R} \cup \mathcal{U} \tag{2.3}$$

For convenience of terminology we will use the lower case letters t, r and u (possibly sub- or superscripted) to refer to specific transmit, receive, and unary events respectively. We will use e and f (possibly sub- or superscripted) to refer to specific events of unknown type.

Next, we define $\mathcal{P}$ as the set of processes that form the distributed computation. Each sequential process is the set of primitive events that compose it together with the relation that totally orders those events within that process. Thus

$$\forall_{p \in \mathcal{P}}\, p =< E_p, \prec_p > \tag{2.4}$$

where $E_p \subseteq \mathcal{E}$ is the set of events of the process and $\prec_p$ totally orders those events. Again, to simplify terminology, we will use E_p to refer to the event set of process p and $\prec_p$ to refer to its ordering relation. When we wish to refer to specific events in process p we will subscript the event identifier with the process identifier. Thus for an event of unknown type in process p we will refer to e_p. Likewise a transmit, receive and unary event in process p would be t_p, r_p and u_p respectively.

The relation $\prec_p$ totally orders the events of the process p according to the order of execution in the process. That is, if event e_p^i occurs in the process p before event e_p^j then $e_p^i \prec_p e_p^j$. As such, it is convenient to simply number the events in the process, starting at 1, and then use natural-number comparison as the ordering relation. We will superscript event identifiers with their associated natural number. We will refer to this natural number as the position of the event within the process or, more simply, as its position. Thus, we may define the ordering relation as

$$e_p^i \prec_p e_p^j \iff i < j \tag{2.5}$$

It should also be noted that an event is uniquely specified by its process combined with its position within that process. We will therefore consider <process, position> to be the event identifier.

Two further points must be dealt with regarding the relationship between processes and events. First, every primitive event occurs in exactly one process:

$$\forall_{p,q \in \mathcal{P}}\, p \neq q \Rightarrow E_p \cap E_q = 0 \tag{2.6}$$

The flip side of this is that there must be a mapping from every event to a process. We therefore define two mapping functions, ϕ and φ that map events to processes and to positions within processes respectively.

[1] As we are modeling distributed computations all of the sets we deal with will be finite. It is not a problem if the computation does not terminate as our model covers only that portion of the distributed computation that has executed thus far, which will always be finite.

Definition 2 ($\phi : \mathcal{E} \longrightarrow \mathcal{P}$)

$$\phi(e) = p \Longleftrightarrow e \in E_p$$

Definition 3 ($\varphi : \mathcal{E} \longrightarrow \mathcal{P} \times \mathbb{N}$)

$$\varphi(e) = (p, n) \Longleftrightarrow e \in E_p \textit{ at position } n$$

Second, it is sometimes useful to define the reflexive equivalent of the $\prec_p$ relation. It is defined as

$$e_p^i \preceq_p e_p^j \Longleftrightarrow e_p^i \prec_p e_p^j \vee e_p^i = e_p^j \tag{2.7}$$

Thus far we have only related events to individual processes. We now incorporate the effect of communication between processes. To do this, we define the Γ relation.

Definition 4 (Gamma Relation: $\Gamma \subseteq \mathcal{T} \times \mathcal{R}$) $(t, r) \in \Gamma$ *if and only if* r *is a receive event corresponding to the transmit event* t*. Note that* $\forall_r\, r \in \mathcal{R} \Rightarrow \exists_t\, (t, r) \in \Gamma$

Every receive event will have at least one pair in the gamma relation, as it must have at least one corresponding transmit event. The corresponding condition does not hold for transmit events, as there is no guarantee that a transmit event will ever have a corresponding receive event. Further, even if the transmit event does have a receive event, we want to be able to model the incomplete state of the computation where the transmit has occurred but the receive has not. We also define the functions ρ and τ to determine the set of receive events corresponding to a transmit event and the set of transmit events corresponding to a receive event respectively:

Definition 5 ($\rho : \mathcal{T} \longrightarrow 2^{\mathcal{R}}$)

$$\rho(t) = \{r \in \mathcal{R} \mid (t, r) \in \Gamma\}$$

Definition 6 ($\tau : \mathcal{R} \longrightarrow 2^{\mathcal{T}}$)

$$\tau(r) = \{t \in \mathcal{T} \mid (t, r) \in \Gamma\}$$

As per our previous observation, $|\rho(t)| \geq 0$ while $|\tau(r)| \geq 1$. For convenience in future proofs we will extend the domains of these two functions to all events, recognizing that an event that is not a transmit has no corresponding receive events and an event that is not a receive has no corresponding transmit events. Thus, $\rho(e) = \emptyset$ when $e \notin \mathcal{T}$ and $\tau(e) = \emptyset$ when $e \notin \mathcal{R}$.

We now define two partial-order relations across the set of events for the whole computation. These are $\prec_{\mathcal{E}}$ and its reflexive equivalent $\preceq_{\mathcal{E}}$.

Definition 7 $\prec_{\mathcal{E}}$ *is the transitive closure of the union of* Γ *and* $\prec_p$ *for all processes. Thus*

$$\prec_{\mathcal{E}} = \textit{Transitive Closure}\,(\Gamma \cup (\cup_{p \in \mathcal{P}} \prec_p))$$

$\preceq_{\mathcal{E}}$ is defined analogously:

Definition 8 $\preceq_{\mathcal{E}}$ *is the transitive closure of the union of* Γ *and* $\preceq_p$ *for all processes. Thus*

$$\preceq_{\mathcal{E}} = \textit{Transitive Closure}\,(\Gamma \cup (\cup_{p \in \mathcal{P}} \preceq_p))$$

There is nothing in the formal model we have defined that would require these relations to be anti-symmetric. If they are not anti-symmetric they cannot be partial-order relations. Consider for example $(t_q^2, r_p^1) \in \Gamma$ and $(t_p^2, r_q^1) \in \Gamma$. $r_p^1 \prec_{\mathcal{E}} t_p^2$. However, $t_p^2 \prec_{\mathcal{E}} r_q^1$, $r_q^1 \prec_{\mathcal{E}} t_q^2$ and $t_q^2 \prec_{\mathcal{E}} r_p^1$. Thus, by transitive closure $t_p^2 \prec_{\mathcal{E}} r_p^1$. Note that what has happened here is that messages must have traveled backwards in time. In fact, any such anti-symmetry would require at least one message to travel backwards in time. While this might be a desirable feature in a distributed system, it does not currently exist. Thus, the relations are anti-symmetric.

The $\prec_{\mathcal{E}}$ relation is the "happened before" relation defined by Lamport [46]. It, together with the base set $\mathcal{E}$, is a *strict partial order* as it is not reflexive. Other terms that are used for either this relation, or the $\preceq_{\mathcal{E}}$ relation, include "precedes" and "causality." The relations themselves more realistically represent potential causality than actual causality. Precedes and happened before both imply a temporal relation which, while they hold for events within the relation, may also hold for many events not within the relation. That said, we will frequently use any of the above terms in writing where it does not cause confusion. When we wish to be precise, we will use the specific relations $\prec_{\mathcal{E}}$ and $\preceq_{\mathcal{E}}$.

The final aspect of the mathematical abstraction which has not been defined is concurrency. We use the reflexive form of the causality relation to define concurrency as follows.

Definition 9 (Concurrent: $\|_{\mathcal{E}} \subseteq \mathcal{E} \times \mathcal{E}$) *Two events are concurrent if they are not in the causality relation:*

$$e_i^j \|_{\mathcal{E}} e_k^l \iff e_i^j \not\preceq_{\mathcal{E}} e_k^l \wedge e_k^l \not\preceq_{\mathcal{E}} e_i^j$$

Note that we need the reflexive form to avoid the problem of defining an event as concurrent with itself.

2.1.1.3 Synchronous Events

We must now incorporate synchronous events in our abstraction. They represent a problem. The problem is the following. A synchronous event is an event that takes place in multiple processes and thus is, in some sense, a collection of events. However, it is, in some logical sense, a single event. That is, it is a primitive, atomic event in the same sense as transmit, receive and unary events are primitive, atomic events. This presents problems with respect to causality. The multiple events that constitute the synchronous event are not concurrent, and yet are not causal, since that would present a causality cycle.

To make this more concrete we will demonstrate the problems by means of an example. A good canonical example of a synchronous event is a remote procedure call (RPC). In the general case an RPC involves two-way information flow: the parameters to the call and the response from the remote procedure. Therefore causality flows in both directions. Any events in the remote process that handles the RPC that precede the reception of the RPC must happen before any events in the local process that occur after the RPC. Likewise, any events in the local process that precede the RPC must happen before any events in the remote process that occur after the RPC.

There are at least five ways in which we can model synchronous events. Kunz presents four of them [39]. The first technique is not to model them at all. It is based on the idea that a synchronous event is likely composed of some asynchronous message transmission and a blocking wait for the asynchronous response. Thus the event is modeled as the four events that constitute the transmitting and receiving of the two asynchronous messages. This is the approach taken by van Dijk and van der Wal [77], though with the modification that the remote receive event is also a transmit event. There are several problems with this approach. The first issue is that it is not clear how to scale it to an N-process synchronous event such as a barrier. There are many possible asynchronous implementations of such an event, and the implications of picking one of them arbitrarily are unclear. Furthermore, even in the two-process case, it is possible that the underlying communication is synchronous. While this is not common it does occur, as in the case of Occam programs running on Inmos transputers [1]. In such a case the asynchronous simulation is an artifact, again with unclear implications. Finally, it is desirable, if possible, to model events at the user-environment level, not at a lower level. This is the reason we prefer source-level debugging over machine-language-level. Although the machine language is what is being executed, it is the source language that is being debugged. For these reasons we dismiss this approach in all instances where we are trying to model the synchronous event as a synchronous event rather than as its constituent events.

Kunz's three remaining approaches all require a defined set of synchronous events. We therefore define the set $\mathcal{S}$ as the set of all synchronous events within the distributed computation. The precise contents and semantics of $\mathcal{S}$ will depend on the approach we take. However, in all instances we do extend the definition of the set of all events to include this set.

$$\mathcal{E} = \mathcal{T} \cup \mathcal{R} \cup \mathcal{U} \cup \mathcal{S} \tag{2.8}$$

For the second and third approaches we will define the elements of $\mathcal{S}$ to be the constituent events of the synchronous events. Thus, in the RPC example there will be two elements in $\mathcal{S}$: one corresponding to the event in the calling process and the other corresponding to the event in the remote process that executes the RPC.

We must then define a relationship that matches the constituent events into a single synchronous event. Kunz [39] defines the relationship $\leftrightarrow_{\mathcal{E}} \subseteq \mathcal{S} \times \mathcal{S}$ as

Definition 10 (Synchronous Pairs: $\leftrightarrow_{\mathcal{E}}$) $e_i^j \leftrightarrow_{\mathcal{E}} e_k^l$ *if and only if* e_i^j *and* e_k^l *are a synchronous pair.*

What is not stated by Kunz is how this is extended to synchronous events that are composed of more than two constituent events. The most reasonable extension is that $\leftrightarrow_{\mathcal{E}}$ is the symmetric and transitive closure of Definition 10. This then forces all the constituents into a single event.

Both the second and third approaches take the view that all events that are not part of the synchronous event must be in the same causal relation with the events that compose the synchronous event. They differ in how the constituent events are causally related. In effect then they differ in how they treat the extension to the precedence relations in the presence of synchronous events. The second approach is to make these constituent events mutually precede each other [64]. Specifically, the $\preceq_{\mathcal{E}}$ relation is extended as

$$e_i^j \leftrightarrow_{\mathcal{E}} e_k^l \Longrightarrow \forall_e \left(e \preceq_{\mathcal{E}} e_i^j \Rightarrow e \preceq_{\mathcal{E}} e_k^l \wedge e_i^j \preceq_{\mathcal{E}} e \Rightarrow e_k^l \preceq_{\mathcal{E}} e \right) \tag{2.9}$$

This, however, creates the rather strange problem that $e_i^j \preceq_{\mathcal{E}} e_k^l$ and $e_k^l \preceq_{\mathcal{E}} e_i^j$, due to the symmetry of $\leftrightarrow_{\mathcal{E}}$ and the reflexiveness of $\preceq_{\mathcal{E}}$. This would be fine but for the fact that $e_i^j \neq e_k^l$. Thus, although this ensures correct causality, it breaks the partial-order model. This is not a triviality. It prevents the application of partial-order theory to the problem, which is significant.

The third approach treats the constituent events as unrelated. This is the approach taken by Fidge [18]). This is achieved by extending $\prec_{\mathcal{E}}$ instead of $\preceq_{\mathcal{E}}$.

$$e_i^j \leftrightarrow_{\mathcal{E}} e_k^l \Longrightarrow \forall_e \left(e \prec_{\mathcal{E}} e_i^j \Rightarrow e \prec_{\mathcal{E}} e_k^l \wedge e_i^j \prec_{\mathcal{E}} e \Rightarrow e_k^l \prec_{\mathcal{E}} e \right) \tag{2.10}$$

We extend $\preceq_{\mathcal{E}}$ from this in the natural way by asserting that

$$e_i^j \preceq_{\mathcal{E}} e_k^l \Longleftrightarrow e_i^j \prec_{\mathcal{E}} e_k^l \vee e_i^j = e_k^l \tag{2.11}$$

The problem with this approach is that the constituent events are now causally concurrent. This does not seem desirable as they are not concurrent in the usual meaning of the term. Rather, they are in very tight communication. One approach, not explicitly stated by any of the authors we are aware of, is to redefine concurrency (Definition 9) such that it explicitly excludes synchronous events:

Definition 11 (Concurrent: $\parallel \subseteq \mathcal{E} \times \mathcal{E}$) *Two events are concurrent if they are not in the causality relation and if they are not synchronous pairs*

$$e_i^j \parallel e_k^l \Longleftrightarrow e_i^j \not\preceq_{\mathcal{E}} e_k^l \wedge e_k^l \not\preceq_{\mathcal{E}} e_i^j \wedge e_i^j \not\leftrightarrow_{\mathcal{E}} e_k^l$$

The implications of this modification are unclear.

The fourth approach abandons the idea of making the constituent events elements of $\mathcal{S}$, but rather treats the synchronous event as a single event [2]. Thus, in the RPC example there will be only one element in $\mathcal{S}$. This is appealing from the partial-order-modeling perspective, but can lead to difficulties when it is necessary to pin down in which process an event occurred. Our approach to overcome this problem is to allow an event to occur in multiple processes. This requires that Equations 2.6 no longer hold, and that the mapping of events to processes (Definitions 2 and 3) be altered as follows:

Definition 12 ($\phi : \mathcal{E} \longrightarrow 2^{\mathcal{P}}$**)**

$$\phi(e) = \{p \mid e \in E_p\}$$

Definition 13 ($\varphi : \mathcal{E} \longrightarrow 2^{\mathcal{P}\times\mathbb{N}}$**)**

$$\varphi(e) = \{(p,n) \mid e \in E_p \textit{ at position } n\}$$

Kunz [39] presents an additional objection to this fourth approach. When performing event abstraction it may be desirable to abstract the constituent events of the synchronous event into different abstract events. We are unconvinced by this objection. An abstraction that results in splitting a synchronous event into its constituents does not appear to be a good abstraction to us.

The final approach is to treat the synchronous event as an abstract event. We will discuss abstract events in detail in Section 3.2.4.1. Briefly, the idea is the complement of the notion of treating abstract events as synchronous events of the fourth type, which we will discuss in that section.

For the purposes of this report, we use the fourth technique.

2.1.1.4 Other Issues

There is nothing in our formal description regarding partial failure, performance issues or varying numbers of processes, although all three of these were observed to be features of a distributed system. The performance issue will be the subject of specific tools, rather than of the model itself. The monitoring code may be designed to determine performance-related problems. The control portion can then be designed to steer the computation to improve performance. For example, this is the approach used in the Falcon system [17, 24, 35].

Correct handling of partial failure must be designed into the distributed system. This includes the correct handling of failure within the monitoring and control system, and specifically the monitoring and control code that is integrated with the distributed computation. For example, a Byzantine failure in the computation could easily spread to the monitoring entity if the information that is claimed to be observed by the monitoring code is not correct. We will not attempt to deal with this issue in this work. We will presume that the information presented by the monitoring code is correct and represents, in some measure, the behaviour of the computation. We will presume it is the responsibility of a user to determine if the observation is in error, and to act to correct it accordingly. This is an area for future investigation.

The problem of a variable number of processes is evaded at present by modeling any process that will ever exist during the course of a distributed computation as being implicitly present, though without events, from the beginning. A process that leaves the computation simply ceases to have events. While there is nothing in this approach that contradicts the formal model, and it is a very simple method of dealing with a variable number of processes, it is not particularly satisfying. The computation may be unbounded (as in the case of a distributed operating system) or may have very many short-lived processes. As we shall see in Section 4.1, the effect of this approach can be very expensive. Unfortunately we do not yet have, and are not aware of, any alternative.

2.1.2 Other Systems Covered by the Model

Finally, we wish to observe that in the course of our abstraction we have, in fact, managed to model a broader range of systems than simple distributed ones. We therefore take this brief digression to discuss how broad our model is and what systems it can, in principle, cover.

We observe that it can trivially cover message-passing parallel computing, as this is largely indistinguishable from distributed computing. The primary difference is in application and in the speed of message passing. These issues are not of relevance to the formal model we have defined. In addition, it can be applied to concurrent programs, such as thread-based applications or multitasking operating systems. Indeed, any system or environment that can be modeled as a collection of co-operating sequential entities is covered by our model.

We deal with the partial order as a partial order. As such, although we have used the term “process” or “sequential process” to this point, we will generalize the term to “trace,” where a trace is simply any sequential entity. It may be a semaphore, a monitor, a shared variable, a thread, or possibly even a sequential process, as per our original motivation. The key property of a trace is that it is a totally ordered set of events.

3 Current Management Techniques

Current systems for observing and controlling parallel and distributed computations revolve around two main points. First, they must store, manipulate and query the partial order of execution. Second, they must display the partial order and manipulate that display.

The first of these points can be restated as a requirement for a partial-order data structure. The primary danger in this view is that while we are modeling the computation as a partial order, it is in some subtle ways not quite as general as a partial order. Partial orders generated by distributed computations have some properties that are not present in arbitrary partial orders. Such properties include the total ordering of events within a trace, the ability to distinguish individual events by means other than the partial-order structure, and the ease with which processing of events can be restricted to be in a linearization of the partial order. We can use these properties to improve the performance of what we have (somewhat falsely) termed the "partial-order data structure." We will discuss the various techniques for encoding the partial order in Section 3.1.

The second aspect, that of visualizing the partial order, is of somewhat less applicability than the first. Any distributed debugging, monitoring, or steering system that uses the partial-order model must store, manipulate and query the partial order of execution. For debugging, visualization is essential. For monitoring and steering, it may not be required at all, and if it is the required display may be substantially different from the debugging case. The debugging case must show the execution to the developer. By contrast, an automated monitoring or steering system may simply be searching for certain patterns or events, that will in turn trigger certain actions. In the case of monitoring, the action will likely occur at the monitoring station. In the case of steering, some command will be sent to the distributed computation to adjust its behaviour. We will discuss various visualization issues in Section 3.2.

Before discussing partial-order data structures, it should be noted that there are several items that may have some impact on both data-structure choice (and implicitly program structure) and visualization. These are primarily whether the management system is interactive or automatic, observing or manipulating, and online or offline [86]. Also of some relevance is whether the system is target-system-dependent or not. As already noted, interactive tools such as debuggers require visualization while automatic ones typically do not, or if they do, the visualization required can be quite different. Observing, or monitoring, systems present data, after greater or lesser analysis, but do not alter the system (beyond the effect of the monitoring). Debugging and steering, by contrast, explicitly manipulate the computation. Online systems can operate during the execution of the distributed computation. Offline systems usually work by having raw execution data dumped to a file, which is then analyzed and possibly displayed in some manner. Target-system dependence may allow the debugging tool to perform optimizations that are not available in the general case.

There are two key points to notice about these issues. First, it should be noted that they are clearly not orthogonal issues. For example, an offline system cannot be a manipulating system. We hope to develop some better set of criteria for describing the various tools. Second, not relevant, and hence not discussed, is where the instrumentation comes from. It might be automatically generated or manually added to the computation. For the purposes of the following discussion, the source of the instrumentation is not important. It is sufficient that the system is instrumented in some manner sufficient to provide the appropriate data.

We will now discuss the two major points described above, keeping in mind the context of the work and how it may affect algorithm and data-structure choices.

3.1 Partial-Order Data Structures

In this section we will describe various techniques for storing, manipulating and querying a partial order. This cannot be done in the abstract, however. We must first identify what queries are performed on the partial order to explain what is needed in such a structure. We may then describe current techniques for implementing such a data structure.

3.1.1 Data-Structure Requirements

To determine the requirements on the partial order, we must know what operations are performed in a debugging, monitoring, or steering context. We have identified the following items as being general operations that are performed in these contexts:

1. Event inspection
2. Seeking patterns
3. Race detection
4. Computation replay
5. Distributed breakpoints
6. Determining execution differences
7. Performance analysis

We have excluded in the list anything pertaining to visualization, as that will be dealt with separately. However, visualization may impose some requirements on the partial-order data structure. We will briefly enumerate these requirements later in this section. We will now discuss the implications of these operations on the requirements of a partial-order data structure.

3.1.1.1 Event Inspection

Event inspection is informing the user of "relevant information" associated with an event. This relevant information can vary widely. It may be type information such as whether the event is a transmit, receive, synchronous, or some other type. These types will be target-environment specific, not the types we have defined in our formal model. Other information may include partner-event identification, the time at which the event occurred, or various text information that the developer may find useful. As long as the total quantity of data is not substantial this may be collected and provided to the user fairly easily. The primary issue is finding the event of interest within the data structure. This can usually be performed by simple indexing techniques, as we can take advantage of the sequential nature of traces and just use the event identifier. It may be marginally more complex in the target-system-independent environment, where the meaning of what an event is must be associated with the target environment in some manner. POET solves this by the use of the target-description file.

It may also be desirable to provide some subset of the sequential state of a process. This can be particularly useful in reasoning about consistent global states. However, if the quantity of state information required is at all substantial, the cost of collecting it may be prohibitive. It may also require some more-complex storage and indexing techniques than if the quantity of information is small.

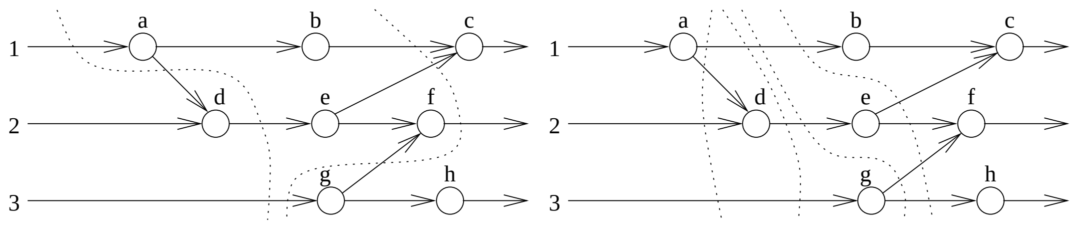
3.1.1.2 Seeking Patterns

Currently there are two key types of patterns that may be sought within a debugging, monitoring, or steering context. First we may seek patterns within the structure of the partial order. For example, we may wish to look for the pattern:

$$e_i \prec_{\mathcal{E}} e_j \wedge e_k \prec_{\mathcal{E}} e_j \wedge e_i \parallel_{\mathcal{E}} e_k \; ; \; i \neq j \neq k \tag{3.1}$$

This particular pattern is a crude form of race detection. We are seeking events in traces i and k that both precede an event in a third trace j but that have no synchronization between them. The events thus form a potential race condition. We will discuss race detection in more detail below.

This form of structural pattern searching is equivalent to directed-subgraph isomorphism. Specifically, it is equivalent to asking if the directed acyclic graph that represents the partial order of the computation contains a subgraph isomorphic to the directed graph that represents the pattern being sought. The directed graphs in this equivalence can be



(a) Inconsistent Cuts (b) Consistent Cuts

Figure 3.1: Global State

either the transitive reductions or the transitive closures of the respective partial orders. This problem is known to be NP-complete, even if the pattern sought is a directed tree [22].

The second type of pattern that we may seek is a pattern within a consistent global state. This is frequently referred to as global predicate detection [11]. We will first explain what is meant by a consistent global state. Consider the execution shown in Figure 3.1. It is not possible for trace 1 to be in the state prior to the event *a* at the same time as trace 2 is in the state between events *d* and *e*, as shown in the first cut in Figure 3.1(a). On the other hand, it is possible for trace 1 to be in the state between events *a* and *b* while trace 2 is in the state between events *d* and *e*, as seen in the second and third cuts in Figure 3.1(b). To explain these points further we must first provide some clear definitions of our terms. The following definition is a modification of the one in [3], adjusted to include synchronous events.

Definition 14 (Consistent Cut: $\mathcal{C} \subseteq \mathcal{E}$) *A subset of the set of events $\mathcal{E}$ forms a consistent cut $\mathcal{C}$ if and only if*

$$\forall_{e_1,e_2\in\mathcal{E}}\ (e_1 \prec_{\mathcal{E}} e_2 \vee e_1 \leftrightarrow_{\mathcal{E}} e_2) \wedge e_2 \in \mathcal{C} \Longrightarrow e_1 \in \mathcal{C} \tag{3.2}$$

Informally, a consistent cut represents a set of events in which no receive event has occurred without the corresponding transmit event having occurred. In the case of synchronous events, either all constituent events of the synchronous event are in the consistent cut, or none are. It is called a cut because it cuts the set of events in two, as shown in Figure 3.1. Those events to the left of the cut line are part of the cut; those to the right are not. It is called a consistent cut because the cut creates a consistent global state. Figure 3.1(b) shows various consistent cuts.

The set of consistent cuts ordered by $\subseteq$ is a complete lattice [3]. The lattice for the computation of Figure 3.1 is shown in Figure 3.2. The computation can then be viewed as proceeding on *some* path from the bottom element of the lattice to the top element. Note that this is not strictly true, as it is possible for more than one event to occur simultaneously. However, any such instance would amount to a race condition. If it is a significant race condition, it is a defect in the program. If not, it will not be relevant which path is chosen. We will address this point further as needed.

Given this view of the computation, and any ascending path as a possible execution sequence, we may then consider the computation to have a possible sequence of global states that is the sequence of states along that path. In other words, a potential global state of the computation is any edge within the lattice. Such a potential global state is referred to as a consistent global state, as it is consistent with a possible execution order of the computation. An alternate way of looking at a consistent global state is to consider it to be a sequence of edges in the process-time diagram which divide the computation to form a consistent cut.

Having acquired a notion of a consistent global state, we can seek patterns within that state. Such patterns are referred to as predicates. There are several varieties that may be sought, such as stable predicates (once the predicate is true, it remains true), definite predicates (the predicate is true on all possible paths in the lattice), possible predicates (the predicate is true on some paths in the lattice), and so forth. From the perspective of a partial-order data structure, the primary concern is the ability to determine what is, or is not, a consistent global state. This is turn means we need the ability to determine consistent cuts.

Given these two pattern-seeking operations, we can identify various requirements for our partial-order data structure.

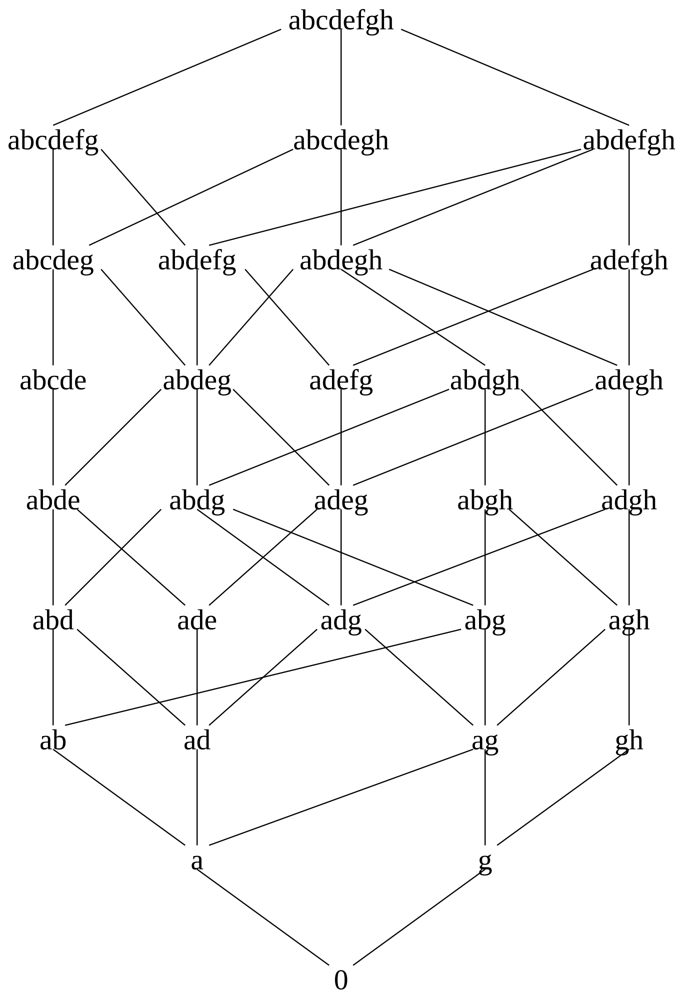


Figure 3.2: Global State Lattice

Clearly, from the first type of pattern, we need to be able to seek out precedence patterns in the structure. Thus, we require efficient determination of the $\prec_{\mathcal{E}}$, $\preceq_{\mathcal{E}}$ and $\|_{\mathcal{E}}$ relations. From the second pattern type, we can also identify a need to determine the sets of greatest predecessors and least successors to an event within each trace. This can be seen in that nothing prior to the greatest predecessor or subsequent to the least successor of an event can form a consistent cut, and hence a consistent global state, with that event. Other requirements of the data structure may emerge as we determine more-definite information about specific pattern-seeking schemes. For the present, these are clearly basic requirements.

3.1.1.3 Race Detection

We have already seen that race detection is, in some sense, a special case of structural pattern seeking. Specifically, Equation 3.1 gave a crude pattern for determining possible races. There are several problems with that particular pattern. First, because it uses precedence, rather than the more specific *covers* relation, it will catch many cases that are simply the transitive closure of the specific race events. Thus, we start by narrowing the pattern to

$$e_i <: e_j \wedge e_l <: e_k \wedge e_j <: e_k \wedge e_i \,\|_{\mathcal{E}}\, e_l \;;\; i \neq j \neq k \neq l \tag{3.3}$$

This is, unfortunately, still insufficient as a pattern, as it has not forced e_i to be in a different trace from that of e_j and e_k, which must be sequentially ordered in the same trace, and themselves, in turn, in a different trace from e_l. Clearly we require a mechanism for specifying that events be in the same or different traces when seeking patterns of this variety. In a more general sense, this requires the use of variables within pattern specifications.

These issues are fairly minor, however. There is a much deeper issue, which is that, even with a perfect specification of a race condition, we have only identified a race condition. We have not concluded that it is significant. In other words, non-determinism exists and is not necessarily incorrect. Indeed, some programming languages are built around the concept of non-determinism (*e.g.*, concurrent logic languages [60] which are built around Dijkstra's guarded command [16]). Finding a race condition in a program written in such a language is not significant in and of itself. It is necessary to determine if the race condition is significant.

Damodaran-Kamal and Francioni [14] refer to programs with race conditions as nonstrict. They are then classified as either determinate or nondeterminate. A nonstrict determinate program is one whose output remains the same under all executions. That is to say, the outcome of race conditions does not affect the program outcome. More precisely, it should not. If it does, the program has a defect. In other words, a race condition is significant in a nonstrict determinate program if the reordering of the outcome of the race causes a different output by the program. Determining if the various races are significant in this instance is, to some degree at least, possible. After recording the execution of the program, the program is replayed, but it is constrained to follow the opposite outcome for each race condition in turn. If the output of the program remains the same, then the race conditions were not significant. This is, of course, both expensive (though automatable in principle) and requires replay capability. We will discuss replay further in the following section.

The second class of nonstrict programs are those that are nondeterminate. These programs produce a different output under different race outcomes. Damodaran-Kamal and Francioni [14] claim that this is undesirable and do not deal with it further. The problem with this view is that there are legitimate programs which create outputs that differ under differing race outcomes. An obvious example is a bank account. Consider \$10 in a joint account. Simultaneously one party to the account deposits \$100 while the other attempts to withdraw \$50. Under one execution the account has \$110 at the end. In another, it has only \$60. Both are valid outcomes. Any distributed operating system will have behaviour that is similar to this. There is no current solution to this problem that we are aware of.

There are, therefore, several issues that must be addressed by our data structure to deal with race detection. First, it must support efficient structural pattern detection as per our previous section. Further, the event inspection must include enough detailed information to support race-detection patterns. Thus, clearly the event identifier must be accessible, but also there must be a tie-in to the code base in the event information. It is not clear if this information can be made target-system independent, automatically generated, provide sufficient information for race-detection purposes and be efficient. The data structure must also support efficient replay, which we will discuss now.

3.1.1.4 COMPUTATION REPLAY

Computation replay is the constrained re-execution of a distributed computation such that it follows a specific partial order. Note that the partial order that the re-execution is constrained to follow need not be the same as the partial-order induced by the original execution. Also note that the constraint may be limited to a point in the re-execution, after which the computation may be no longer so constrained. There are several reasons for doing computation replay.

First, the act of debugging a computation interferes with the execution of that computation. This is known as the probe effect [21]. To minimize this effect, we must minimize the impact of debugging. One way of achieving this is to execute the program while collecting the minimum amount of information necessary to determine the partial order of execution. This causes the least perturbation. In order to now closely examine the program execution, we re-execute it, constraining it to follow the partial order of the original execution. This is the technique used by Yong and Taylor [88].

While computation perturbation can be minimized using methods such as the above monitoring process, it cannot be eliminated. However, because of the event model that we are using, we can capture the effects of these perturbations [49]. Specifically, any perturbation of the computation caused by monitoring is equivalent to the delaying of an event in one trace relative to some event in another trace (possibly caused by some unintended synchronization resulting from the monitoring). This is nothing more than a race condition. The objective in this case is then to execute the program, capturing the partial order of execution. The program is then re-executed, constrained to follow the original partial order up to the relevant race condition, and then the outcome of the race condition may be altered to the opposite of that which occurred in the original execution. This may be performed automatically, by automating the search for race conditions and then invoking the appropriate re-execution. We are not aware of any system that does this, because it is difficult to identify relevant race conditions and the possibly legitimate different outcomes in the presence of such nondeterminism. Alternately, the re-execution may be performed interactively, with the developer selecting which race conditions should be tested for. This is the approach used by Kransmueller *et al.* in their EMU [23], ATEMPT [37] and PARASIT [38] tool suite.

A third reason for computation replay is the need for efficient distributed breakpoints. We will describe that in the following section.

The requirement for computation replay makes several demands of our partial-order data structure. We must, at the very least, be able to determine minimal elements from a consistent cut. These would represent the set of events that could occur next while satisfying the partial order. A more sophisticated approach would allow several traces to be re-executed concurrently, while still following the partial order. Yong [87] achieves part of this with the idea of replay intervals, which is a contiguous sequence of RPC calls within an OSF DCE environment. This amounts to collapsing a sequence of unary events into a single event for replay purposes. We suspect that the more sophisticated approach we envision would require a search for a causally consistent block of events with no race conditions within the block. Such a block of events could be re-executed without need for debugger intervention. This would improve the efficiency of the replay operation. We are not aware of work that has been done in this regard.

An alternate technique is to checkpoint the computation at various intervals, and use the checkpoints to avoid much of the replay. This is only applicable if the intent of the replay is to reach a specific point in the computation and then proceed with some action. The reasons we have cited all fall into this category, though there may be other reasons for which this technique would not be applicable.

3.1.1.5 DISTRIBUTED BREAKPOINTS

A distributed breakpoint is intended to perform the same function for a distributed computation that a normal breakpoint does for a sequential computation. That is, it is intended to allow a user of a distributed debugging tool to stop the computation on some specified trigger (typically reaching a line of code or a change in the value of a variable) and examine the program state. This is clearly a hard problem for a distributed debugger as there is no well-defined global state, but rather a set of possible global states. We will first describe the relevant issues involved in distributed breakpoints and then indicate what support would be needed from our data structure.

First, a trigger event may be as simple as a change in value or reaching a piece of code in a sequential process that is part of the whole computation. However, there is no reason why it should be so specific. The trigger event may be distributed over the computation, in which case it would amount to a pattern that needed to be discovered, per our previous discussion. We will name traces where trigger events occur as trigger-event traces. Likewise, those that do not contain trigger events will be called non-trigger-event traces.

Given that a trigger event has occurred in the computation, we then need to decide exactly what needs to be stopped and where. In a sequential debugger, the entire computation is stopped. In the distributed case it is not quite as obvious. Clearly the trigger-event traces must be halted, and this is a simple matter of applying the relevant sequential halting solution. However, for non-trigger-event traces, the solution is not quite as clear. One solution would be to allow such traces to continue, on the presumption that they will halt as they become causally dependent on the halted traces. At that point they would block, in a causally consistent state, and sequential debuggers could be applied to those non-trigger event traces as needed.

There are at least two problems with this approach. First, there is no guarantee that non-trigger-event traces will ever synchronize, and thus halt. For example, they may be slave processes that perform a computation, asynchronously return the result, and then terminate. Thus, after start up, they may no longer be causally dependent on a trigger-event trace. This problem may be solved by forcing non-trigger-event traces to halt as soon as we have detected the trigger event. By definition the computation will halt in a consistent global state, since it would not be possible for a receive to occur before the corresponding transmit. However, there may be outstanding messages (ones that have been sent but not yet received), and care must be taken not to lose these. This solution is the method employed by Miller and Choi [50].

The second problem is that relevant debugging information may be lost. Specifically, the reason we desire non-trigger event traces to be halted is that the trigger event may be causally dependent on events that occurred in non-trigger event traces. However, by allowing those traces to continue, the relevant causes of the trigger event may be discarded. Thus Fowler and Zwaenepoel [20] proposed the idea of a *causal distributed breakpoint*. In this technique every trace must be halted at the greatest predecessor event of the trigger event. This ensures that further processing in non-trigger-event traces does not obscure the state that led to the causal message transmission. This technique is not guaranteed to prevent the loss of relevant debugging information, since the happened-before relation reflects potential causality, not actual causality. However, for well-written programs potential causality and actual causality should coincide.

The Miller and Choi approach, in so far as it is useful, makes no requirement on our data structure. The Fowler and Zwaenepoel technique, however, does in two ways. First, we must be able to identify the greatest-predecessor set of events. Second, it cannot be efficiently implemented without using replay. To implement it without replay would require that every transmit event be followed by a local temporary halt until it could be determined whether or not that event was a greatest-predecessor event to the breakpoint-trigger event. Note that because of transitivity it is not even clear quite how this might be determined in an online manner. This is, to our knowledge, an open problem. Given this problem, the alternative is to perform a computation replay up to the greatest-predecessor cut.

CHECKPOINTING

Replay may be time consuming. It may therefore be desirable to checkpoint the individual traces at various times during the course of the computation. Replay to a given breakpoint is then accomplished by first starting at a set of checkpoints and proceeding until the breakpoint is reached. This is more complex than replay from the start. The reason is that the set of checkpoints must either form a consistent cut of the partial order in which no messages are outstanding (*i.e.*, the cut, C, must satisfy $\forall_t\, t \in C \Rightarrow \rho(t) \in C$ in addition to equation 3.2.), or the relevant message information must also be logged.

There are two principle approaches to implementing the no-message-logging technique. The first method is to ensure that when checkpoints are made, there are no outstanding messages. A replay operation then requires the identification of the latest such checkpoint that is prior in all traces to the causal distributed breakpoint. While the creation of such a checkpoint is feasible, it would substantially impact the performance of the system since it is effectively a global synchronization operation. The second method is to checkpoint individual traces as seems appropriate. When a replay operation is required, a set of checkpoints is sought out that form a consistent cut with no outstanding messages and is prior to the causal distributed breakpoint. This second method can rely on our data structure to determine the required cut. Note that this method has no guarantee that the required cut will be found.

If message logging is possible then it is not necessary to find such a restricted cut. It may not even be necessary to find a consistent cut. The Fowler and Zwaenepoel technique [20] uses a coordinator to initiate checkpoints in a two-phase protocol. First, all processes are requested to checkpoint. Each process informs the coordinator what the last local event identifier is that it checkpointed. Having received responses from all processes, the coordinator issues a checkpoint confirmation to all processes, indicating the maximum event checkpointed by each process. After a checkpoint is initiated, each receiver logs all messages received until the checkpoint confirmation. After that, receivers log all messages which have a transmit event identifier that is less than the identifier indicated in the checkpoint confirmation. All dependency information is also recorded throughout. This is then sufficient information to restore the computation and commence a replay, even though the restoration may not be to a consistent global state. Using this technique, the requirement of our data structure for checkpointing is simply to be able to identify the checkpoint prior to the causal distributed breakpoint.

3.1.1.6 Execution Difference

Debugging systems are frequently used in a compile, test, debug cycle. A primary question that a user may wish to know the answer to after making a code fix is whether or not the fix made any difference, and if it did, what difference it made. The same cycle and questions occur in debugging distributed computations. However, when presented with two displays of non-trivial-size partial orders, a user cannot easily determine where those orders differ. Han [25] therefore designed a system to determine those differences (more precisely, the point at which the differences start, by trace). This is similar to structural pattern recognition, though with a couple of important differences. First, the "pattern" being sought is of the same size as the partial order it is being sought within. Second, where we will reject patterns that do not match (*i.e.*, the pattern matching case is strict subgraph isomorphism), we explicitly wish to know the differences between the two orders. That is, our primary concern is not an answer to the graph-isomorphism question (*i.e.*, are these two partial orders identical?). This is one reason why graph-isomorphism approaches to this problem are insufficient. A second, and stronger reason, is that execution-difference algorithms can take advantage of factors in the partial order of computation that are not present in general graph isomorphism. For example, the partial order has totally ordered traces within it. Once it is possible to match traces between partial orders, then event differences are simply difference within those traces.

The data-structure support necessary for this is primarily event-identification information and matching-partner information. In addition, trace-identification information might be useful. This trace-identification information may be of limited value, since it is not guaranteed (in an arbitrary target environment) that trace identification or order of appearance will remain the same between executions.

As a side note, it should be noted that Han's algorithm is offline. It might be useful to have an online version of this. Two possible online approaches are possible. First, if we have executed a computation, and recorded its partial order of execution, then it may be desirable to monitor its re-execution and invoke the debugger in the event that differences are detected. Second, if may be useful to compare two simultaneously executing computations.

Finally, it would be useful if more-sophisticated differences could be observed. The Han algorithm is limited to indicating the point at which traces diverge. It may be that there is a portion of difference, after which the computations continue with an identical order again. If this were the case, identifying this difference in whole, rather than the starting point only, would be quite useful.

3.1.1.7 Performance Analysis

Performance analysis, and enhancement, is often a key requirement for observation and control. For example, in Vetter's definition [78], program steering is either for performance enhancement or application exploration. Enhancement mechanisms tend to be application-specific, even when the steering mechanisms are of a more general nature, such as load balancing. The analysis that leads to the need for a specific mechanism, though, can be general. We will therefore omit further discussion of enhancement in favour analysis.

There are various aspects of the partial-order model that make it attractive for performance analysis. Specifically, the minimum execution time can be bounded by determining the longest chain within the partial order (see, for example, the critical-path diagram of ParaGraph [27]). In doing such a check, the partial-order data structure would have to maintain real-time information for each event, and the longest chain would have to be determined with respect to this information. Such a minimum execution time will only be achievable if there are a sufficient number of resources present during the execution. This number is not simply the longest antichain, as traces do not have a one-to-one correspondence with processes. This is apparent in Figures 2.2 and 2.3. As such, it is necessary to determine the longest antichain that corresponds to processes. This is likely target-specific. Also of value is the variation in parallelism over the course of the computation. This is akin to some notion of what the longest antichain is at a given point in time. This may help in the dynamic scheduling of processors between multiple distributed computations so as to maximize throughput for the whole. In a similar vein would be determining the minimum execution time in the presence of a fixed number of processors that is smaller than the maximum degree of parallelism. This is equivalent to determining the maximum chain in the event of creating an extension to the partial order such that the longest antichain is no more than the number of processors. The common theme in all of these ideas is the requirement on the data structure for determining antichains and longest chains, and for extending the partial order so as to reduce the length of the longest antichain.

3.1.1.8 Visualization Requirements

The requirements visualization imposes on a partial-order data structure are heavily dependent on the desired visualization, which is in turn dependent on the specific application need. It is not the intention of this section to describe the various visualization methods. That will be done in Section 3.2. Rather, this is an enumeration of the requirements under various visualizations.

The most elementary visualization simply presents raw events, ordered by trace. Such a visualization needs little data-structure support. It merely requires the events to be ordered within a trace, though the partial-order displayed may be incorrect. That is to say, events in one trace may be displayed in a manner that makes it appear that they occurred after events in another trace, when the ordering relationship is such that they must have occurred earlier. To present a plausible display requires enough support to give a linearization of the partial order. To go further, and allow a user to determine the precedence relationship between events in the display requires precedence-determination support.

Since no non-trivial computation can be displayed in entirety in a single screen, scrolling support is needed within the display. The elementary end of scrolling techniques provides a window over the whole partial order and displays that window. Scrolling shifts the portion of the partial order that is within the display window. More-sophisticated techniques recognize that the partial order cannot be treated as a total order that has some portion displayed at a given moment, but rather observe that events must be allowed to slide independently to reflect the partial-order accurately. This appears to require precedence-determination support.

Other desirable visualization features include the ability to place events in a correct real-time placement. Since clocks are not necessarily synchronized within a distributed computation, support is needed to make adjustments to the real-time so that it is consistent with the partial order of execution. Collectively this requires an ability to identify immediate-predecessor events and to have a real-time associated with each event. The ability to display code corresponding to events likewise requires line-number information support. None of these requirements are onerous, or beyond what is required by the non-visualization operations.

Additional visualization operations include abstracting collections of events into a single abstract event, and likewise abstracting collections of traces into a single (possibly not totally ordered) trace. The semantics of abstract events vary. As such the support requirements vary. The primary requirement is the ability to collapse collections of events of the partial-order data structure into single events. That said, various approaches, such as that of Basten [2] abandon the partial-order representation as they create abstract events. However, they usually maintain a notion of order, and as such are moving from a directed-acyclic-graph representation to the more general directed-graph representation. We will discuss this issue in greater depth in Section 3.2.4.1.

The semantics of trace clustering are better understood, though it is not uniform among various research projects. Some simply remove the set of events and trace lines. Others represent them as a total order, with the associated confusion that is caused by presenting causality to the user that is not present in the computation. The preferred approach provides a single trace that is partially ordered such that the precedence relationships of the computation are not altered in the display. We will describe these approaches, and associated data-structure support required, in Section 3.2.4.2.

Finally, some notion of viewing area, and distance from the viewing area, is desirable to allow the presentation of fish-eye and cylinder views of the order. This amounts to a path-length-determination requirement. Since there may be multiple paths, knowledge of the range of path lengths is potentially of value. In addition, some notion of distance between traces can be of value in presenting a clear display. In some sense there is no formal ordering between different traces. However, this is not an entirely accurate view. Traces between which frequent communication occurs are "closer" to each other than those between which communication does not occur. This "closeness" metric can be extended to a notion of the shortest and average distances between traces, where distance is based on the path length

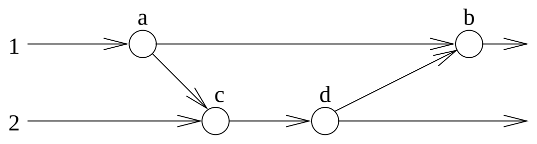


Figure 3.3: Not Quite Transitive Reduction

between events on the two traces. Data-structure support for computing these path lengths is desirable.

3.1.2 A Brief History of Timestamps[1]

We now describe current data-structure techniques for storing, manipulating and querying the partial-order data structure. Since precedence is foundational to most of the other query requirements, we will focus on structures that can satisfy this requirement. We have termed this a brief history of timestamps as this is what the work has largely revolved around. In particular, little work has been applied from directed-graph data structures, though this is perhaps a consequence of the more specialized nature of the distributed-computation partial orders. For example, this is evident in the work of Han [25] on execution differences.

The techniques vary somewhat based on the nature of the specific problem being solved. For example, the more limited visualization techniques are not intended to allow user querying, and as such do not tend to store or manipulate the data structure at all. Rather they simply display it in real-time and allow the user to scroll through that display as though the partial order were a fixed total order. The primary example of this technique is the XPVM [34] software, which simply displays the real-time order of events, without performing any clock synchronization. Going beyond this is the Xab [6, 5] addition to XPVM, which ensures a correct linearization of the partial order of execution.

To actually query and manipulate the partial order requires that it be represented by a data structure. The simplest method for this would be to store the partial order as a directed acyclic graph. In such a case precedence determination is a constant-time operation because the partial order is transitively closed and so there is an edge between any two events that are ordered. However, the space consumption for this method is likely to be unacceptably high. A naive implementation would require $O(n^2)$ space to store the matrix of edges, where n is the number of events in the computation. We are not aware of any work that has approached the problem starting from this point and attempting to improve the space consumption. The primary problem, from our perspective, is that it ignores the optimizations available from the sequentiality of the traces.

All systems we are aware of store, approximately, the transitive reduction of the partial order. The qualifier is required because they will continue to maintain the full sequential-trace information, which is sometimes slightly more than a transitive reduction. This can be seen in Figure 3.3, where the a to b edge is not in the transitive reduction, but will be stored implicitly. This approach is the least space-consumptive possible. Precedence determination then requires determining if there is a directed path between the two events. This is a potentially quite slow search operation. To compensate for this deficiency some additional information is added to allow for a more efficient precedence test. The most common form this additional information takes is a logical timestamp, which receives its name from the fact that it determines the time ordering of events. In the directed-graph view, timestamps amount to the addition of some edges, beyond the transitive reduction of the partial order, to reduce the search time. We now describe five significant timestamp techniques that have found varying application in distributed debugging.

3.1.2.1 Lamport Time

Logical clocks were first introduced by Lamport [46]. The idea is as follows. Each process p possesses a logical clock c_p. The requirement of clock c_p is that it should assign a timestamp c_p^i to each event e_p^i in process p. The requirement of the timestamps produced by the clock are that if an event "happened-before" another event, it should have a lower

[1]With apologies to Stephen Hawking.

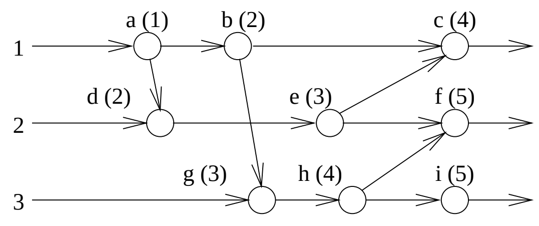


Figure 3.4: Lamport Timestamps

timestamp. This may be expressed formally as:

$$\forall_{e_i^j, e_k^l} \; e_i^j \prec_{\mathcal{E}} e_k^l \Rightarrow c_i^j < c_k^l \tag{3.4}$$

Collectively the processes are maintaining a nearly-linear extension of the causality relation, though no given process has full knowledge of this extension.

To maintain logical time each process maintains a clock integer. The value of this clock is initially zero. It is incremented as follows. For each atomic event it is incremented by one. For a transmit event it is incremented and appended to the message being transmitted. For a receive event it is set to the maximum of the received clock value plus one and the local clock value plus one. It is fairly easy to prove that this algorithm will satisfy the above clock requirement [46]. It should be noted that any non-negative increment and any initial value will suffice for the algorithm to be correct. Thus, a trivial application is to adjust real-time clocks so that events are time-stamped in a manner consistent with the partial order of execution. An example of Lamport timestamps is shown in Figure 3.4.

In distributed observation we do do not always have the capacity to arbitrarily alter the system code. If we wish to determine Lamport time for events in a partial-order data structure, absent this capacity to maintain it in processes, we may do it as follows:

$$\mathcal{L}(e) = \begin{cases} \max\left(\forall_{f <: e} \; \mathcal{L}(f)\right) + 1 & \text{if } \exists_f \; f <: e \\ 1 & \text{otherwise} \end{cases} \tag{3.5}$$

where $\mathcal{L}(e)$ is the Lamport time of event e. We introduce this notation as we will use it for the remaining logical-time techniques. Our motivation for this is that in some of the techniques that follow it would be quite expensive to have processes maintain logical time, and in one case it is not clear how this could be done at all.

3.1.2.2 VECTOR TIME

Given two events e and f with $\mathcal{L}(e) < \mathcal{L}(f)$ the only thing that can be concluded is that $f \not\preceq_{\mathcal{E}} e$. We cannot determine with Lamport time whether $e \parallel_{\mathcal{E}} f$ or $e \prec_{\mathcal{E}} f$. The reason is that Lamport time is imposing a total order on the partial order. From a directed-graph viewpoint, what is going on with Lamport time is that the edges being added to the graph are not simply reducing the precedence-test search time by adding edges implied by the transitive closure of the partial order. Rather, Lamport time is extending the partial order, adding edges to the graph not implied by the partial order.

The remaining logical-time techniques are able to determine precedence between events. They are given the name vector time, as they use a vector or an associative array rather than a single integer to represent the timestamp.

FIDGE/MATTERN TIMESTAMPS

The Fidge/Mattern timestamp [18, 19, 48, 59] is designed as follows. Each event e is assigned a vector timestamp $\mathcal{FM}(e)$ in terms of the events it covers. First, for event f, define $\mathcal{FM}'(f)$ as:

$$\mathcal{FM}'(f)[p] = \begin{cases} \mathcal{FM}(f)[p] + 1 & \text{if } p = \phi(f) \\ \mathcal{FM}(f)[p] & \text{otherwise} \end{cases} \tag{3.6}$$

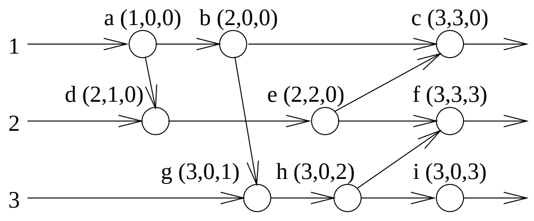


Figure 3.5: Fidge/Mattern Timestamps

(Recall that $\phi(f)$ maps event f to its process.) Then the Fidge/Mattern timestamp of e is:

$$\mathcal{FM}(e) = \max\left(\forall_{f <: e}\, \mathcal{FM}'(f)\right) \tag{3.7}$$

where $\max$ is the element-wise maximum of a set of vectors. A minimal event (that is, one that covers no other events) is assigned a vector as follows:

$$\nexists_f f <: e \Longrightarrow \mathcal{FM}(e)[p] = \begin{cases} 1 & \text{if } p = \phi(e) \\ 0 & \text{otherwise} \end{cases} \tag{3.8}$$

Precedence testing between two events, e_i and e_j, can be determined by the equivalence:

$$e_i \to e_j \Longleftrightarrow \mathcal{FM}(e_i)[\phi(e_i)] < \mathcal{FM}(e_j)[\phi(e_i)] \tag{3.9}$$

An example of Fidge/Mattern timestamps is shown in Figure 3.5.

Note that the timestamp can be implemented either as an associative array or, more simply, as a vector of size equal to the number of processes. The associative-array implementation requires, in the worst case, twice the space of the vector approach. The precedence test is constant time in both cases. As with Lamport time, any non-negative increment in Equation 3.6 and any initial values in Equation 3.8 that satisfy the requirement that the $p = \phi(e)$ value be the larger, will suffice for the algorithm to be correct. The desirability in incrementing by one and having the initial elements as one and zero is that the vector then also represents the number of events that are causally prior in other processes. There are variations on this specific vector timestamp, including extensions to support event abstraction, trace clustering and synchronous events (*e.g.*, [12, 64]). The commonality in all of these variations is that they have the same space consumption, they can determine precedence in constant time, and they can determine the greatest-predecessor events within each trace. Indeed, from the directed-graph viewpoint, they are precisely a set of edges connecting each event to its greatest predecessor in each trace.

If Fidge/Mattern clocks were implemented in a distributed system, each process would maintain a vector of size N and would append that vector to every message that was transmitted. The rules for maintaining such clocks would be essentially as described above. The only point of note is that the increment of Equation 3.6 would take place prior to appending the vector to the transmitted message. The maximum calculation of Equation 3.7 would be between the local and the received vector timestamps.

FOWLER/ZWAENEPOEL TIMESTAMPS

The Fowler/Zwaenepoel timestamp [20] is defined as follows. Each event e is assigned a vector timestamp $\mathcal{FZ}(e)$ in terms of the events it covers. For all events f, covered by event e, we define $\mathcal{FZ}'(f)$ thus:

If $\phi(e) = \phi(f)$ then:

$$\mathcal{FZ}'(f)[p] = \begin{cases} \mathcal{FZ}(f)[p] + 1 & \text{if } p = \phi(f) \\ \mathcal{FZ}(f)[p] & \text{otherwise} \end{cases} \tag{3.10}$$

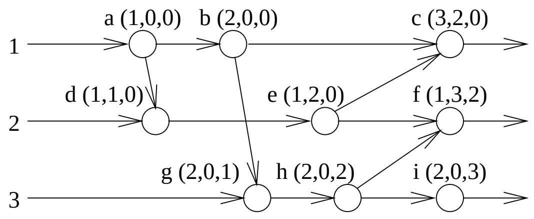


Figure 3.6: Fowler/Zwaenepoel Timestamps

Otherwise:

$$\mathcal{FZ}'(f)[p] = \begin{cases} \mathcal{FZ}(f)[p] & \text{if } p = \phi(f) \\ 0 & \text{otherwise} \end{cases} \tag{3.11}$$

Then the Fowler/Zwaenepoel timestamp of e is:

$$\mathcal{FZ}(e) = \max\left(\forall_{f <: e}\, \mathcal{FZ}'\,(f)\right) \tag{3.12}$$

A minimal event is assigned a vector in the same way as the Fidge/Mattern vector, described in Equation 3.8.

There is no specific precedence test for Fowler/Zwaenepoel timestamps as their intended application was causal distributed breakpoints. In that application, what is determined is the set of greatest predecessors within each trace. This is done by recursive search through the events referenced in the timestamp. Each branch of the search is terminated if its timestamp does not indicate a greater predecessor than is currently known. Thus, in Figure 3.6, which shows an example of Fowler/Zwaenepoel timestamps, if we wished to determine the greatest predecessors of event f in each trace, we would tentatively identify events a, e and h. We would visit these events in turn. The first two would turn up no additional information, but visiting event h would identify event b as being a greater predecessor in the first trace than event a. We would then visit event b at which point the search would terminate. Given the ability to determine greatest predecessors by trace we can answer the precedence question. However, if we only wished to answer a precedence question we may be able to terminate the search earlier. For example, we can determine $a \preceq_{\mathcal{E}} f$ just by the Fowler/Zwaenepoel timestamp of f. The cost of determining precedence is in the worst case linear in the number of messages, and in the best case constant time. The cost of determining greatest predecessors by trace has the same worst case, and in the best case $O(N)$, where N is the number of traces, as an event on every trace must be visited in the search. We do not know the cost of these operations for typical computations, nor are we aware of any study of this issue.

As with the Fidge/Mattern timestamp, we may implement the algorithm using either an associative array or a vector of size equal to the number of traces. In this case it is probably better to use the associative array, as the size of the array is proportional to the number of traces that directly communicate with the trace in which the event being timestamped occurs. The reason is that the Fowler/Zwaenepoel timestamp is only tracking direct dependencies

From the directed-graph viewpoint, the Fowler/Zwaenepoel timestamp represents a set of edges connecting each event e to the greatest event in a trace that has communicated directly with the trace that e is in. These greatest events may not be the greatest predecessors of e, because the timestamps do not capture transitive dependency, but only direct dependency. Also note that these greatest events must be either transmit or synchronous events.

Fowler/Zwaenepoel timestamps can be implemented within a distributed system by appending just the local event identifier to each message transmitted. This can be used in the maximum calculation of Equation 3.12 with the other vector elements implicitly being set to zero.

JARD/JOURDAN TIMESTAMPS

Jard/Jourdan timestamps [31] are an extension of Fowler/Zwaenepoel timestamps. As with both Fidge/Mattern and Fowler/Zwaenepoel timestamps, they were intended to be used within the distributed system, rather than as a mechanism for querying the partial-order data structure. Since it is not entirely obvious how they might be applied to our

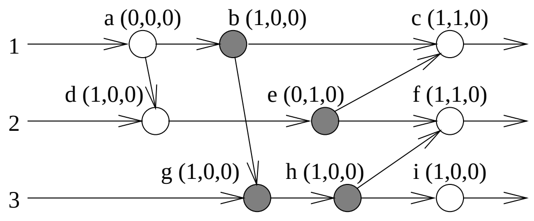


Figure 3.7: Jard/Jourdan Timestamps

use, we will describe their implementation within a distributed system and explain the problems of their use in our application.

Jard/Jourdan timestamps introduce two key concepts: observability and pseudo-direct dependence. Not all events are considered to be worth observing, and so events are divided into observable and unobservable ones. However, the model of distributed computation is such that any event,[2] whether observable or not, can cause a transitive dependency that we wish to capture. For this reason, Jard/Jourdan timestamps introduce the notion of pseudo-direct dependency. An event e is pseudo-dependent on an event f (written $e \ll f$) if there is a path from e to f with no observable events on that path. The intention is to create Fowler/Zwaenepoel timestamps, though just for the observable events.

As with the previous cases, each event e is assigned a vector timestamp $\mathcal{J}(e)$ in terms of the events it covers. For event f we define $\mathcal{J}'(f)$ as follows:

If f is observable then:

$$\mathcal{J}'(f)[p] = \begin{cases} \mathcal{J}(f)[p] + 1 & \text{if } p = \phi(f) \\ 0 & \text{otherwise} \end{cases} \tag{3.13}$$

Otherwise:

$$\mathcal{J}'(f) = \mathcal{J}(f) \tag{3.14}$$

Then the Jard/Jourdan timestamp of e is:

$$\mathcal{J}(e) = \max\left(\forall_{f <: e}\, \mathcal{J}'(f)\right) \tag{3.15}$$

A minimal event is assigned a zero vector, whether observable or not. The precedence test between observable events is the same as that of the Fowler/Zwaenepoel timestamp. Since the intention was to use these within the system, the only events that would be collected for observation would be observable events, and thus no precedence test exists for unobservable events. It is unclear what would be needed to create such a test. In the absence of such a test, this technique could not be used for a partial-order data structure.

An example of Jard/Jourdan timestamps is shown in Figure 3.7. The unshaded events are the observable events.

The Jard/Jourdan timestamp is then Fowler/Zwaenepoel-like with respect to the observable events. If all events are observable it degenerates to the Fowler/Zwaenepoel timestamp with one exception. The difference is that the Fowler/Zwaenepoel timestamp maintains information in its vector about dependencies in other traces even after intervening events. The Jard/Jourdan timestamp only maintains information about immediate direct dependencies. As a result, the Jard/Jourdan timestamp is likely to have a notably longer search time in determining precedence if all events are observable.

We may implement the algorithm using either an associative array or a vector of size equal to the number of traces. As with Fowler/Zwaenepoel, it is probably better to use the associative array, for essentially the same reason.

From the directed graph viewpoint, the Jard/Jourdan timestamp represents a set of edges connecting each event e to the greatest observable event in a trace that has communicated pseudo-directly with the event e. As with

[2]Unary events excepted, but then their existence is predicated on their usefulness.

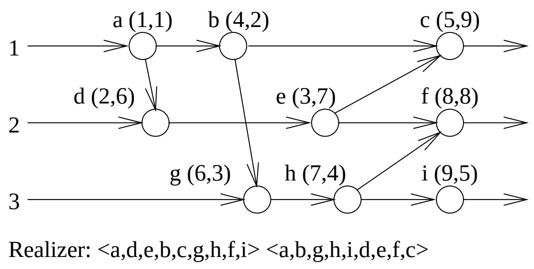


Figure 3.8: Ore Timestamps

Fowler/Zwaenepoel, these greatest events may not be the greatest predecessors of e, both because the timestamps do not capture transitive dependency and because it ignores unobservable events.

Jard/Jourdan timestamps can be implemented within a distributed system by appending value of $\mathcal{J}'(t)$ to the message transmitted by event t. This can be used in the maximum calculation of Equation 3.15. Since the size of this vector can be as large as the number of traces (as it grows with unobserved events across multiple traces), Jard/Jourdan offers a bound by inserting a null event that is observed for the purpose of shrinking the vector. How significant this is remains to be seen. Their paper on this subject has just one program and two experiments.

ORE TIMESTAMPS

The Ore timestamp [54, 64] is unique among vector timestamps in that it does not take a process view. Rather, it is based on having a realizer for the partial order. The linear extensions are arbitrarily ordered from 1 to d, where d is the number of extensions. Each event e in each extension l_i is assigned an integer $\mathrm{leid}(l_i, e)$ to indicate its position within that extension. The following equivalence must hold for this assignment:

$$e_j \rightarrow_{l_i} e_k \Longleftrightarrow \mathrm{leid}(l_i, e_j) < \mathrm{leid}(l_i, e_k) \tag{3.16}$$

The Ore timestamp $\mathcal{O}(e)$ for event e is defined as:

$$\forall_{i:0<i\leq d}\, \mathcal{O}(e)[i] = \mathrm{leid}(l_i, e) \tag{3.17}$$

Precedence testing between two events, e_j and e_k, can be determined by the equivalence:

$$e_j \rightarrow e_k \Longleftrightarrow \forall_{i:0<i\leq d}\, \mathcal{O}(e_j)[i] < \mathcal{O}(e_k)[i] \tag{3.18}$$

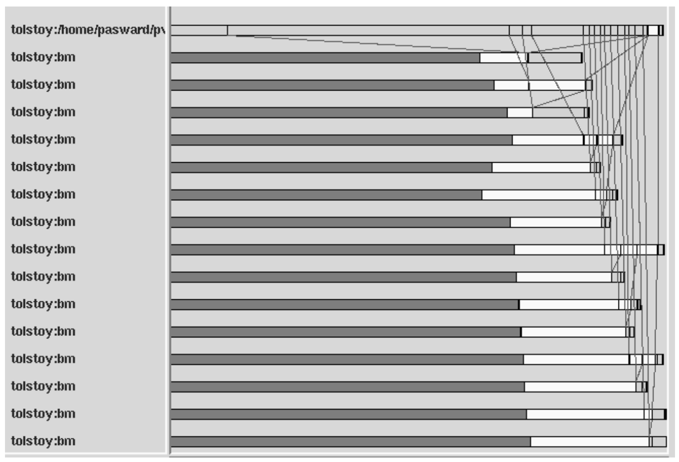
An example of Ore timestamps is shown in Figure 3.8.

Note that the vector size is equal to d, while the test is $O(d)$. The value of d can be as large as $O(n!)$, where n is the number of events in the computation. For any reasonable implementation of a realizer it would be no larger than the width, which can be no larger than the number of traces. It can be as small as the dimension of the partial order. We know of no online algorithm to build a realizer.

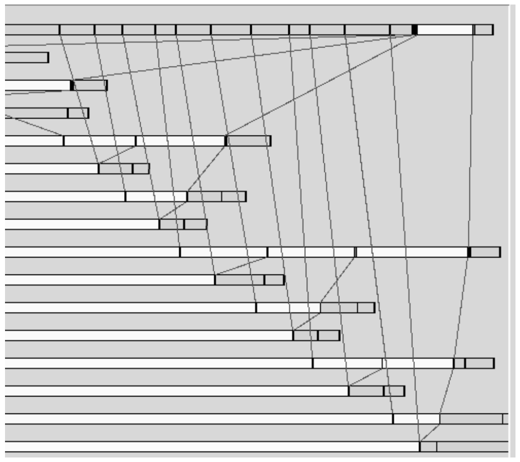
3.2 VISUALIZATION

Visualization of distributed computations is an enormous area of research. It is certainly beyond the scope of this document, or any piece of research, to make all systems for visualizing distributed computations scalable. We will therefore limit our scope to the spatial visualization of the partial-order of execution, with a specific focus on abstraction issues. This then excludes several areas, including animations, program graphs, and statistical displays,[3] among

[3]That is, unless the display is some statistical representation of the partial-order data structure (See, for example, the work of Dan Reed [52]). This class of visualization may be a bar graph, kiviat diagram, or other display, representing such variables as processor utilization and traffic volume.



(a) Wide Scale



(b) Focussed Scale

Figure 3.9: XPVM Displays at Different Time Scales

others. We will also not consider application-specific displays, though we will look at the generalizable features of environment-specific visualizations.

There is still a large amount of material for this constrained form of visualization. This work includes Stone's concurrency maps [63], the thread-lifetime view of Falcon [17, 24, 35], network-visualization tools [56], the Interactive Distributed Debugger [26], Moviola [47], GOLD [61], ATEMPT [37], the space-time diagrams of ParaGraph [27], XPVM [34], POET [44, 69], and many others. In addition to these systems, there is a large body of literature on graph drawing [15]. The specific categories that these forms of partial-order display fall into are Hasse diagrams and layered digraphs. We will not discuss that generic literature, though it may shed light on problems that occur in large displays. Also, rather than attempting to describe each of the specific visualization systems separately, we will discuss the collective features of the displays as they pertain to visualizing the partial-order data structure.

There are several aspects to the visualization of the partial-order data structure. The essence of the problem is to draw the directed graph of the transitive reduction of the partial order. Given this, there are various issues involved. The graph may be drawn based only on the partial ordering, based only on the real-time of the event, or based on the combination of these. Since no non-trivial computation induces a partial-order that can be displayed in entirety in a single screen, scrolling support is needed within the display. Clearly this scrolling will be in the time dimension. For computations involving a significant number of processes, the trace dimension will also have scrolling. An alternate approach for dealing with large computations is to display the entire computation on the screen and allow zoom-in for detailed examination. The display may be generated offline, after the event information is collected, or online, during the course of the computation. If a visualization tool cannot answer the questions that a partial-order data structure can answer (other than by careful observation), we then describe it as an output-only display. Finally, the user may wish to abstract portions of the display for various reasons. We will now describe these aspects in more detail, both in terms of the features and the required algorithms to implement those features.

3.2.1 DISPLAY TYPE

The most elementary visualization of the partial-order data structure that we are aware of is the XPVM [34] space-time view. It draws a horizontal line for each PVM task. Communication between tasks is represented by a line drawn between the two tasks, with the endpoints of the line determined by the "real-time" of the transmission and reception events. We have placed "real-time" in quotation marks as, at least with the earlier versions, no attempt is made to

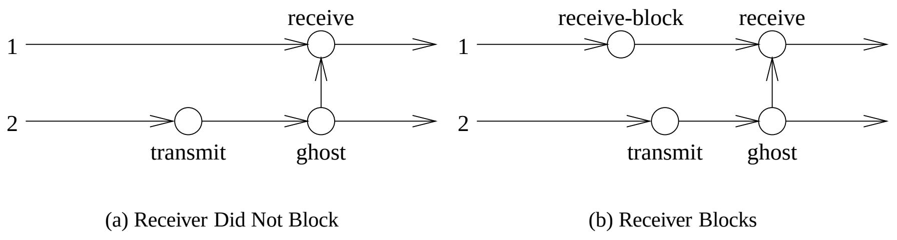


Figure 3.10: POET's Synchronous Real-Time Display

adjust the timestamp values at each PVM task to ensure that messages are not viewed as traveling backwards in time. Such messages are referred to as *tachyons*. This problem of tachyons, combined with the absence of any other visual cue (such as an arrow), means transmit and receive events are not distinguished in the basic display. In fairness to XPVM, there is support in the GUI to determine this information on a message-by-message basis.

Real-time displays suffer from other problems than tachyons. The real-time of events may be such that they are clustered in some locations, while sparse in others. An example of this phenomenon is shown in Figure 3.9(a), which illustrates a master-to-slaves distribution followed by a binary merge. Given that most, though not all, of the communication is at the right hand end of the display, Figure 3.9(b) shows a focussed display on this portion. Unfortunately, it now misses four of the earlier transmit events from the master PVM task, even though there is, in principle, room to show them. An alternate way of dealing with this issue is to have a time scale that is sufficient to distinguish events, and then allow for breaks within the time-axis of the display. The POET system uses this in its real-time display.

Synchronous events present a third problem for real-time displays. Logically all constituent events that compose a synchronous event occur simultaneously. Physically, the constituent events will occur at different times.[4] XPVM does not deal with this problem at all. It has a specific target environment, *viz* PVM, and it treats that environment as though it does not support synchronous events. In reality there are PVM operations that can be viewed as synchronous operations. For example, the pvm_barrier() function implements a barrier synchronization operation. This function is implemented by a group of tasks sending barrier messages to the group coordinator. When the coordinator has received the requisite number of barrier messages, it multicasts a continuation message to the group tasks which will block at the barrier pending the arrival of this message. XPVM does not visualize any of this communication. Rather it simply indicates in each task that it is performing a pvm_barrier() function call. The starting and ending time of these will vary by task, though in all tasks there will be an overlap point when the barrier is seen to be reached at the coordinator.

POET only supports synchronous pairs, which effectively map to synchronous point-to-point transmit and receive events. POET's solution is to show a unary "transmit" event at the real-time of the transmit. In addition, a ghost transmit event is displayed on the transmitting trace at the real-time of the receive event. An example of this is shown in Figure 3.10(a). If the receiver blocks prior to the transmit event this will be indicated as shown in Figure 3.10(b). To form a complete synchronous RPC the receiver will return the results of the computation in a mirror image operation of Figure 3.10(a).

We are not aware of other monitoring systems that provide a combination of real-time and synchronous event support. In particular, we are not aware of any system that supports synchronous events comprising more than two constituent events, with or without real-time support. It is not clear how the POET solution could be extended to deal with these more complex synchronous events. A possible solution is to use the method used for abstract event display (see Section 3.2.4.1) though this might cause confusion as, in general, abstract events are not viewed as occurring at a single logical time.

[4]Usually. In the case of a language such as Occam executing on Inmos transputers the local and remote events occur, for practical purposes, simultaneously.

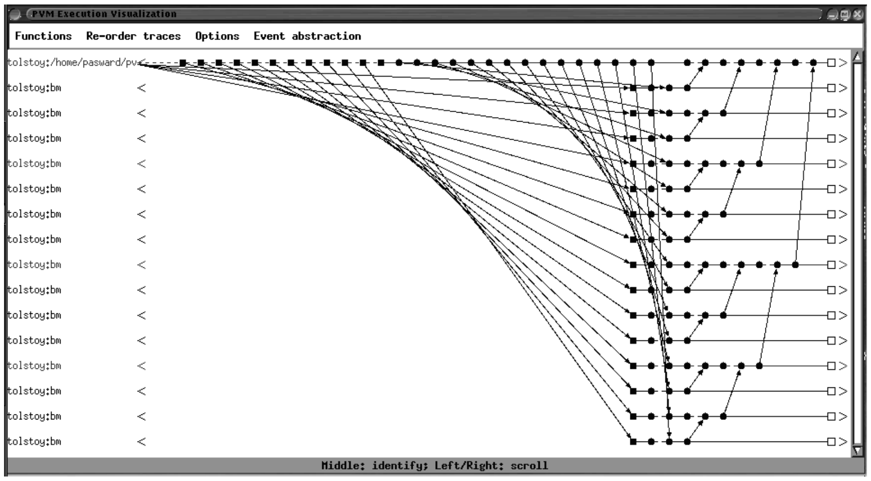


Figure 3.11: POET Partial-Order Display

Visualization based only on real-time requires little data-structure support. It must record the desired event information and the real-time of the event occurrence. There need not be any specific connection between events recorded, as is evident in XPVM.

While displays based on the real-time of events may be useful in some applications, such as performance analysis, it can hinder others. If the objective is debugging or program understanding, then the logical ordering of events (*i.e.*, the partial order of execution) is often more relevant than the real-time ordering of events. Figure 3.11 shows a partial-order display of the PVM computation whose real-time display was illustrated in Figure 3.9. Note that this is one of many possible displays, as partial-order displays are not, in general, unique. We believe that this display more clearly delineates the logical execution of the program.

The primary issue in creating a partial-order display is choosing an event placement given the wide variety of possibilities afforded by the partial-order of execution. The most common method of event placement (used, for example, in ATEMPT [37], the Conch Lamport view [71], PVaniM [73], and others) is to maintain Lamport clocks in the distributed computation and use them to timestamp events. Event placement is then performed in a grid fashion, with the location on the grid being determined by the trace number and Lamport timestamp of the event. Figure 3.12 shows an example of this type of display, again, for the same computation as is shown in the prior figures.

One of the primary problems with this method of event placement is that it yields a rigid display, as Lamport time is totally ordered. Thus events in one trace may appear to occur substantially prior to events in another trace, even though they may be causally concurrent. The Xab approach, or the dual timestamping method of PVaniM, go some way to dealing with this, by moderating the Lamport time by the real-time. We will discuss that method below.

A second technique is to use the idea of phase time [47]. The approach is to divide the events into collections of concurrent events, and to display each collection as occurring at the same logical time. Figure 3.11 illustrates such a phase-time view, though phase time is not a feature of POET *per se*. A key problem with this approach is it is not possible in general to implement it. Specifically, although we claimed that Figure 3.11 shows such a display, this is not strictly true. All of the first events in all of the slave tasks are concurrent with all of the second set of transmit events in the master task (those transmit events that are received at the third event in each slave task). This is not apparent in this display. While this could be corrected for, what cannot be corrected for is the initial events in the slaves themselves. They are concurrent with each other, but not all are concurrent with the first sequence of transmit events from the master. Some are successors and some are concurrent.

Stone's concurrency maps deals with the above problem by showing regions in each trace, rather than individual events, in a similar fashion to the XPVM display. Regions have starting and ending points within the time-axis. If a region R_1 in one trace is causally prior to a region R_2 in another trace then the end point of R_1 is placed at a lower

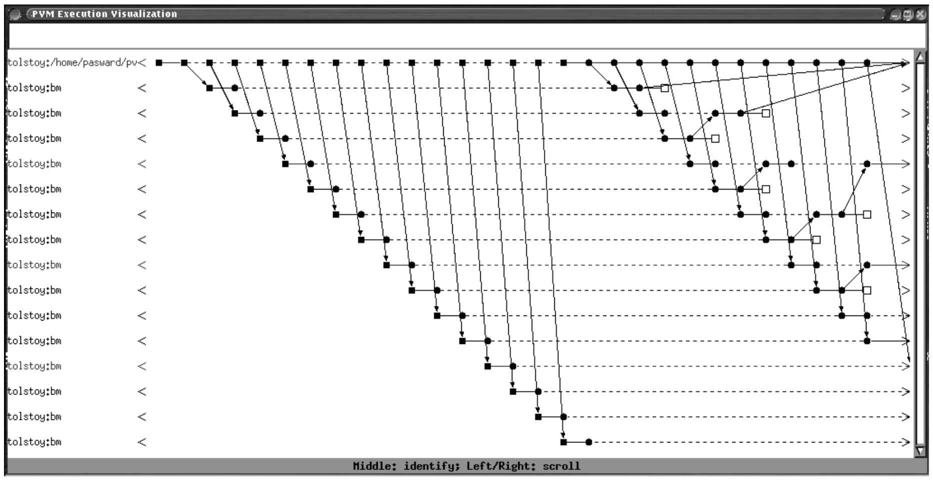

PVM Execution Visualization
tolstoy:/home/pasward/pv
tolstoy:bm
Middle: identify; Left/Right: scroll

Figure 3.12: Lamport-Style Display

time location in the time-axis than the start point of R_2, if this is possible. The *caveat* is necessary because it is not always possible to satisfy this requirement, as illustrated by Summers [64]. One valuable feature of the concurrency map is the ability to illustrate an abstract set of events within a trace that is in various precedence relationships with an event or region in other traces.

The Falcon system [17, 24, 35] thread-lifetime view and XPVM both use a method that is akin to concurrency maps, though with substantially different implementation approaches. The XPVM method, as previously noted, is not partial-order-based. The Falcon technique uses a rule-based approach. Different event types have rules associated with them to determine precedence of events as they arrive at the visualization engine. For example, there is a rule that a mutex operation must be preceded by a thread-create operation. It seems doubtful if this technique could work in general, as the complexity of these rules will grow with event types and the likelihood of an error, resulting in an incorrect precedence display, would then be high.

Other techniques for implementing partial-order displays include maintaining vector clocks within the distributed computation, using causal message ordering, and topological sorting of the events based on event identity. The GOLD system [61] uses Fowler/Zwaenepoel dependency vectors within the computation, though it is unclear what they gain by this. POET uses Fidge/Mattern timestamps, but not within the computation. Rather, it computes them within the monitoring entity for the purposes of answering partial-order queries. Parulkar *et al* [56] use causal message ordering within the network monitoring system. Presumably the monitoring station must be a part of the causal-message-ordering system, and then it will never receive events out of causal order. We do not believe such a system is practical because of the high overhead of ensuring causal message ordering (it is $O(n^2)$ in the number of processes [10]). The minimal approach necessary is to order events within traces and match transmit and receive events. In other words, the monitoring entity should perform a topological sort on the events. ParaGraph [27] does this after all of the event information is collected. There is no fundamental reason why this cannot be performed online. Each event would be sent to a priority queue determined by trace id. Events processed from the priority queues would be causally ordered if all prior events in the trace were already processed and receive events had had their corresponding transmit event processed. It is therefore unclear what value is obtained by using other techniques if causal ordering of the display is the only requirement.

The basic problem with partial-order-based displays is that no single view will suffice. The reason is that any single view must order events that are unordered. A reasonable partial-order-based display must allow scrolling within the display in such a manner that events can slide relative to other events in recognition of their unordered nature. We will discuss scrolling in detail in Section 3.2.2.

The final display technique used is to combine both real-time and partial-order-based displays. There are two features of this combined approach. First, some systems provide two data displays, one real-time-based and the other partial-order-based. This is the approach taken by POET and ATEMPT [37]. The second aspect is correcting the real-time of events to ensure that it is consistent with the partial order of execution. Systems that perform such corrections include the Xab system [5, 6], POET and PVaniM [72]. There are several ways in which these corrections may be performed. The simplest is to apply the Lamport clock algorithm [46] to the real-time of events. However, it has been observed that this can significantly distort real-time values, and so Taylor and Coffin developed an alternate algorithm which minimizes such distortions [68]. The PVaniM system uses a dual-timestamp approach [72]. It presents partial-order displays using a Lamport timestamp and Gantt charts (essentially concurrency maps) using a Lamport-adjusted real-time timestamp. It is not entirely clear what the benefit is of maintaining the Lamport timestamp.

Most of the systems we have discussed are tied to specific target environments. Thus, XPVM is tied to PVM, Falcon is specific to Cthreads, and so forth. However, the features of these systems that we have discussed are generalizable beyond their specific target. POET has achieved a high degree of target-system independence [70], and can work with almost a dozen different target environments.

3.2.2 SCROLLING AND ZOOMING

Any non-trivial computation cannot be meaningfully displayed within a single screen. There are two (somewhat orthogonal) solutions to this problem. First, a detailed display may be created, and then the user can scroll within that display. Alternately, the complete display may be shown, and the user given the opportunity to zoom within that display. This problem affects both the time and trace dimensions of the display. We will deal first with time-dimension scrolling.

The vast majority of systems present a text-editor-like approach to this problem. That is, conceptually they create a fixed display of the entire partial-order. They present a portion of it, which is effectively a window over that fixed display. Scrolling is then a question of moving the location of the window over the fixed partial-order display. Given that creating such a complete display is neither generally feasible nor necessary, various optimizations are applied. Usually only a portion of the partial-order display is created, with the window into a subset of that. As long as the scrolling operation is within the portion of the display that has been computed, it will be a fairly quick operation. When it moves outside of that range a new display will have to be computed. For a real-time based display this would appear to be the preferred form of time-dimension scrolling.

Taylor [65, 66] argues fairly effectively that this is the wrong model for scrolling partial-order based displays. Specifically, in providing only a possible ordering of events, or an ordering consistent with the partial-order of execution, it effectively treats the partial-order as though it were a total order. This will give a user an incorrect view of the system. Taylor's model is to drag an event to one or other end of the currently visible portion of the time dimension. Other events are then moved as required by the partial-order constraints. Note that other events will only move if they have to, and only by as much as they have to. Thus, if an event is concurrent to the one being dragged to the edge, it will likely change its position relative to the dragged event.

In the trace dimension most systems also treat the display in a text-editor-like fashion. As with the time dimension, this is likely not a correct approach. Traces do not have order with respect to each other, except insofar as is afforded by the structure of the software being monitored. The default ordering of traces in most systems is determined by the order of arrival of events from the monitoring entity, which, other than partial-order constraints, is determined by the arrival order from the distributed computation. This may not be the best ordering possible, though it is necessary for an online display algorithm. Alternate trace orderings are offered by various systems. Both XPVM and POET allow arbitrary trace re-ordering by the user. In addition, POET offers the ability to optimize trace location and move by precedence. Trace-location optimization minimizes the apparent communication distance. This potentially corresponds to the structure of the distributed computation software in that entities that communicate frequently are likely related, while those that do not do so are likely less-connected in the software structure. Move by precedence moves traces relative to some event. Specifically, traces that have causal predecessors or successors to that event are moved closer to the trace containing that event.

The alternate solution to scrolling is zooming. Zooming should not be considered to be simple magnification [55], though many systems limit it to this. Rather, it should provide additional information. A very good example of such a zooming system is the System Performance Visualization Tool for the Intel Paragon [30]. The full view shows the processors in the system, with colouring to reflect activity. Zooming in shows actual load averages on a small number of processors and message traffic load to neighbouring processors. Further zooming in reveals detailed information about performance within a processor.

With this view in mind, there are several examples of zooming. Most have not traditionally been considered to be zooming, though they are in this wider view. XPVM provides traditional magnification zooming, though the original image is then lost. The information-mural approach [33] retains the original image of the entire computation, while allowing a zoom-in on a portion. While it appears that this could provide additional information on zoom-in, rather than simple magnification, it does not appear that it does. Further, it does not appear to be the focus of their work.

There are at least three other common features of monitoring systems that can be described as forms of zooming: trace clustering, event abstraction and source-code display. We will discuss trace clustering and event abstraction in detail in Section 3.2.4. Source-code display could be viewed as the highest level of zoom-in, though it is usually implemented in the form of querying the display, and so we will discuss it in the next section.

3.2.3 Querying Displays

While the primary purpose of a visualization is to present information to a user in such a manner as to allow the information to be readily apparent, for a non-trivial computation this is not possible. Some information can be made clear, but only at the expense of making other information obtuse. All of the information that a user needs to know cannot be shown in a single display. Rather, the display of information prompts questions from the user. Some of these questions, such as “What happens if I drag this event to the left?”, may be resolved by scrolling around the display. Others, such as “What does this look like up close?”, require zooming in or out. Still other questions, such as “What is causally prior to this event?”, do not really fit either the zooming metaphor or scrolling, and must be asked more directly. The large variety of questions can be grouped into two broad categories: questions pertaining to individual events and those concerning the relationship between events. We will now describe the ways in which various systems deal with these two groups of questions.

With the exception of scrolling capabilities, a significant fraction of visualization tools cannot answer questions from either of these groups. Systems such a ParaGraph simply present a display and hope that it is of value to the viewer. Such an approach we term an output-only display. It requires little data-structure support, and scrolling within it is, of necessary, text-editor-like. The typical approach in implementing such a system is to have a separate visualization engine that acquires data from the monitoring entity but does not otherwise interact closely with it.

The next level, answering questions about events, is provided for in a variety of ways. A typical minimal query capability is provided by XPVM. It allows a user to click on an event and it will indicate the task identity, the PVM function and parameters that the event represents, and the starting and ending times of the function call. ATEMPT allows the user to display the line of code that the event corresponds to, providing a simple text window into the relevant source-code file. Note that showing lines of source code does not provide the dynamic information that XPVM provides. The combined functionality of these features is achieved in Object-Level Trace (OLT [29]) which allows a direct connection between the visualization tool and a sequential debugger. As with output-only displays, little data-structure support is required to implement these capabilities. Specifically, each event requires the relevant information to be collected and stored with the event.

The most complex query types pertain to relationships between events. XPVM provides only the crudest support for this, in that a user can query messages. The result of this query is that XPVM indicates which task the message is from, which it is to, the transmission time, the message tag and the number of bytes. While this gives the illusion of value, in reality it provides no more information than is available from the simple event querying. As with event querying, it requires only that the relevant information is collected with the transmit event, with the exception of the message receipt time which must be collected from the receive event. The POET system allows several types of user query.

It can display all successor, concurrent, and predecessor events to a given event. Pattern-searching and execution-difference operations, described previously, are displayed through the visualization engine. These operations require the full partial-order data-structure capability described previously. Finally, the PARASIT systems [38], which is built on top of the ATEMPT system, allows a user to reorder race message outcomes and re-execute to determine the effect of this change. This requires replay capability. In systems such as POET and PARASIT the visualization engine interacts closely with the monitoring entity, rather than being a passive receiver of data.

3.2.4 ABSTRACTION

Abstraction is the act of ignoring the irrelevant to focus on the essential. What is irrelevant and what is essential is very much a function of the user's current requirements. We shall discuss some of the user's criteria in the subsequent sections. In the context of our previous discussion, abstraction can be seen as zooming in on some portion of the display, or zooming out on the whole. It is zooming in because it focuses on essentials. It is zooming out because details are omitted. Particular abstractions can be seen more as one or other of these views.

There are several desiderata in abstraction. The primary purpose is to aid user understanding, and insofar as this is achieved, the abstraction performed is a good one. In addition to this, in principle abstraction can also help the distributed management systems cope with excessive data. Specifically, it should be able to limit the amount of data that must be dealt with in various algorithms and operations. While some systems use this in various aspects, we know of no current system that can use abstraction for the efficient realization of a partial-order data structure. As such, at the core all of the data must be processed.

There are currently two main abstraction techniques used in distributed observation and control. These are event and trace abstraction. It is not clear if there are other ways in which abstraction may be performed.

3.2.4.1 EVENT ABSTRACTION

At a very basic level, event abstraction is the act of creating abstract events from primitive ones. In the most general definition, an abstract event is a set of primitive and other abstract events. It is usually implicitly assumed that an abstract event cannot contain itself, either directly or through a recursive chain. This yields a hierarchy of abstract event sets, with primitive events at the base:

$$\mathcal{A}_N \subseteq \begin{cases} 2^{\bigcup_{i=0}^{N-1} \mathcal{A}_i} & N > 0 \\ \{\{e\} \mid e \in \mathcal{E}\} & N = 0 \end{cases} \tag{3.19}$$

$\mathcal{A}_N$ is the set of abstract events that can be defined in terms of primitive events and abstract events in the abstract events sets 1 through $N-1$. These sets may be combined to produce the set of all abstract events:

$$\mathcal{A} = \bigcup_{i=0}^{\infty} \mathcal{A}_i \tag{3.20}$$

Note that this equation abuses notation as there cannot in fact be infinitely many abstract event sets.

It is frequently simpler to consider all abstract events as composed of primitive events. This is a reasonable possibility since, insofar as they are composed of other abstract events, those would be composed (ultimately) of primitive events. We may define this set of abstract events as:

$$\mathcal{A} \subseteq 2^{\mathcal{E}} \tag{3.21}$$

In the following discussion, we will presume that we are working with this definition when defining things in terms of the primitive event set. We will use the abstraction hierarchy of Equation 3.19 otherwise.

Finally, it may be desirable to restrict the hierarchy of Equation 3.19 such that abstract events at level N can only be composed of abstract events at level $N-1$, defining level 0 to be the set of primitive events, thus:

$$\mathcal{A}_N \subseteq \begin{cases} 2^{\mathcal{A}_{N-1}} & N > 0 \\ \{\{e\} \mid e \in \mathcal{E}\} & N = 0 \end{cases} \tag{3.22}$$

We believe that such a definition would not significantly affect the use of event abstraction, but would potentially be useful, augmented with further structure over the set, in the implementation of abstract events. For the remainder of this document we will use the letter a, possibly sub- or superscripted, to indicate an abstract event. We will use the subscript to refer to the abstract set level in which the event resides. Thus $a_i \in \mathcal{A}_i$. We will also describe an abstract event that is composed of a single event as "trivial." Note that every abstract event defined at level i exists as a trivial abstract event in every higher level of the abstraction hierarchy. An abstract event that is composed of two or more events is "non-trivial."

In addition to these simple definitions, various researchers have proposed further structure on abstract events. Most of this structure has revolved around two issues. The first issue is attempting to make abstract events in some sense atomic. What this means in practice is that after inputs are satisfied, the abstract event should be able to execute to completion, without an intervening dependency. This can be viewed as attempting to achieve synchronous events of the fourth type.

The second issue is the question of what precedence relationships should be defined between abstract events. There are several possibilities for extending the primitive precedence relation. The first requirement, satisfied by all proposed extensions, is that it suitably degenerate to the correct case for abstract events that are composed of single primitive events. That is:

$$\{e_i\} \prec_{\mathcal{A}_0} \{e_j\} \iff e_i \prec_{\mathcal{E}} e_j \tag{3.23}$$

where $\prec_{\mathcal{A}_0}$ is the precedence relation for abstract events in abstract set $\mathcal{A}_0$. In addition to this requirement, it is usually presumed that the precedence relation over all abstract events, $\prec_{\mathcal{A}}$, is the union of the precedence relations over the individual levels. It is not always clearly defined what precedence means between abstract events at differing levels in the abstraction hierarchy, or even if such a definition is meaningful. Given our previous definition of trivial events, we will presume that precedence is resolved by "raising" the lower abstract event to the same level in the hierarchy as that event with which it is being compared, and then using whatever precedence relation is defined at that level. The final assumption in most prior work is that the definition of concurrency is essentially unchanged from that of Definition 9, altering only the precedence relation from that over primitive events to that over abstract events. It is unclear whether this model of precedence and concurrency is a good one for abstract events. Specifically, where primitive events are atomic, non-trivial abstract events have duration. As such, it is not clear that their causal inter-relationship will form a partial order. Disclaimers aside, we will now describe the various forms of abstract-event precedence.

Kunz [39] lists two obvious possible definitions for abstract-event precedence. First, all primitive events in one abstract event must precede all primitive events in the other abstract event:

$$a^i \prec_{\mathcal{A}} a^j \iff \forall_{e_k^l \in a^i} \forall_{e_m^n \in a^j}\, e_k^l \prec_{\mathcal{E}} e_m^n \tag{3.24}$$

Note that while this definition is in terms of precedence of primitive events, it is is isomorphic to the definition based solely on abstract events:

$$a^i \prec_{\mathcal{A}} a^j \iff \forall_{a^k \in a^i} \forall_{a^l \in a^j}\, a^k \prec_{\mathcal{A}} a^l \tag{3.25}$$

This approach yields a partial order over the set of abstract events, which has some appeal. However, there are few abstract events for which precedence would then hold [13]. It would also require a modification of the definition of concurrency. The existing definition would imply that abstract events were concurrent even though there might be significant communication between constituent events. Indeed, the constituent events could be totally ordered, but the abstract events considered concurrent under the present definition (See Figure 3.13).

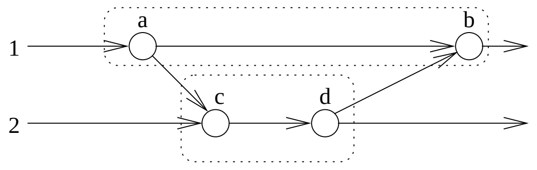


Figure 3.13: Ordered Primitive Events forming Concurrent Abstract Events

The second obvious approach is that some primitive event in one abstract event must precede some primitive event in the other abstract event.

$$a^i \prec_{\mathcal{A}} a^j \iff \exists_{e_k^l \in a^i} \exists_{e_m^n \in a^j} \, e_k^l \prec_{\mathcal{E}} e_m^n \tag{3.26}$$

Likewise, this is isomorphic to the equivalent definition using just abstract events. While this definition requires no modifications to the definition of concurrency, it has the interesting property that it is neither transitive, nor anti-symmetric. The Ariadne debugger [13] takes this approach as is, but defines the *overlaps* relation for such cases of symmetric dependency between abstract events. It does not appear to deal with lack of transitive dependency at all. Cheung [12], Summers [64], Basten [2] and Kunz [39] also take this view, but attempt to prevent symmetric abstract events from being created by imposing further structure on abstract events.

To add additional structure, both Cheung and Summers define the input and output event sets for an abstract event. These are as follows:

Definition 15 (Input Set: $I \subseteq a$**)** *The input set, I of abstract event a is the set of events in a which have an immediate predecessor that is not in a.*

$$I = \{i \mid i \in a \wedge (\exists_e \, e <: i \wedge e \notin a)\} \tag{3.27}$$

The output set is defined analogously:

Definition 16 (Output Set: $O \subseteq a$**)** *The output set, O of abstract event a is the set of events in a which have an immediate successor that is not in a.*

$$O = \{o \mid o \in a \wedge (\exists_e \, o <: e \wedge e \notin a)\} \tag{3.28}$$

Given these definitions, three classes of abstract events were defined: the central-event, the complete-precedence, and the contraction.

The central-event class of abstract events requires that there exist an event within the abstract event such that all input events precede it and all output events succeed it. This definition does ensure that the precedence relation over abstract events is both anti-symmetric and transitive. However, it appears to be excessively restrictive. A simple two-way information exchange between processes could not be formed into an abstract event, even though this might be desirable. The problem is the requirement for a single event through which all precedence must flow.

The complete-precedence class of abstract events relaxes this restriction and enforces only that all input events must precede all output events. This definition still ensures that the precedence relation over abstract events is a partial order. This is probably the weakest structural restriction that allows arbitrarily interconnected abstract events and still provides this guarantee. However, it is arguably too restrictive a definition. It does not, for example, permit multiple concurrent events to be a single abstract event. This particular usage of abstract events would be prevalent for parallel computations in which multiple processes perform the same actions on different data (the SPMD model).

Cheung studied a more general class of abstract event, contractions, which was first defined by Best and Randell [7]. The definition is as follows:

Definition 17 (Contraction)

1. *All primitive events are contractions*
2. *An abstract event a_i^j is a contraction if and only if*

   (a) $\forall_{a_{i-1} \in a_i^j} \; a_{i-1}$ *is a contraction*

   (b) $\forall_{a_i^k ; a_i^l} \; a_i^k <:_{\mathcal{A}_i} a_i^j <:_{\mathcal{A}_i} a_i^l \Longrightarrow \exists_{a_{i-1} \in a_i^j} \; a_i^k \prec_{\mathcal{A}_{i-1}} a_{i-1} \prec_{\mathcal{A}_{i-1}} a_i^l$

A contraction that has only one immediate predecessor and one immediate successor is a simple contraction. Systems of simple contractions retain the anti-symmetric and transitive properties with respect to precedence. As such they have some appeal. The EBBA Toolset [4] is an example of a distributed debugging system using simple contractions for abstraction. Strictly speaking they are less restrictive than complete-precedence events. The manner in which they are less restrictive is that there can be input events that do not precede output events and output events that are not preceded by input events. This appears to have no practical value. Contractions that are not simple cannot be arbitrarily interconnected and reflect the transitivity of the underlying event set. Summers created a set of rules for interconnection that would ensure transitivity, though it is not clear that this could be efficiently used in practice.

Given the problematic nature of ensuring transitivity, Kunz [39] and Basten [2] abandoned it as a requirement, but attempted to maintain anti-symmetry by defining the class of convex abstract events as follows:

Definition 18 (Convex Abstract Event) *An abstract event a_i is convex if and only if*

$$\forall_{a_{i-1}^j, a_{i-1}^k \in a_i \, ; \, e \in \mathcal{E}} \; a_{i-1}^j \prec_{\mathcal{A}} e \wedge e \prec_{\mathcal{A}} a_{i-1}^k \Rightarrow e \in a_i$$

Several notes should be made about this definition. First, the terminology is somewhat imprecise, since it is taken, essentially as is, from the relevant paper by Basten. He uses $\preceq$ though this will not alter the substance of the definition. He also does not clearly indicate exactly what set the precedence relation is over, and the most we can presume is that it is over the whole set of abstract events. Kunz presents a similar definition, though he does not require a primitive event as an intervening event e. This is potentially quite problematic, so we have chosen Basten's definition. The notion behind this definition is that an abstract event, once all input events have occurred, can complete execution independently of external influence. This was the premise behind contractions, and thus it is a reasonable one, and appeared easier to work with than contractions. With Kunz's definition, it appears that what is a convex event depends on what has already been made into a convex event. It is unclear how efficiently this determination can be made. With Basten's definition this problem does not occur, but convex events can be created that cannot be displayed using current display algorithms. Specifically, convex events can be created that are symmetric in the abstract precedence relation.

The data-structure requirements for event abstraction have largely been formed around the question of what timestamps are required to reflect the relevant precedence relationship. The timestamps developed have in turn been based on Fidge/Mattern timestamps at the primitive-event level.

Many distributed observation systems do not display or otherwise process abstract events. The ATEMPT system incorporates edge contractions, a subset of contractions. Specific unary events may be merged in the display into a single abstract event. This is displayed using a triple event object, to indicate that it is hiding multiple unary events. The POET system incorporates convex abstract events. Figure 3.14 shows a possible event-abstraction view of the binary-merge computation shown in Figures 3.9 and 3.11. The abstract events are rectangles, usually covering multiple traces, with solid coloration on traces where primitive events exist as constituents of the abstract event.

3.2.4.2 TRACE ABSTRACTION

Just as event abstraction is the act of creating abstract events from primitive ones, trace abstraction creates abstract traces from primitive ones. In the most-general definition, an abstract trace is a set of primitive and other abstract traces. It is usually implicitly assumed that an abstract trace cannot contain itself, either directly or through a recursive

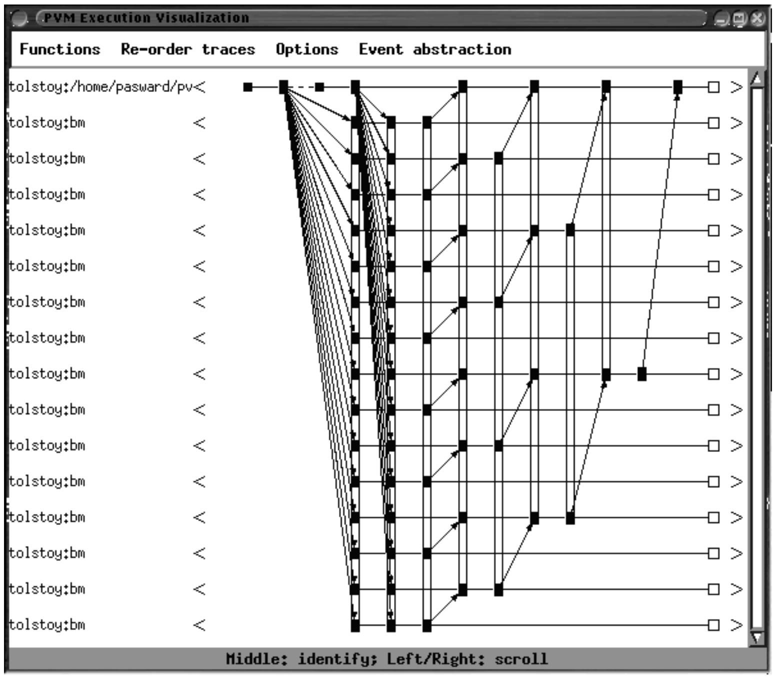


Figure 3.14: POET Event Abstraction

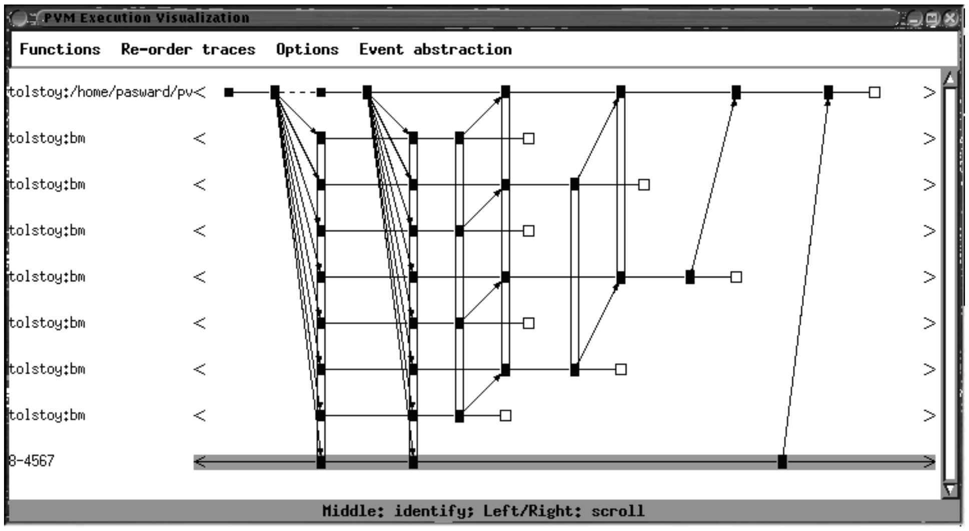
chain. This yields a hierarchy of abstract trace sets, with primitive traces at the base:

$$\mathcal{P}_N \subseteq \begin{cases} 2^{\bigcup_{i=0}^{N-1} \mathcal{P}_i} & N > 0 \\ \{\{p\} \mid p \in \mathcal{P}\} & N = 0 \end{cases} \tag{3.29}$$

As in event abstraction, the primary problem is to deal with the correct representation and manipulation of event precedence. In this regard, Cheung [12] defined a consistent interface cut as follows. Consider a set of traces, which are to be divided into two clusters. Create a virtual interface, the interface cut, between the two clusters, which intersects messages that are passed between the clusters. Every message between the clusters creates two events in the interface cut, a receive from the sending cluster and a transmit to the receiving cluster. Such an interface cut is consistent if the events it is composed of are ordered such that no new precedence relationships are implied by its presence.

There are several problems with this idea of a consistent interface cut. First, it cannot exist in the general case, since Cheung required it to be totally ordered. His solution to this was to require that a cluster interface point have a representative trace through which all communication would flow. This may appear to be an excessive restriction, though it is not difficult to overcome in practice. It is possible to declare there to be multiple interfaces, one for each communicating trace between the two clusters. This approach is effectively equivalent to a single partially-ordered interface, composed of multiple totally-ordered interface cuts. Note that creating a minimal set of such cuts would be NP-hard.

A more serious problem, observed by Taylor [67], is that a consistent interface cut can cause user confusion. Specifically, the events on the interface cut are totally ordered even though the communication that they represent may not be so ordered. This point remains true even with clusters that have a representative trace. As such, POET takes the view that events on the interface should maintain the same partial-order relationship as the communication events that they correspond to. The effect of this is to move from a symmetric view, where the objective is to maintain the same view on both sides of the cluster interface, to an asymmetric one, where only the traces inside the cluster are used to determine the interface. The quality of the resulting abstract traces is then a function of the quality of the clustering. If traces that communicate together frequently are clustered together, and in distinct clusters from those with which



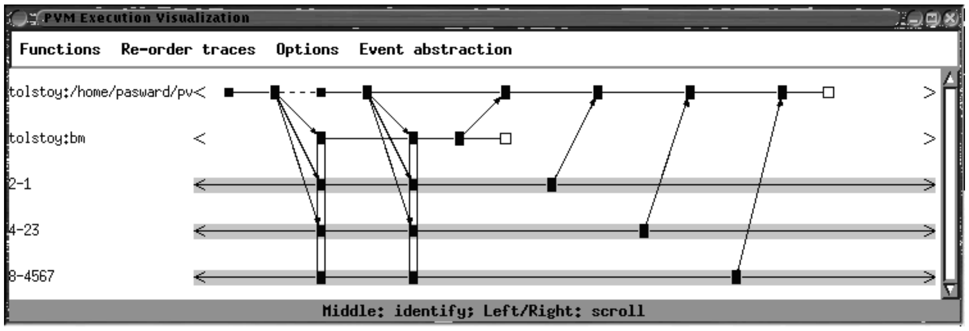


Figure 3.15: POET Trace Abstraction

they communicate infrequently, then the abstract traces should be of good quality. If the clustering is poorer, then it is more likely that the abstract traces will require as many totally-ordered interface cuts as the real traces they represent.

The primary work of data structures for trace abstraction revolves around the problem of precedence-relationship determination in the presence of trace abstraction. Specifically, as with event abstraction, the issue has tended to be posed in terms of what is the appropriate timestamp for an event given that it is in a clustered trace and other traces are also in clusters. This can be decomposed into two distinct problems. The first is the timestamp determination for the events that are in the interface cuts. Cheung provides an algorithm for this in term of Fidge/Mattern timestamps. The second issue is determining if it is possible to reduce the amount of information that needs to be processed by the observation system when using trace abstraction. Both Cheung and Summers give cluster-timestamp algorithms, but both are offline and require an initial calculation of Fidge/Mattern timestamps for all events. Summers provides a rather interesting technique based on an internal view of a cluster, which may have application to scalable timestamp creation.

Displays of abstract traces tend to be similar to displays of real traces, with some simple marker to indicate that it is a cluster, rather than a single trace. The ATEMPT system simply labels the relevant interface cut with the user-selected name. It is able to do this as no effort is made to have a correct interface cut, which in the general case requires more than a single trace line. The POET system uses an alternate background colour for the cuts that compose the interface. An example of the POET trace abstraction method is shown in Figure 3.15. It is the same binary-merge computation as that of Figure 3.14, with the same abstract events. The difference is that in the top illustration the lower eight processes have been abstracted into a single cluster and in the bottom one a hierarchy of trace abstractions has been created.

The figures also have some errors in them, as a result of incorrect interaction between trace and event abstraction. Specifically, abstract events that have constituents both within and without the cluster do not have an event displayed at the cluster interface trace if those events within the cluster are not communication events with partner events outside the cluster. Thus, the third multi-trace abstract event in the upper figure fails to show an event in the interface trace even though that abstract event spans the cluster. The essence of this problem is that trace abstraction is not completely orthogonal to event abstraction. In particular, event abstraction can cover multiple traces, as well as multiple events within a trace. The lower figure has an additional problem that the bottom interface trace abstract events do not have the transmissions to them displayed. This problem may simply be a code defect, since there is no clear abstraction-interaction issue.

4 Scalability Problems

Having described the current state-of-the-art for distributed observation and control, we now wish to look at what breaks as the system under observation gets larger. There are two, non-orthogonal, ways in which distributed systems can grow under our model: an increased number of events and an increased number of traces. In this section we will look at what problems occur as these increase and why they occur. As with the discussion of current systems, we will address the scalability issues in terms of partial-order data structures and visualization.

4.1 The Vector-Clock-Size Problem

As we have seen in Section 3.1.2, current partial-order data-structure techniques revolve around the use of vector timestamps. The actual storage of events tends not be be described, though as the number of events grows, naive storage methods would have to give way to more sophisticated, though not particularly novel, methods such as B-trees. As the number of traces grows, however, the various vector-timestamp techniques can suffer from scaling problems.

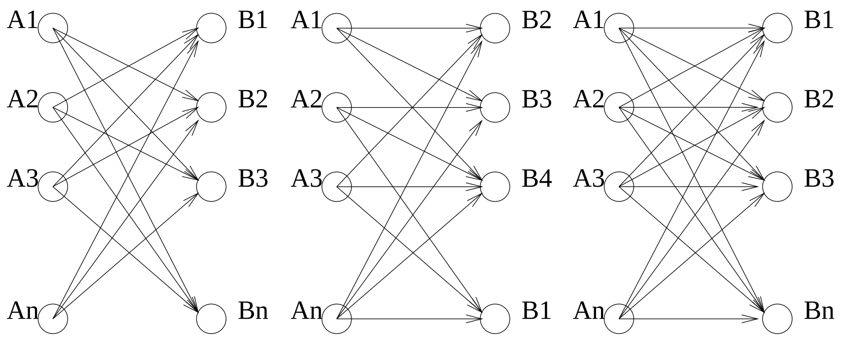
The Fidge/Mattern technique requires a vector of size equal to the number of processes. For a partial-order data structure, this implies such a vector for each element stored in the structure. This is not feasible in space consumption as the number of traces approaches several hundred. The Fowler/Zwaenepoel timestamps potentially scale in terms of space consumption. This is not a clear point, as we are not aware of work that has studied it. Insofar as an increase in the number of traces does not imply that all of these traces communicate, then it will be true. Note, however, that once communication has taken place, the Fowler/Zwaenepoel technique will forever require a vector element representing the source trace of that communication. Jard/Jourdan does not suffer this deficiency, though it is not clear in the first place how Jard/Jourdan can be used in a partial-order data structure, nor how it would scale if it were so used. Note also that the Fowler/Zwaenepoel technique requires a potentially costly search to determine precedence and this search time may increase as the number of traces increases, though again, this is not currently known. The Ore timestamp scales with dimension, not with the number of traces, so it is potentially more scalable.

The justification for vectors of size equal to the number of processes is as follows. To capture precedence in a partial order it is necessary to have a vector (or equivalent) of size equal to the dimension of that partial order [54]. Further, the dimension of a partial order can be as large as the width [75]. Crown $\mathbf{S}_n^0$ is the standard example of such a partial order, and is shown in Figure 4.1(a). By shifting each B_i element of this partial order we create the distributed computation shown in Figure 4.1(b). While this computation does not violate our model of distributed computation, it is unusual in that it requires both multicast transmit operations and corresponding multi-receive operations. While both of these operations do have real systems counterparts, it is important to emphasize that neither is necessary. Charron-Bost [9] translated crown $\mathbf{S}_n^0$ into a point-to-point distributed computation with dimension n. Both crown $\mathbf{S}_n^0$ and the Charron-Bost computation correspond to all processes sending a message to all other processes, with the exception of their left neighbour.

The limitation of the Charron-Bost proof is precisely in the nature of what the crown $\mathbf{S}_n^0$ distributed computation represents. In practical terms, this is not a realistic distributed computation. In addition, the more likely computation, in which each process broadcasts a message to all other processes (shown in Figure 4.1(c)), has dimension 2. We have already demonstrated [81, 82] that, in a significant number of distributed computations, with up to 300 processes, the dimension is typically much smaller than the number of processes. In the cases we examined it was 10 or less.

In addition to Fowler/Zwaenepoel and Jard/Jourdan timestamps, there is one other technique that has been studied for Fidge/Mattern vector-timestamp scalability. This is differential encoding. In the case of a unary or a transmit event, the Fidge/Mattern algorithm changes only a single vector element. It is clearly possible to look at storing just the change, rather than the entire vector.

The primary work in this area is a technique by Singhal and Kshemkalyani [62]. The problem with this work is that it comes from the viewpoint of maintaining a vector clock within a distributed computation. Thus, they were concerned with reducing the size of the information that would have to be transmitted between processes of a distributed



(a) Crown $\mathbf{S}_n^0$ (b) Shifted crown (c) Broadcast

Figure 4.1: Crown and broadcast partial orders

computation that are maintaining vector clocks, and not significantly concerned with the size of data structure that was needed at each process. Their technique requires an $O(n^2)$ matrix at each process to determine what has changed since the last communication and thus what to transmit. This matrix is likewise required at the receiver to recover the Fidge/Mattern vector timestamp. We note in passing that any information-compression technique can be applicable in this context of maintaining vector clocks, but is not generally applicable to the creation of a partial-order data structure. The differential technique may be useful in our context, but clearly not in the manner just described.

4.2 Visualization Limitations

There are two key problems with visualization as the distributed computation scales. The first problem is that, since we wish to display the partial order of execution, we find ourselves limited by the typical 21-inch computer monitor and by human cognition. Such a screen is limited to displaying approximately 50 traces and maybe 75 events per trace. While we could show more events and traces with some perhaps more-compact trace arrangement or a larger screen, the number is ultimately constrained by screen size. Human cognition presents a more fundamental limitation in this regard. Any user of such a visualization tool cannot be expected to extract meaningful information out of a few thousand traces even if they could all be displayed in one screen.

Clearly better visualization techniques may help here. A number of suggestions in this area include drawing the traces on a cylinder, using a fish-eye view, or creating a three-dimensional visualization. For example, Ware and Franck [85] have shown that a user can process three times the amount of information if a display is three-dimensional rather than two-dimensional. Removing extraneous clutter from the display, such as messages whose endpoints are not currently visible, would also help. None of these approaches, however, are going to effectively enable scalability. They are largely visualization tricks to show more information. However, there is simply far too much information for any user to deal with. This leads to the second key problem, which is that some forms of abstraction are needed to provide effective visualization.

Some work has been done in this area of abstraction [12, 39, 40, 41, 42, 43, 67]. However, most of it has focussed on the forms and mechanics of abstraction, and little has been done in the area of presentation. The notable exception is the information mural [33] which is effectively a crude form of abstraction. There are three issues with regard to scalable abstraction: first, the applicability of current abstraction ideas at scale; second, the effective display of such abstractions; and third scalable algorithms for the current abstractions.

At present it is unclear whether or not the current abstraction ideas scale. That is, it is unknown if event and trace abstraction are meaningful as the number of events and traces grows. This issue is only resolvable by creating a scalable observation system and determining empirically if the abstraction ideas work at scale. Further, the abstractions must be derivable automatically, and such derivation techniques would have to be scalable.

The current display of trace abstraction should scale with the number of traces. However, abstract-event displays probably do not scale well. There are two problems associated with them. First, they span all traces in which they have a member event (presuming correct abstraction interaction). As the computation scales, abstract events must grow. If they do not, the visualization will fail to scale. However, as they grow they will likely span more traces. This will result in a display that has a very large number of traces, with a very large number of abstract events covering most of the traces. This is not likely to be meaningful to a user. In addition, there are likely to be more concurrent abstract events whose member trace sets overlap, and thus must be displayed sequentially. Second, as they grow, their duration will likely increase. Thus, the current technique of having only one solid rectangle for a trace where member events occur will become unreflective of the underlying reality.

Finally, as we noted in our discussion of abstraction, the data-structure mechanics of current abstractions is closely tied to Fidge/Mattern timestamps. Since these timestamps do not scale, alternate mechanisms that do scale must be developed. In addition, in the case of trace abstraction, current techniques must examine all events within a cluster to determine what must be displayed at the interface. This is undesirably expensive, and may not scale well as clusters grow.

5 Solution Approaches

Having identified the scalability issues, we see that there are three problems that need to be solved to provide scalable distributed-system observation. First, we require a scalable partial-order data structure. Second, the various abstraction algorithms must be adapted to this data structure. Third, we require new display mechanisms to present the data.

We have gone some way toward demonstrating that a scalable partial-order data structure is possible. Specifically, we have shown that distributed computations frequently have a low dimension relative to their width [80, 81, 82]. We have extended this work to develop a framework algorithm for a dynamic variant of the Ore timestamp [83]. This framework algorithm has allowed us to identify two key problems that need to be resolved for this approach to be successful. The first problem is the incremental creation of the linear extensions. The core issue here does not appear to be the placement of events as they arrive to ensure the reversal of critical pairs. Rather, the tricky problems are the placement of events in extensions that do not need to reverse the critical pairs and the creation of new extensions when it is found to be necessary. Some solutions in this area have been developed, though it is currently unclear how they will work in practice.

The second key problem that the framework algorithm identified was the issue of timestamp distribution. This is not a problem for the creation of a scalable partial-order data structure *per se*, but it is a problem for the development of a distributed partial-order data structure. Specifically, the POET system is structured as a client/server distributed-observation system. Event structures, including timestamp information, are passed between processes. This effectively implements a distributed partial-order data structure. It is not clear at this point how well the online Ore framework algorithm can work in this environment, since it requires timestamp adjustment operations at various times.

In addition to this work on Ore timestamps, we have recently developed an online variant of Summers' cluster timestamp algorithm [84]. It reduces the vector size at the expense of a multi-step precedence test. The appropriate tradeoffs in this approach need to be examined more closely.

Given a scalable partial-order data structure, we plan to adapt the various abstraction algorithms to that data structure. The primary problem in this area is that the various abstraction algorithms are expressed in terms of Fidge/Mattern-like timestamps. That is to say, they presume that the primitive events have Fidge/Mattern-like timestamps, and that the abstract and interface events created should likewise have Fidge/Mattern-like timestamps. The algorithms must therefore be adjusted to be described in terms of the data structure operations. We have not yet seen fundamental dependencies between the algorithms and Fidge/Mattern timestamps, though this remains to be seen. In addition to this, we would like to examine the possibility of more closely integrating abstraction approaches and the data structure, with a view to obtaining improved abstraction performance.

The third scalability issue, new display mechanisms for abstract events, is largely orthogonal to the previous two. We propose to address this by displaying abstract events as single entities, rather than maintaining the current trace/event view. In this technique we envision having an automatic event-abstraction mechanism that is able to create abstract events to the very highest level. That is, ultimately the whole computation would consist of a single abstract event. A user would then be given the capacity to explore any given abstract event by clicking on the event. Conversely, a user could zoom out of some set of events to see the higher-level abstract event. The effectiveness of such an approach is a function of the effectiveness of the automatic event-abstraction mechanism in creating meaningful abstract events.

An alternate approach to this problem is to explore the issue of how to model abstract events. In particular, given that abstract events are not atomic in the sense that primitive ones are, it is reasonable to model them in a more complex manner. We do not expect to take this approach in our work.

There are several other issues we have raised in this report that could be explored, but that we do not anticipate examining. These include the modeling of synchronous events, alternatives to convex events, different visualization techniques, and the use of abstraction to facilitate the creation of a scalable partial-order data structure. We will examine these issues only insofar as it is necessary to achieve our three main scalability objectives.

References

[1] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, Redwood City, California, 1991.

[2] A. A. Basten. Hierarchical event-based behavioural abstraction in interactive distributed debugging: A theoretical approach. Master's thesis, Eindhoven University of Technology, Eindhoven, 1993.

[3] Twan Basten, Thomas Kunz, James P. Black, Michael H. Coffin, and David J. Taylor. Vector time and causality among abstract events in distributed computations. *Distributed Computing*, 11(1):21–39, December 1997.

[4] Pete C. Bates. Debugging heterogenous distributed systems using event-based models of behaviour. *ACM SIGPLAN Notices*, 24(1):11–22, January 1989.

[5] Adam Begulin. Xab: A tool for monitoring PVM programs. Technical Report CMU-CS-93-164, Carnegie Mellon University, School of Computer Science, Carnegie Mellon University, June 1993.

[6] Adam Begulin and Erik Seligman. Causality-preserving timestamps in distributed programs. Technical Report CMU-CS-93-167, Carnegie Mellon University, School of Computer Science, Carnegie Mellon University, June 1993.

[7] Eike Best and Brian Randell. A formal model of atomicity in asynchronous systems. *Acta Informatica*, 16:93–124, 1981.

[8] T.L. Casavant and M. Singhal, editors. *Readings in Distributed Computing Systems*. IEEE Computer Society Press, Los Alamitos, California, 1994.

[9] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Information Processing Letters*, 39:11–16, July 1991.

[10] B. Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, asynchronous and causally ordered communication. Submitted to Distributed Computing.

[11] Craig M. Chase and Vijay K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11:191–201, 1998.

[12] Wing Hong Cheung. *Process and Event Abstraction for Debugging Distributed Programs*. PhD thesis, University of Waterloo, Waterloo, Ontario, 1989.

[13] Janice Cuny, George Forman, Alfred Hough, Joydip Kundu, Calvin Lin, Lawrence Snyder, and David Stemple. The ariadne debugger: Scalable application of event-based abstraction. *ACM SIGPLAN Notices*, 28(12):85–95, May 1993.

[14] Suresh K. Damodaran-Kamal and Joan M. Francioni. Nondeterminancy: Testing and debugging in message passing parallel programs. *ACM SIGPLAN Notices*, 28(12):118–128, May 1993.

[15] Giuseppe Di Battista, Peter Eadea, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: An annotated bibliography. *Computational Geometry: Theory and Applications*, 4(5):235–282, June 1994.

[16] Edsger Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, August 1975.

[17] Greg Eisenhauer, Weiming Gu, Eileen Kraemer, Karsten Schwan, and John Stasko. Online displays of parallel programs: Problems and solutions. In *Proceedings of the International Conference on Parallel and Distributed Prcessing Techniques and Applications*, pages 11–20. PDPTA'97, 1997.

[18] Colin Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.

[19] Colin Fidge. Fundamentals of distributed systems observation. Technical Report 93-15, Software Verification Research Centre, Department of Computer Science, The University of Queensland, St. Lucia, QLD 4072, Australia, November 1993.

[20] Jerry Fowler and Willy Zwaenepoel. Causal distributed breakpoints. In *Proceedings of the 10th IEEE International Conference on Distributed Computing Systems*, pages 134–141. IEEE Computer Society Press, 1990.

[21] Jason Gait. A probe effect in concurrent programs. *Software — Practice and Experience*, 16(3):225–233, March 1986.

[22] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, 1979.

[23] Siegfried Grabner, Deiter Kransmueller, and Jens Volkert. EMU — Event Monitoring Utility. Technical report, Institute for Computer Science, Johannes Kepler University Linz, July 1994.

[24] Weiming Gu, Greg Eisenhauer, Karsten Schwan, and Jeffrey Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 6(2), 1998.

[25] Jessica Zhi Han. Automatic comparison of execution histories in the debugging of distributed applications. Master's thesis, University of Waterloo, Waterloo, Ontario, 1998.

[26] Paul K. Harter, Dennis M. Heimbigner, and Roger King. IDD: An interactive distributed debugger. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 498–506, May 1985.

[27] M. T. Heath and J. A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, pages 29–39, September 1991.

[28] Alfred A. Hough and Janice E. Cuny. Belvedere: Prototype of a pattern-oriented debugger for highly parallel computation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 735–738, 1987.

[29] IBM Corporation. WebSphere application server, object level trace. Technical Report http://www-4.ibm.com/software/webservers/appserv/olt.html, IBM Corporation, 1998.

[30] Intel Corporation. System performance visualization tool user's guide. Technical Report 312889-001, Intel Corporation, 1993.

[31] Claude Jard and Guy-Vincent Jourdan. Dependency tracking and filtering in distributed computations. Technical Report 851, IRISA, Campus de Beaulieu – 35042 Rennes Cedex – France, August 1994.

[32] Dean Jerding, John T. Stasko, and Thomas Ball. Visualizing interactions in program executions. In *International Conference on Software Engineering*, pages 360–370. IEEE, 1997.

[33] Dean F. Jerding and John T. Stasko. The information mural: A technique for displaying and navigating lare information spaces. Technical Report GIT-GVU-97-24, Georgia Institute of Technology, Atlanta, GA, 1997.

[34] James Arthur Kohl and G. A. Geist. XPVM 1.0 user's guide. Technical Report TM-12981, Oak Ridge National Laboratory, Oak Ridge, Tennessee 37831, November 1996.

[35] Eileen Kraemer. Causality filters: A tool for the online visualization and steering of parallel and distributed programs. In *Proceedings of the 11th IPPS*. IEEE, 1997.

[36] Eileen Kraemer and John T. Stasko. Creating an accurate portrayal of concurrent executions. *Concurrency*, 6(1):36–46, 1998.

[37] Deiter Kransmueller, Siegfried Grabner, R. Schall, and Jens Volkert. ATEMPT — A Tool for Event ManiPulaTion. Technical report, Institute for Computer Science, Johannes Kepler University Linz, May 1995.

[38] Deiter Kransmueller, Siegfried Grabner, and Jens Volkert. PARASIT — Parallel simulation tool. Technical report, Institute for Computer Science, Johannes Kepler University Linz, December 1994.

[39] Thomas Kunz. *Abstract Behaviour of Distributed Executions with Applications to Visualization*. PhD thesis, Technische Hochschule Darmstadt, Darmstadt, Germany, 1994.

[40] Thomas Kunz. Visualizing abstract events. In *Proceedings of the 1994 CAS Conference*, pages 334–343, October 1994.

[41] Thomas Kunz. Automatic support for understanding complex behaviour. In *Proceedings of the International Workshop on Network and Systems Management*, pages 125–132, August 1995.

[42] Thomas Kunz. High-level views of distributed executions. In *Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging*, pages 505–512, May 1995.

[43] Thomas Kunz. Evaluating process clusters to support automatic program understanding. In *Proceedings of the 4th Workshop on Program Comprehension*, pages 198–207, March 1996.

[44] Thomas Kunz, James P. Black, David J. Taylor, and Twan Basten. POET: Target-system independent visualisations of complex distributed-application executions. *The Computer Journal*, 40(8):499–512, 1997.

[45] L. Lamport and N. Lynch. Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science*, volume 2, pages 1157–1199. Elsevier Science Publishers B. V., 1990.

[46] Leslie Lamport. Time, clocks and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, 1978.

[47] Thomas J. LeBlanc, John M. Mellor-Crummey, and Robert J. Fowler. Analyzing parallel program executions using multiple views. *Journal of Parallel and Distributed Computing*, 9:203–217, 1990.

[48] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et al., editor, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Chateau de Bonas, France, December 1988. Elsevier Science Publishers B. V. (North Holland).

[49] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4), December 1989.

[50] Barton P. Miller and Jong-Deok Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th IEEE International Conference on Distributed Computing Systems*, pages 316–323. IEEE Computer Society Press, 1988.

[51] Sape Mullender, editor. *Distributed Systems*. Addison-Wesley, New York, 2 edition, 1993.

[52] Oleg Y. Nickolayev, Philip C. Roth, and Daniel A. Reed. Real-time statistical clustering for event trace reduction. *Journal of Supercomputing Applications and High-Performance Computing*, 11(2):144–159, 1997.

[53] E. R. Olderog. Operational petri net semantics for CCSP. *IEEE Network*, pages 34–43, September/October 1997.

[54] Oystein Ore. *Theory of Graphs*, volume 38. Amer. Math. Soc. Colloq. Publ., Providence, R.I., 1962.

[55] Cherri M. Pancake. Applying human factors to the design of performance tools. In P. Amestoy, P. Berger, M. Daydé, I. Duff, V. Frayssé, L. Giraud, and D. Ruiz, editors, *EuroPar'99 Parallel Processing*, Lecture Notes in Computer Science, No. 1685, pages 44–60. Springer-Verlag, 1999.

[56] Guru Parulkar, Douglas Schmidt, Eileen Kraemer, Jonathan Turner, and Anshul Kantawala. An architecture for monitoring, visualization, and control of gigabit networks. *IEEE Network*, pages 34–43, September/October 1997.

[57] Michel Raynal and Mukesh Singhal. Capturing causality in distributed systems. *IEEE Computer*, 29(2):49–56, 1996.

[58] L. E. T. Rodrigues and Paulo Verissimo. Cauasal separators for large-scale multicast communication. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, pages 83–91, Vancouver, June 1995. IEEE Computer Society Press.

[59] Reinhard Schwarz and Friedemann Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[60] Ehud Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3), September 1989.

[61] Joseph L. Sharnowski and Betty H. C. Cheng. A visualization-based environment for top-down debugging of parallel programs. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 640–645. IEEE Computer Society Press, 1995.

[62] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43:47–52, August 1992.

[63] Janice M. Stone. A graphical representation of concurrent processes. *ACM SIGPLAN Notices*, 24(1):226–235, January 1989.

[64] James Alexander Summers. Precedence-preserving abstraction for distributed debugging. Master's thesis, University of Waterloo, Waterloo, Ontario, 1992.

[65] David J. Taylor. Scrolling displays of partially ordered execution histories. In preparation.

[66] David J. Taylor. A prototype debugger for hermes. In *Proceedings of the 1992 CAS Conference*, volume 1, pages 29–42, November 1992.

[67] David J. Taylor. The use of process clustering in distributed-system event displays. In *Proceedings of the 1993 CAS Conference*, pages 505–512, November 1993.

[68] David J. Taylor. Integrating real-time and partial-order information in event-data displays. In *Proceedings of the 1994 CAS Conference*, pages 505–512, November 1994.

[69] David J. Taylor. Event displays for debugging and managing distributed systems. In *Proceedings of the International Workshop on Network and Systems Management*, pages 112–124, August 1995.

[70] David J. Taylor, Thomas Kunz, and James P. Black. Achieving target-system independence in event visualisation. In *Proceedings of the 1995 CAS Conference*, pages 296–307, November 1995.

[71] Brad Topol, John T. Stasko, and Vaidy Sunderam. Integrating visualization support into distributed computing systems. Technical Report GIT-GVU-94-38, Georgia Institute of Technology, Atlanta, GA, October 1994.

[72] Brad Topol, John T. Stasko, and Vaidy S. Sunderam. Dual timestamping methodology for visualizing distributed application behaviour. *International Journal of Parallel and Distributed Systems and Networks*, 1(2):43–50, 1998.

[73] Brad Topol, John T. Stasko, and Vaidy S. Sunderam. PVaniM: A tool for visualization in network computing environments. *Concurrency: Practice and Experience*, 10(14):1197–1222, 1998.

[74] William T. Trotter. Graphs and partially-ordered sets. In R. Wilson and L. Beineke, editors, *Selected Topics in Graph Theory II*, pages 237–268. Academic Press, 1983.

[75] William T. Trotter. *Combinatorics and Partially Ordered Sets: Dimension Theory*. Johns Hopkins University Press, Baltimore, MD, 1992.

[76] William T. Trotter. Partially ordered sets. In R. Graham, M. Grötschel, and L. Lovász, editors, *Handbokk of Combinatorics*, pages 433–480. Elsevier Science, 1995.

[77] G. J. W. van Dijk and A. J. van der Wal. Partial ordering of synchronization events for distributed debugging in tightly-coupled multiprocessor systems. In *Proceedings of the 2nd European Distributed Memory Computing Conference*, pages 100–109. Springer Verlag, 1991.

[78] J.S. Vetter. Computational steering annotated bibliography. *ACM SIGPLAN Notices*, 32(6):40–44, June 1997.

[79] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. Technical Report SMLI TR-94-29, Sun Microsystems Laboratories, Inc., November 1994.

[80] Paul Ward. On the scalability of distributed debugging: Vector clock size. Technical Report CS98-29, Shoshin Distributed Systems Group, Department of Computer Science, The University of Waterloo, Waterloo, Ontario, Canada N2L 3G1, December 1998. Available at ftp://cs-archive.uwaterloo.ca/cs-archive/CS-98-29/CS-98-29.ps.Z.

[81] Paul A.S. Ward. An offline algorithm for dimension-bound analysis. In Dhabaleswar Panda and Norio Shiratori, editors, *Proceedings of the 1999 International Conference on Parallel Processing*, pages 128–136. IEEE Computer Society, 1999.

[82] Paul A.S. Ward. An online algorithm for dimension-bound analysis. In P. Amestoy, P. Berger, M. Daydé, I. Duff, V. Frayssé, L. Giraud, and D. Ruiz, editors, *EuroPar'99 Parallel Processing*, Lecture Notes in Computer Science, No. 1685, pages 144–153. Springer-Verlag, 1999.

[83] Paul A.S. Ward. A framework algorithm for dynamic, centralized dimension-bounded timestamps. In *Proceedings of the 2000 CAS Conference*, November 2000.

[84] Paul A.S. Ward and David J. Taylor. A hierarchical cluster algorithm for dynamic, centralized timestamps. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems*. IEEE Computer Society Press, 2001.

[85] Colin Ware and Glenn Franck. Evaluating stereo and motion cues for visualizing information nets in three dimensions. *ACM Transactions on Graphics*, 15(2), 1996.

[86] Roland Wismüller, Jöse Trinitis, and Thomas Ludwig. OCM — A monitoring system for interoperable tools. In *SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 1–9, New York, 1998. ACM.

[87] Yuh Ming Yong. Replay and distributed breakpoints in an OSF DCE environment. Master's thesis, University of Waterloo, Waterloo, Ontario, 1995.

[88] Yuh Ming Yong and David J. Taylor. Performing replay in an OSF DCE environment. In *Proceedings of the 1995 CAS Conference*, pages 52–62, November 1995.