Event-Predicate Detection in the Debugging of Distributed Applications

by

Christian Eugene Jaekl

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 1996

Abstract

Trends in the development of computer hardware are making the use of distributed systems increasingly attractive. The collection of event-trace data and the construction of process-time diagrams can provide a useful visualization tool. In practical situations, however, these diagrams are too large for users to find them comprehensible. The ability to detect and locate arbitrary (complex) predicates within an event trace can help to alleviate this problem.

This thesis enumerates five classes of problems that a successful event- detection strategy should be able to identify: phase transitions, mutual- exclusion violations, subroutines, communication symmetry, and performance bottlenecks. Some previous efforts in this area offer an expressivity which is close to that required to meet these goals, but are hampered by an insufficient understanding of the partial order which underlies causality in a distributed-execution trace. This work defines a partial-order precedence relationship for compound events, and extends two timestamping algorithms to support it.

A new syntax for event-predicate definition, which comes closer to fulfilling the aforementioned framework than any of the previous efforts, is presented. Finally, a prototypical implementation, within Taylor's Partial-Order Event Tracer (POET), is described, issues encountered during its construction are discussed, and its performance is evaluated.

Acknowledgements

First and foremost, I would like to thank my supervisor, David Taylor, a charming, witty and ever-helpful guide to the academic experience—David, it has been a pleasure. I also owe a significant debt of gratitude to my readers, Peter Buhr and Thomas Kunz, for their careful attention to detail and thoughtful suggestions. Further thanks are due to James Black, Michael Nidd, Raoul Medina, and to many students at the University of Waterloo, for their input, suggestions, and general assistance; this work has been aided greatly by the atmosphere which all of these people have helped to foster.

I would like to thank my parents; without their prodding, I might never have embarked on this experience in the first place, and I would have missed out on a great deal.

Finally, I gratefully acknowledge the financial support of International Business Machines Corporation, the Information Technology Research Centre, and the Department of Computer Science at the University of Waterloo.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Case for Distributed Systems

"It is not to be expected that the necessary [parallel] programming techniques will be worked out overnight. Much experimenting remains to be done. After all, the techniques that are commonly used in programming today were only won at the cost of considerable toil several years ago. In fact the advent of parallel programming may do something to revive the pioneering spirit in programming, which seems at the present to be degenerating into a rather dull and routine occupation."

—S. Gill, 1958 [18]

The concept of parallel computation is not a new one. To give but two examples, the "Pilot ACE" (1953) had the ability to run multiplication and division operations independently of the control unit [42], and the CDC 6600 (1964) could execute as many as ten instructions in parallel [36, page 20]. Indeed, all modern operating systems support some form of concurrency, often going to great lengths to simulate parallelism on a uniprocessor.

Why, then, is programming still viewed as a fundamentally sequential operation? This is a product of the mind-set into which the available languages have trained programmers [2], and (perhaps more importantly) the fact that, until recently, there were significantly more users than processors. A quarter century ago, Grosch observed that the computational power of a processor varied as the square of its price; under these conditions, it was best to invest in the largest single processor one could afford [37, page 3]. In the past decade, the limited speed of electric current flow, and the difficulties inherent in heat dissipation, have slowed the advance of superprocessors; at the same time, economies of scale have drastically reduced the price of integrated circuits. “Grosch's law” has been turned on its head, and future price/performance gains will require substantial parallelism.

1.2 Development Difficulties

Nevertheless, the great wave of distributed applications, which has been predicted to be imminent for four decades, remains conspicuously absent. Why is this so?

It is inherently more difficult to manage a team of workers, orchestrating their set of actions to accomplish a unified goal, than it is to give directions to a single person; likewise, concurrent applications will always be more difficult to conceptualize, implement, and debug, than their sequential counterparts. With the aforementioned economic impetus, a number of techniques have been developed to assist the *construction* of concurrent programs.[1] At this point, however, a corresponding degree of improvement has not been seen in the techniques and tools for *debugging* concurrent applications once they have been constructed.

When debugging a sequential application, the technique of choice is to repeatedly run the program, within a source-level debugger, under conditions which reproduce the error; by setting breakpoints, reviewing trace output, stepping through the code and examining the contents of variables, the focus of the search is iteratively narrowed until the source of the error is located. Recent work addresses a number of the difficulties inherent in extending this process to a distributed situation:

- **No global clock.** Applications may be spread across several processors, each of which has its own local clock that cannot maintain perfect synchronization with the other clocks in the system; thus, there is no available global time to reveal the global order of execution. This issue can be addressed by the use of logical clocks to maintain a partial order, and is discussed in Chapter 2.

- **Trace-output format.** Trace information is difficult to convey. Combining all the output into a sequential listing yields a linearization that conveys the impression of a total ordering to which the actual execution was not constrained; separating traces on a per-process basis fails to convey information about the partial order which is present. The best solution proposed thus far involves the construction of **process-time diagrams**; Figure 2.2, on page 10, provides an example. Parallel lines indicate the sequential execution local to each trace; symbols on the lines (circles in this example) indicate events within that process; arrows indicate communication, with the arrow-

[1]These include, but are by no means limited to, remote procedure calls, conditional critical regions, and monitors.

head pointing at the receive event within the pair. One can easily ascertain the causality relationship between any two events by examining such a diagram. If there is a directional path between the two events, then the origin happened before the destination; otherwise, there is no causal relationship. The Partial-Order Event Tracer (POET), developed by the Shoshin research group at the University of Waterloo, can automatically construct such a diagram (see Section 4.1).

- **Non-deterministic execution.** The relative speed of execution of separate processes is unpredictable, and the application is likely to be faced with resource contention from other programs on the system. Thus, separate runs on the same input do not necessarily produce the same output; this nondeterminism can present a novel problem when attempting to iteratively "zero in" on a bug. The best available solution is to first capture an execution trace that contains the error condition, and then replay that execution with the aid of a **replay facilitator**, which constrains the application to maintain the observed partial order on inter-process communication. Yong [43] provides a replay mechanism that is both target-independent and portable.

- **Implementation details.** Khouzam [23] extends this replay facility to provide a distributed analogue of single-stepping, and Yu [45] demonstrates how a traditional (sequential) source-level debugger can be connected to a process within a distributed system, which is halted at a breakpoint, allowing the user to examine data structures.

This thesis attempts to address three open problems in the extension of the sequential-debugging paradigm to distributed applications.

- **Searching through large traces.** For non-trivial applications, the event trace generates a process-time diagram which is so huge that it is extremely difficult to peruse and to locate relevant sections of interest in the course of debugging. Some sort of *search* feature is needed; it should be roughly analogous to that provided within a text browser, but adapted to handle the partial order inherent in such a trace.

- **Conditional breakpoints.** A sequential debugger normally provides a directive to break execution if a given condition (e.g., $x = 7$) is satisfied; this grants the user an extra degree of specificity, allowing rapid examination of a certain section of the program in some specific intermediate state.

- **Global assertions.** If an application enters a state that has not been foreseen by the programmer, it may be preferable for it to halt with an error message rather than to continue and to produce behaviour that is likely to be incorrect. Developers often install this type of safeguard by using assertions; in a distributed application, such an assertion may involve a condition spread across several processes.

The astute reader should realize that these are three variations on a common theme. What is needed is a mechanism to define and detect patterns of execution within a distributed system.

1.3 Thesis Overview

This thesis describes a strategy for the definition and detection of predicates which can meet the aforementioned goals. Chapter 2 discusses the logical time relating primitive events in distributed execution traces and the vector timestamps which are used to describe it; these timestamps (and their associated algorithms) are extended to deal with compound events, a prerequisite for any sound predicate-detection strategy. Chapter 3 reviews previous developments in predicate-detection, assesses their strengths and weaknesses, and presents a new predicate-definition language. Chapter 4 describes a prototypical implementation of this language, and discusses the more interesting difficulties which were encountered in its construction; an example post-mortem debugging session illustrates the nature of the functionality which it currently provides. Chapter 5 discusses the factors which limit the performance of the current implementation, and sketches avenues which might improve the speed of predicate-recognition. Finally, the sixth chapter summarizes the contributions of the thesis and outlines possible lines for future exploration.

Chapter 2

Causality and Compound Events

2.1 The Problem

In order to make sense of trace information, it is essential that the temporal relationships among the events in the processes be observable. The obvious solution, stamping each event with the actual (scalar) global time, presents a unique difficulty in a distributed system. Different processes[1] may not have access to the same clock, and different clocks are unlikely to remain in precise agreement on the current time.

Example 1 (Clock Drift)

Consider two tasks running on two separate processors, each of which has its own local clock (c_1 and c_2 respectively); even if c_1 and c_2 are perfectly

[1]Here, the term *process* designates a single line on a process-time diagram. This may, in fact, be a thread, process, task, semaphore, monitor, or TCP stream, depending on the terminology used by the target system and the granularity of the communication data being collected.

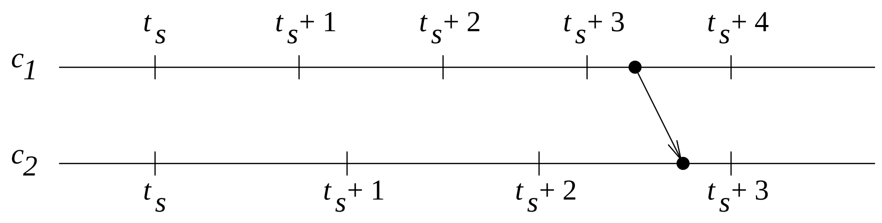


Figure 2.1: Clock-drift example

synchronized at some time t_s, any difference in their rates guarantees that the two clocks will drift apart significantly after that point. Eventually, a message sent from the first process to the second appears to arrive before it departs (see Figure 2.1).

Clock-synchronization algorithms can mitigate this problem. However, since all such algorithms are limited by inter-processor-communication delays [27], they cannot achieve sufficient accuracy for trace-information analysis. In fact, a total ordering of events is neither required nor desired. What is really desired here is a mechanism to describe the causal structure of a distributed execution.

2.2 Defining a Partial Order

Lamport [26] recognized that it is sometimes impossible to determine a strict temporal ordering for two events in a distributed system; thus, event traces necessarily form a partial order. Lamport defined the **happened before** relation as follows:

Definition 1 (Sequential composition)

Let a, b and c be events. $a \rightarrow c$ (a "happened before" c) iff

- *a and c are events in the same process and a comes before c, or*
- *a is the transmission of a message, and c is the receipt of that transmission, or*
- *$a \rightarrow b$ and $b \rightarrow c$*

This is often referred to as the **causality relationship** because $a \rightarrow b$ implies that it is possible for a to causally affect b. If two events are not causally related, then they are considered to be **concurrent**.

Definition 2 (Concurrent composition)

Let a and b be events. $a \parallel b$ (a and b are "concurrent") iff $\neg(a \rightarrow b)$ and $\neg(b \rightarrow a)$.

Example 2

In the process-time diagram shown in Figure 2.2, intra-process time flows from left to right; i.e., $a \rightarrow b \rightarrow c \rightarrow d$ and $e \rightarrow f \rightarrow g \rightarrow h$. Also, since f is a send event, and c is the corresponding receive, $f \rightarrow c$ and, applying the transitivity of the happened-before relation, $e \rightarrow f \rightarrow c \rightarrow d$. Finally, because of the absence of any other inter-process communication, $a \parallel e$, $a \parallel f$, $a \parallel g$, $a \parallel h$, $b \parallel e$, $b \parallel f$, $b \parallel g$, $b \parallel h$, $c \parallel g$, $c \parallel h$, $d \parallel g$ and $d \parallel h$.

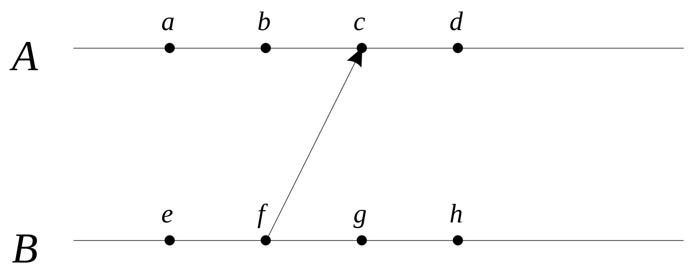


Figure 2.2: Process-time diagram discussed in Example 2

2.3 Timestamping (Exclusively Asynchronous Communication)

2.3.1 Fidge/Mattern Timestamps

Fidge [14, 15, 16] and Mattern [28] separately developed the notion of vector timestamps, which provide a means of determining the causality relationship between two events without consulting the communication graph. In a system with n processes, each process p maintains a local vector clock C_p of size n; the p^{th} component of C_p "ticks" each time an event occurs on process p, and the other components maintain information about the latest "known" clock values on other processes.

Definition 3 ($proc()$)

> *If a is a primitive event, then $proc(a)$, abbreviated p_a, is the (index of the) process in which a occurs.*

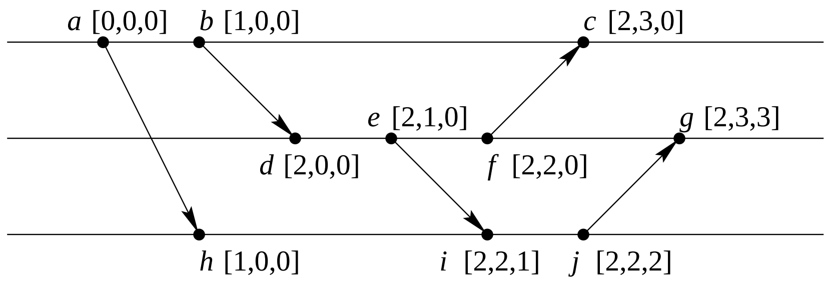


Figure 2.3: (Exclusively asynchronous) communication and Fidge timestamps

Algorithm

Initialize each local clock C_p (recall that p is the ordinal identifier for the current process) to the zero vector, except for the p^{th} element, which is assigned the value (-1).

When an event a is encountered on process p_a, $C_{p_a}[p_a]$ is incremented. Then,

- if a is a *send*, a copy of C_{p_a} is appended to the message being sent.
- if a is a *receive* from process p_b, the received clock value C_{p_b} is used to calculate the new value of C_{p_a} as follows:

$$\begin{aligned} C_{p_b}[p_b] &\mathrel{+}= 1; \\ \forall i \in \{1,\ldots,n\},\ C_{p_a}[i] &:= \max(C_{p_a}[i], C_{p_b}[i]); \end{aligned}$$

Once this calculation is complete, the current value of C_{p_a} is the **timestamp** T_a for event a (see Figure 2.3).

Timestamp Test

Given two timestamps T_a and T_b, the causal relationship between a and b can be determined as follows:

Theorem 1 (Fidge/Mattern timestamp test)

> *Let a and b be events which occur in (not necessarily distinct) processes p_a and p_b. Then $a \rightarrow b$ iff $T_a[p_a] < T_b[p_a]$.*

2.3.2 Summers Timestamps

Motivation

Ore [30] provides a method of assigning timestamp-like "coordinates" to the elements of a partially ordered set; this produces timestamps that can be used to determine whether $a \rightarrow b$ without any knowledge of the processes (p_a, p_b) to which the events (a, b) belong. Inspired by Ore's work, Summers [35] proposed an extension to Fidge's timestamp algorithm which supports both process-based (Fidge-like) and process-independent (Ore-like) timestamp tests.[2]

Algorithm

Each local clock is initialized with the vector $[0, 0, \ldots, 0]$.

When each event a is encountered, $C_{p_a} += 2$ if the previous event in p_a was a *send* event, or $C_{p_a} += 1$ otherwise. Then,

- If a is a *send* event, a copy of C_{p_a} is appended to the message being sent.

[2]Summers' algorithm offers the additional property $a \rightarrow a$ for any event a; this is, however, of no practical concern in a debugging context.

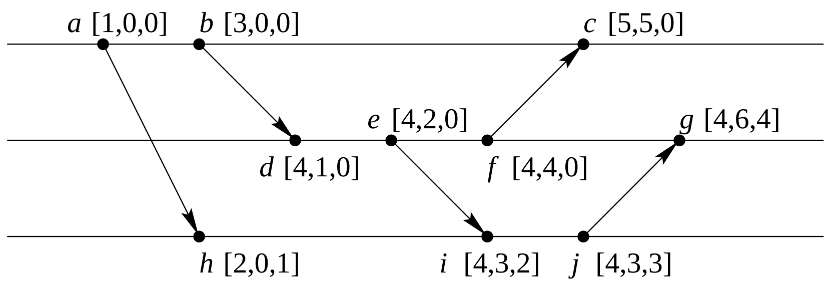


Figure 2.4: (Exclusively asynchronous) communication and Summers timestamps

- If a is a *receive* from process p_b, the received value of clock C_{p_b} (which had been appended to the message) is used to calculate the new value of C_{p_a} as follows:

$$\begin{aligned} C_{p_b}[p_b] \quad &+= \quad 1; \\ \forall i \in \{1,\ldots,n\},\ C_{p_a}[i] \quad &:= \quad \max(C_{p_a}[i], C_{p_b}[i]); \end{aligned}$$

Once this calculation is complete, the current value of C_{p_a} is the **timestamp** T_a for event a (see Figure 2.4).

Timestamp Tests

Definition 4 (Vector $\leq$)

As a notational convenience, define the symbol $\overset{vec}{\leq}$ *as follows: given two vectors A and B, both of size n,* $A \overset{vec}{\leq} B$ *iff* $\forall i \in \{1,\ldots,n\} : A[i] \leq B[i]$.

Let a and b be events which occur in (not necessarily distinct) processes p_a and p_b.

Theorem 2 (Summers process-independent timestamp test)

For any two events, $a \rightarrow b$ *iff* $T_a \overset{vec}{\leq} T_b$

Theorem 3 (Summers process-based timestamp test)

$a \rightarrow b$ *iff* $T_a[p_a] \leq T_b[p_a]$

These timestamp tests are introduced, and proven correct, in Section 2.7 of [35]. Obviously, the second test (which requires only a single comparison) is faster than the first one; however, the vector-comparison test has the advantage that it does not require knowledge about which events happened in which processes.

2.4 Extensions for Synchronous Communication

"Synchronous" communication between two processes in a distributed system is a somewhat confusing concept. In theory, both the send (say, a) and the receive (say, b) occur simultaneously; in practice, there is a slight (but non-zero) delay between the initiation of the transmission at a and the beginning of the reception at b. For the purposes of event-trace-diagram construction and analysis, it is desirable to view both a and b as the same event, with the same timestamp, which just happens to occur in more than one process.

The aforementioned timestamping algorithms do not make provisions for synchronous communication, but they can be extended to do so easily.

2.4.1 Fidge Synchronous-Communication Timestamps

Cheung [9] proposed the following adjustment to the Fidge timestamping algorithm. Alongside the rules for dealing with (asynchronous) *send* and *receive* events, this new rule should be inserted:

- If a is a *synchronous-communication* event with a partner b, then each trace sends the other a copy of its current local clock. Each trace updates its clock value to the vector maximum of both clocks:

  $$\forall i \in \{1, \ldots, n\} : C_{p_a}[i] := C_{p_b}[i] := \max(C_{p_a}[i], C_{p_b}[i]) \ ;$$

  Then, each event takes that clock value (instead of the final value, at the end of processing for this event) as its timestamp:

  $$T_a := T_b := C_{p_a} \ ;$$

  Finally, both processes increment the component of their clock corresponding to their partner's trace by one:

  $$\begin{aligned} C_{p_a}[p_b] &+= 1 \ ; \\ C_{p_b}[p_a] &+= 1 \ ; \end{aligned}$$

An example process-time diagram with timestamps generated by this algorithm is presented in Figure 2.5.

2.4.2 Summers Synchronous-Communication Timestamps

As Summers points out [35], his timestamping algorithm can also be adjusted to accommodate synchronous communication. This adjustment is accomplished by stressing that the rules which deal with *send* and *receive* events in the aforementioned algorithm are to be applied only to *asynchronous* communication events, and by adding a new rule to deal with *synchronous send* and *receive* events. The complete algorithm is given below, with the additions for synchronous communication set in **boldface** type.

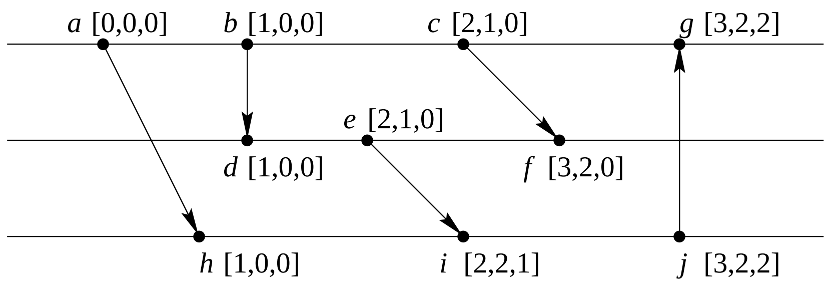


Figure 2.5: Fidge timestamps with synchronous communication

Algorithm

Each local clock is initialized with the vector $[0, 0, \ldots, 0]$.

When each event a is encountered, $C_{p_a} += 2$ if the previous event in p_a was an ***asynchronous*** *send* event, or $C_{p_a} += 1$ otherwise. Then,

- If a is an ***asynchronous*** *send* event, a copy of C_{p_a} is appended to the message being sent, and $T_a := C_{p_a}$.

- If a is an ***asynchronous*** *receive* from process p_b, the received value of clock C_{p_b} (which had been appended to the message) is used to calculate the new value of C_{p_a} as follows:

$$\begin{aligned} C_{p_b}[p_b] \quad &+= \quad 1; \\ \forall i \in \{1, \ldots, n\},\ C_{p_a}[i] \quad &:= \quad \max(C_{p_a}[i], C_{p_b}[i]); \end{aligned}$$

  Then $T_a := C_{p_a}$.

- **If a is a *synchronous-communication* event with a partner b, then each trace sends the other a copy of its current local clock. Each trace updates its clock value to the vector maximum of both clocks:**

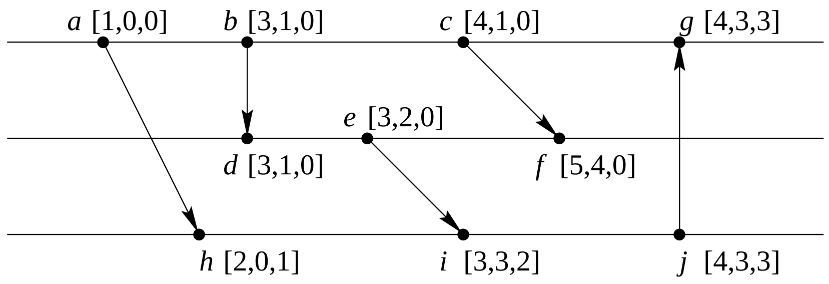


Figure 2.6: Summers timestamps with synchronous communication

$$\forall i \in \{1, \ldots, n\} : C_{p_a}[i] := C_{p_b}[i] := \max(C_{p_a}[i], C_{p_b}[i]) \ ;$$

Then, each event takes that clock value as its timestamp:

$$T_a := T_b := C_{p_a} \ ;$$

- Otherwise (if the event does not involve communication), $T_a := C_{p_a}$.

2.5 The Meaning of Precedence for Compound Events

Often, in the course of analyzing the trace output from a distributed execution, it becomes desirable to group primitive (i.e., single, observed) events into larger conceptual units. To reduce confusion, it is conventional to denote primitive events with lower-case letters, and composite events with capitals.

Once higher-level events have been introduced, it is useful to define a precedence (or causality) relation between them. As Kunz [24] observes, there are two obvious (and conflicting) notions of precedence among such events:

1. $A \leq B$ iff $\forall a \in A : \forall b \in B : a \rightarrow b$.
2. $A \preceq B$ iff $\exists a \in A : \exists b \in B : a \rightarrow b$.

Previous work at Waterloo [4, 5, 24] has focussed on abstracting away levels of detail, and thus reducing the complexity of the trace output to make it comprehensible for a human observer, by grouping successive levels of detail into **abstract events**. Toward that end, the latter definition is probably preferable, since it indicates a precedence between two abstract events even when they are only partially related, thus yielding a richer display. However, this relation is neither anti-symmetric ($A \preceq B \not\Rightarrow \neg(B \preceq A)$) nor transitive ($A \preceq B \wedge B \preceq C \not\Rightarrow A \preceq C$), and so it does not define a partial order.

The first definition avoids these difficulties. It also possesses the desirable property that if $A \leq B$, then all components of A are guaranteed to have completed execution before the first component of B begins; thus, $A \leq B$ guarantees that no race condition exists between A and B.

Finally, on a more pragmatic note, the first definition poses less of an implementation challenge than the second; while both require two timestamps to correctly determine the precedence between arbitrary compound events, the latter requires either a timestamp in reverse-vector time (which makes the on-line monitoring of applications computationally infeasible) [4] or a substantial amount of additional computation involving forward timestamps [5]. Kunz *et al.* [5, 24] circumvent this difficulty by restricting the composition of their "abstract" events to **convex** sets, but this is too much of a limitation to impose on event-predicate definitions.[3]

[3]As will be discussed below, a literal interpretation of the second precedence definition (i.e., an event-by-event comparison) is feasible, so long as the number of primitive events per predicate remains small.

On balance, this author recommends that users be encouraged to employ the compound-precedence relation given by the first definition, but that, in the interest of maintaining the greatest possible semantic expressiveness, that the other relation should also be made available, with the caveat that its use may cause significant performance degradation.

2.6 "Recommended" Precedence Relation

Definition 5 (Precedence for Compound Events)

Let A and B be compound events. Then $A \rightarrow B$ iff $\forall a \in A : \forall b \in B : a \rightarrow b$.

Definition 6 (Concurrency for Compound Events)

$A \parallel B$ iff $\neg(A \rightarrow B) \wedge \neg(B \rightarrow A)$.

Definition 7 (Front Timestamp of a Compound Event)

*Let A represent a compound event; then the **front timestamp**[4] of A, $T_{A.front}$, is defined as follows: $\forall i$, the ith component of the vector $T_{A.front}$, $T_{A.front}[i] = \min_{a \in A} T_a[i]$.*

Definition 8 (Back Timestamp of a Compound Event)

*The **back timestamp** of A is defined in a similar manner: $\forall i : T_{A.back}[i] = \max_{a \in A} T_a[i]$*

Corollary 1

Notice that, if $A = \{a\}$ (i.e., the compound event is composed of a single primitive event), then $T_{A.front} = T_{A.back} = T_a$, the conventional primitive-event timestamp; compound events are a superset of primitive events.

[4]This concept and the front/back naming convention were introduced by Basten [4].

A timestamp-based compound-precedence test can be achieved by extending any arbitrary primitive-event timestamping scheme. This thesis introduces an implementation based on Fidge timestamps but, first, considers the conceptually simpler problem of constructing such a test using Summers timestamps.

2.6.1 Using Summers Timestamps

The availability of a process-independent precedence test makes Summers' timestamping scheme conceptually attractive; based on it, one can construct a compound timestamp test, using these front and back timestamps, which does not need to know which processes are involved in the events being compared.

Theorem 4 (Summers-Based Compound Timestamp Test)

Let A, B be compound events, then $A \rightarrow B$ iff $T_{A.back} \overset{vec}{\leq} T_{B.front}$.

Proof:

Start with the left-hand side.

$A \rightarrow B$

compound precedence, Definition 5

$\Leftrightarrow \forall a \in A : \forall b \in B : a \rightarrow b$

Summers process-independent timestamp test, Theorem 2

$\Leftrightarrow \forall a \in A$: $\forall b \in B : T_a \overset{vec}{\leq} T_b$

Definition 4 ($\overset{vec}{\leq}$)

$\Leftrightarrow \forall i : \forall a \in A : \forall b \in B : T_a[i] \leq T_b[i]$

$\Leftrightarrow \forall i : \max_{a \in A} T_a[i] \leq \min_{b \in B} T_b[i]$

Definitions 7 and 8 (front and back timestamps)

$\Leftrightarrow \forall i : T_{A.back}[i] \leq T_{B.front}[i]$

Definition 4

$\Leftrightarrow T_{A.back} \overset{vec}{\leq} T_{B.front}$

which is the right-hand side. Since each step of the above reasoning involves a bidirectional implication, the proof is complete. $\square$

Notice that, by Corollary 1, Theorem 4 still holds if one (or both) of the events involved is composed of a single primitive event.

One drawback of the above timestamp test is that it requires up to n comparisons to determine the precedence relation. By applying Theorem 3, this can be reduced to a single comparison when the left operand is a primitive event.

Theorem 5 (Special-Case Optimization of Theorem 4)

If a is a primitive event which occurs on process p_a, and B is a compound event, then $a \to B$ iff $T_a[p_a] \leq T_{B.front}[p_a]$.

Proof:

$a \to B$

$\Leftrightarrow \forall b \in B : a \to b$

Summers process-based timestamp test, Theorem 3

$\Leftrightarrow \forall b \in B : T_a[p_a] \leq T_b[p_a]$

$\Leftrightarrow T_a[p_a] \leq \min_{b \in B} T_b[p_a]$

$\Leftrightarrow T_a[p_a] \leq T_{B.front}[p_a]$ $\square$

2.6.2 Using Fidge Timestamps

It is tempting to think that the result from Theorem 4 might also be applicable with Fidge timestamps, substituting $<$ for $\leq$. However, as Example 3 illustrates, that hypothetical timestamp test would lead to incorrect conclusions. The appropriate timestamp test for use with a Fidge-based system requires a different compound timestamp in place of the back timestamp $T_{A.back}$ given in Definition 8. To calculate this timestamp, an extra bit-vector of information, describing which processes contain events which are members of A, must either be maintained along with the timestamp, or calculated at comparison time.

Definition 9 ($active()$)

Given a compound event A*,* $active(A) := \{i \in \mathcal{Z} \mid \exists a \in A : \; i = proc(a)\}$*, i.e.,* $active(A)$ *is the set of all processes which contain at least one component primitive event of* A*.*

Definition 10 ($active <$)

As a notational convenience, define the symbol $\overset{act}{<}$ *as follows:* $T_A \overset{act}{<} T_B$ *iff* $\forall i \in active(A) : T_A[i] < T_B[i]$*.*

Example 3 (Counterexample: Theorem 4 with Fidge timestamps)

Theorem 4, with the naive substitution of $<$ for $\leq$, does not work with Fidge timestamps. Using the trace shown in Figure 2.7, let $A := \{a, c\}$ and $B := \{b\}$; then, $T_{A.front} = [0,0,0]$, $T_{A.back} = [1,2,1]$ and $T_{B.front} = T_{B.back} = [1,2,2]$. Notice that $A \to B$, but $T_{A.back}[0] = T_{B.front}[0] = 1$, so $\exists j : \neg(T_{A.back}[j] < T_{B.front}[j])$.

Definition 11 (Metron Timestamp)

Given a compound event A*, its* ***metron timestamp***,[5] $T_{A.metron}$*, is defined*

[5]The name is derived from the time-honored concept of the μέτρον ἀριστὸν; $\forall i \in active(A) : T_{A.front}[i] \leq T_{A.metron}[i] \leq T_{A.back}[i]$, and it is the "best," or "most appropriate," one for the purpose at hand. See Horace, *Odes, II, 10.*

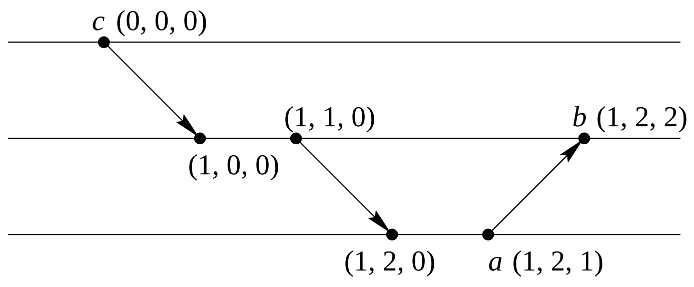


Figure 2.7: Counterexample: Theorem 4 with Fidge time stamps

as follows:

$$T_{A.metron}[i] := \begin{cases} \max\limits_{a \in A,\ p_a = i} T_a[i] \\ -1 \ \textit{otherwise} \end{cases}$$

Theorem 6 (Fidge-Based Timestamp Test for Compound Events)

Given two compound events A and B, then $A \rightarrow B$ iff $T_{A.metron} \overset{act}{<} T_{B.front}$.

Proof:

$A \rightarrow B$

$\Leftrightarrow \forall a \in A : \forall b \in B : a \rightarrow b$

$\Leftrightarrow \forall a \in A : \forall b \in B : T_a[p_a] < T_b[p_a]$

$\Leftrightarrow \forall a \in A : T_a[p_a] < \left(\min\limits_{b \in B} T_b[p_a]\right)$

$\Leftrightarrow \forall a \in A : T_a[p_a] < T_{B.front}[p_a]$

$\Leftrightarrow \forall i \in active(A) : \left(\max\limits_{a \in A,\ p_a = i} T_a[i]\right) < T_{B.front}[i]$

$\Leftrightarrow \forall i \in active(A) : T_{A.metron}[i] < T_{B.front}[i]$

$\Leftrightarrow T_{A.metron} \overset{act}{<} T_{B.front}$ □

Corollary 2

$A \rightarrow B$ *iff* $\forall i : T_{A.metron}[i] < T_{B.front}[i]$.

Proof:

$$A \to B \;\Leftrightarrow\; \forall i \in active(A):\; T_{A.metron}[i] < T_{B.front}[i]$$
$$\bigwedge \forall i \in (\{1,\ldots,n\} \;\setminus\; active(A)):\; T_{A.metron}[i] = -1 \;\wedge\; T_{B.front}[i] \geq 0$$
$$\Rightarrow\; A \to B \;\Leftrightarrow\; \forall i \in \{1,\ldots,n\}:\; T_{A.metron}[i] < T_{B.front}[i] \quad \Box$$

2.7 "Alternative" Precedence Relation

As has been discussed above (see page 18), it is arguably desirable to provide the user with the option of using the following "alternative" definition for compound precedence:

Definition 12 (Alternative Precedence for Compound Events)

$A \leadsto B$ *iff* $\exists a \in A:\; \exists b \in B:\; a \to b.$

This naturally gives rise to a new definition of compound concurrency.

Definition 13 (Alternative Concurrency for Compound Events)

$A \wr\wr B$ *iff* $\neg(A \leadsto B) \wedge \neg(B \leadsto A).$

Corollary 3

$A \wr\wr B$ *iff* $\forall a \in A:\; \forall b \in B:\; \neg(a \to b) \wedge \neg(b \to a).$

Corollary 4

$A \wr\wr B$ *iff* $\forall a \in A:\; \forall b \in B:\; a \,||\, b.$

As has already been mentioned, it is computationally expensive to compute timestamps for this sort of compound-event precedence test. Primitive events, however, can be timestamped reasonably quickly, and the determination of precedence between two arbitrary individual events requires a maximum of two integer comparisons. Thus, for reasonably small compound events, the "brute force" approach can determine the precedence between two *specific* compound events in a reasonable length of time.

Algorithm 1 (Determining Alternative Precedence)

```
function AltPrecedes ($A, B$)
    for each $a \in A$
        for each $b \in B$
            if not $(a \rightarrow b)$
                /* $\neg(a \rightarrow b) \Rightarrow \neg(A \rightsquigarrow B)$ */
                return FALSE ;
    return TRUE ;
```

The running time of this algorithm is, obviously, $O(|A| \cdot |B|)$, where $|A|$ denotes the number of primitive events contained within A.

Nonetheless, as is discussed in Section 4.2.7, the nature of this relationship makes it extremely difficult to restrict the search for partner candidates of a possible sub-match to a relevant subset of the event-trace data. Thus, the use of this definition of compound precedence within a complex predicate-specification does pose a significant performance problem.

2.8 Summary

Building on Lamport's notion of a precedence relationship, Fidge and Summers have both developed timestamps which permit the rapid determination of the precedence relation between two arbitrary primitive events. Extending those mechanisms, this chapter presents two alternative definitions of precedence for **compound** events.

Efficient timestamp mechanisms are developed and proven correct for the first of these compound precedence relations; its use is recommended. A fast-running implementation for the second relation is an elusive goal and may well be unattainable; this author suggests that users should be given the option of using it, with the caveat that it might hinder performance.

Chapter 3

Event-Detection Strategies

3.1 Objectives

3.1.1 General Intent

On a high level, the objective of an event-detection strategy is to address the open problems enumerated in Section 1.2.

First, as its name implies, such a strategy ought to be able to answer the question, "Has x occurred?" This ability could be used to assess what stage a complex computation (e.g., the Cholesky factorization of a matrix) has reached and, thus, to extrapolate how much of the calculation remains to be performed. It could also serve to trigger actions conditionally (e.g., "If x has happened, then do y"), including the conditional breakpoints mentioned on page 4, and, on a related note, to provide a form of global assertion ("If x, y, or z happens, then output an error message and/or abort the computation").

Also, such a strategy should offer an effective means of searching for a particular compound event within the massive process-time diagram which a typical event trace of a distributed application produces, offering a fast and effective way to locate and examine relevant sections within the display produced by a tool such as POET.

3.1.2 Classes of Problems

Stepping down a level, however, the problem becomes somewhat murkier. What sort of compound events is it desirable, or even useful, to detect? Any attempt to answer that question now can only be an incomplete guess. Just as the 1977 business plan for Apple Computer Corporation failed to identify the potential markets for and uses of their new personal computer,[1] the potential applications of distributed systems remain largely uncharted territory, and the questions for which developers and administrators of such systems are likely to seek answers are not yet clear. From a pragmatic point of view, the best course of action at this juncture is to provide people with a sample tool, assess its strengths and weaknesses on the best available facsimile of a large distributed application, and then embark on a process of iterative refinement. It may, nevertheless, be constructive to provide a tentative list of problems and conditions whose detection is desirable; while this list will most likely prove to be incomplete, it should at least provide a framework within which to evaluate and attempt to improve upon various existing event-detection methods.

[1]They "decided to aim for three market segments: the computer hobbyists who were already committed to buying computers; professionals such as doctors and dentists, who had the spare cash to buy gadgets and the intellectual skills to appreciate computers; and the home security/control market, where a computer could conceivably run sprinklers, burglar alarms, lights, and garage door openers" [44, page 138].

1. **Phase Transitions** Perhaps the most obvious type of global event which one may wish to detect is that wherein a distributed computation enters a new, distinct, and stable phase or state. This involves a condition which, once it becomes true, remains true indefinitely. Examples include the termination of a computation (all processes have exited), deadlock, or the loss of some necessary (and, in theory, continually present) commodity, such as the token on a token ring. While far from trivial [19], this problem has been extensively studied, and a reasonable (centralized) algorithm to detect such global events is available [8].

2. **Mutual-Exclusion Violations** A plethora of algorithms has been developed to provide mutual exclusion across a distributed system [7], [37, pages 134-140]. In general, such algorithms are quite complex and, thus, prone to errors of implementation; those of lower complexity may, in certain cases (which are often argued to be highly unlikely) fail to guarantee the required exclusion. Also, when dealing with third-party library routines, it may not be clear *a priori* whether certain actions fail if accessed concurrently. When faced with unexplained, incorrect behaviour, one might hypothesize that this is a result of concurrent accesses to a faulty routine; it could be helpful if a detection mechanism allowed verification that the routine was, indeed, being accessed concurrently, before embarking on the potentially lengthy chore of enforcing mutual exclusion among such calls (Ponamgi, Hseush and Kaiser [31] give an example of such a scenario).

3. **Finding Subroutines** In a large process-time diagram, it can often be extremely difficult just to locate the relevant section of the display, in order to perform a more detailed *ad hoc* examination of some subroutine's behaviour.

Thus, it would be helpful if a tool were available to locate calls to a specific "subroutine" within the context of a large event trace.

4. **Symmetry in Communication** Often, the constraints on inter-process timing are more complex than simple mutual exclusion. For example, a "producer" should not generate items faster than the corresponding consumer can retrieve and process them (see page 41). This relationship can often be quite intricate, involving multiple groups of processes, within the limited flexibility that the bounded buffer provides. It appears that the ability to detect the presence and/or violation of such symmetry could be helpful in the understanding and monitoring of distributed applications.

5. **Identifying Bottlenecks** Ideally, such a tool would also assist in the identification of performance bottlenecks. For example, one might wish to locate positions during an event trace when more than n processes from some set S were simultaneously blocked.

3.2 Previous Work on Event Recognition

This section reviews the history of work related to event recognition, with particular emphasis on the syntactic strengths and weaknesses of the various approaches, and on their resulting expressivity. In the interest of greater clarity, notational substitution has been employed throughout this section; e.g., where various authors have used Lamport's happens-before relation, this is rendered here by the symbol $\rightarrow$, while the papers in question render it variously as $\rightarrow$, $<$, and $\prec$.

3.2.1 Bates-Wileden Event-Definition Language (EDL)

Bates and Wileden [6] made an early attempt to provide a mechanism for the definition of event predicates. They presented a language including the following operators:

- **catenation ('):** $(a \text{ ' } b) \Leftrightarrow (b \text{ occurs after } a)$
- **alternation** ($\vee$)
- **shuffle (ˆ):** $(a \text{ ˆ } b) \Leftrightarrow (a \text{ ' } b \vee b \text{ ' } a)$
- **named subblocks:** names can be assigned to event definitions and these names can, in turn, be used to build further definitions, producing a hierarchy of abstractions
- **binding:** Multiple matches of an event definition can be distinguished from one another by the use of indices which are bound to each instance. For example, the predicate

  node_failed[1] ˆ node_failed[2]

  could be used to detect a situation where two (arbitrary, but distinct) nodes within a multi-node system have failed.

Their approach includes a preprocessor-like filter which removes irrelevant event information, and only passes events that could possibly form a part of a match on to the predicate recognizer; this is a useful performance enhancement.

The Bates-Wileden scheme is fundamentally flawed because it relies upon the availability of an accurate, global time of occurrence for each event. As Chapter 2 mentions, a global clock with the requisite accuracy cannot be made available in an arbitrary distributed system. Also, the use of real-time information instead of causality relationships significantly restricts the expressivity of EDL; e.g., the shuffle operator cannot detect true concurrency (i.e., causal independence). Real-time information can only convey the relative order of events on a particular execution, whereas causality relationships accurately reflect all possible execution orderings.

Nonetheless, filtering, named subblocks and instance binding are useful concepts which have often been overlooked in subsequent work.

3.2.2 Chandy-Lamport Distributed Snapshots

Chandy and Lamport [8] presented another early attempt at event detection. They defined a **stable** predicate to be one which, once it becomes true, remains true for the remainder of the computation, i.e., a phase transition as described in Section 3.1.2.

The authors presented an elegant, if somewhat abstract, algorithm to record a **snapshot** of the global system state. (Essentially, processes cooperate to ensure a consistent representation of the global state by passing "marker" tokens from one to the next.) This algorithm provides the following guarantee: if S_ι is the initial state of a system, S_ϕ is the current state of the system, S_c is a current snapshot, and $y(S)$ is a stable predicate, then

- $y(S_c) \Rightarrow y(S_\phi)$, and
- $\neg y(S_c) \Rightarrow \neg y(S_\iota)$

This allows a process testing for the predicate $y(S)$ to conclude that the stable state has been reached once $y(S_c)$ is true.

While interesting, this approach is of limited use in a debugging context; a human operator can discern the presence of a stable predicate from trace output with relative ease, and the authors do not address the detection of non-stable predicates.

3.2.3 Miller-Choi Predicates

Recognizing the importance of a Lamport-like causality relationship, Miller and Choi [29] proposed the following predicate definition for use in specifying distributed breakpoints:

- **Simple Predicates (SP):** these are single, primitive events.
- **Disjunctive Predicates (DP):** a disjunctive predicate is composed of one or more SPs combined by the logical disjunction operator:

  $$\text{DP} := \text{SP}\ [\vee\ \text{SP}]*$$

  That is, a DP is satisfied iff one or more of its constituent SPs is satisfied.
- **Linked Predicates (LP):** a linked predicate consists of one or more DPs which must occur in the order specified.

  $$\text{LP} := \text{DP}\ [\rightarrow \text{DP}]*$$

- **"Conjunctive" Predicates (CP):** These would more accurately be labeled "concurrent" predicates, since the operator upon which they are based is concurrent composition (||, defined in an unusual fashion because of the absence of vector clocks):

$$\text{CP} := \text{SP } [\| \text{ SP}]*$$

Since this paper appeared prior to the widespread publication of the Fidge / Mattern vector-timestamp breakthrough, it is perhaps unsurprising that the authors do not employ vector timestamps. Most likely because of the lack of this foundation, they restrict their linked predicates to disjunctions of primitive events, and offer a detection algorithm only for linked predicates and not for conjunctive predicates. Thus, the expressive power of their system is rather limited.

By imposing these restrictions, however, their distributed algorithm is able to detect the occurrence of predicates in real time, with a small computation and communication overhead.

3.2.4 Haban-Weigel Predicates

At roughly the same time, Haban and Weigel [20] also proposed a predicate-specification language; following Mattern's suggestion, they incorporated vector timestamps into their proposal. In fact, their specification language is quite powerful, and a model which this author has decided to follow closely.

They define a **global predicate** to consist of single (primitive) events, combined with the following operators:

- **sequential** ($\rightarrow$) and concurrent ($\|$) composition
- **conjunction** ($\wedge$) **and disjunction** ($\vee$)
- **'negation' (@):**[2] a @ b is satisfied iff $\exists a, b : a$ is satisfied and b has not (yet) become satisfied

[2]This should not be confused with the Hseush-Kaiser closure operators, nor with the repetition operators introduced in Section 3.3.3.

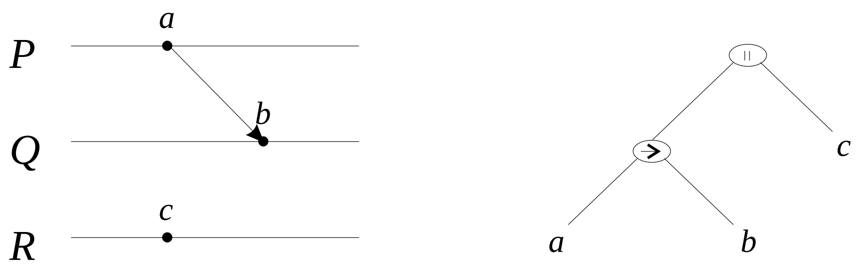


Figure 3.1: Example Haban-Weigel predicate

- **between (@):** $@b(a, c)$ is satisfied iff $\exists a, c : a \rightarrow c$ is satisfied, and $\nexists\ b$ such that $a \rightarrow b \rightarrow c$

The *between* operator is an interesting innovation. It allows the specification of predicates to detect communication-synchronization problems, a goal which most predicate-definition languages are unable to meet. Nevertheless, the Haban-Weigel syntax lacks a key feature: it should be possible to employ wildcards in the specification of primitive events, because predicates to detect certain situations will otherwise balloon into an unwieldy disjunction of combinatorial possibilities.

In an effort to distribute the load of the detection algorithm throughout the system, the authors maintain a local debugger (d_i, $i = 1 \ldots n$, where n is the number of nodes in the system) on each node, and a coordinating central test station (CTS). The latter builds a parse tree for the predicate and distributes a copy of it to each of the former.

Example 4

Consider Figure 3.1. To detect the tri-event predicate depicted in the process-time diagram on the left, Haban and Weigel would use the predicate $(a \rightarrow b) \parallel c$, which yields the parse tree depicted on the right.

Whenever an event e_i occurs on a node i, the d_i applies it to the parse tree by attaching its timestamp T_{e_i} to all leaf nodes, and then tests the tree to determine whether the predicate is satisfied. So that each local debugger can maintain an overview of the entire execution, the debugger d_i notifies each d_j, $j \neq i$, of the occurrence of every event e_i and its corresponding timestamp. Since no attempt is made to queue arriving events and to enforce a consistent state, non-uniform delays in inter-node communication could cause the parse tree to reflect a state which is inconsistent with the program's execution, thus leading to the detection of spurious predicates and/or the failure to detect actual occurrences. It is also interesting to note that every local debugger does as much predicate-testing computation as would be required of a central debugger, and that the debugging system's total communication overhead is n times greater than that which would be required if a single, central debugger were used. The moral is clear: while a means of distributing the task of event-detection may be desirable, a poor distributed algorithm can perform significantly worse than a simple, centralized algorithm, and the prospect of designing an effective distributed algorithm is a daunting one.

There is, however, a more fundamental flaw in the Haban-Weigel approach. They use a single vector timestamp, equivalent to the *back* timestamp from Definition 8, to describe the time at which a compound event has occurred; as Schwarz and Mattern [32] point out, no *single* timestamp can accurately convey precedence information for a non-atomic event. In order to achieve a well-defined, sensible precedence relation over anything other than primitive events, a compound-event timestamp scheme like those discussed in Chapter 2 must be employed.

3.2.5 Hseush-Kaiser Data-Path Expressions

Hseush and Kaiser [21, 31] avoided the problem of extending vector time to compound events by doing completely without vector clocks. Their **data-path expressions (DPEs)**, based on primitive events, are built up recursively using the following operators:

- **sequencing (;):** $a; b$ iff a is the immediate causal predecessor of b (not equivalent to $\rightarrow$)
- **concurrency** ($\|$)
- **exclusive 'or'** ($\oplus$)
- **sequential closure** ($*$)**:**[3] $a* := \varepsilon \oplus a \oplus (a; a) \oplus (a; a; a) \oplus \cdots$
- **concurrent closure (@):** $a@ := \varepsilon \oplus a \oplus (a \parallel a) \oplus (a \parallel a \parallel a) \oplus \cdots$

For event recognition, their approach employs a central "stabilizer" which receives event-trace information from the various debuggees, filters out those events which are obviously irrelevant to the predicate under consideration, linearizes the remainder in a manner consistent with the causal partial-order, and then feeds the resulting stream of events to a **predecessor automaton (PA)** (see Figure 3.2). Predecessor automata resemble finite-state automata, but each transition is labeled with a vector. The first element of this vector is the event which must be encountered to trigger the transition, and the remaining elements form the immediate causal predecessors which the event must have in order for the transition to take place.

[3]The reader should take care not to confuse these closure operators with the repetition operators introduced in Section 3.3.3, nor with Haban-Weigel's @.

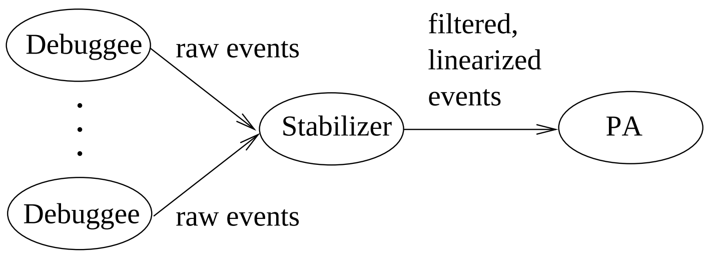


Figure 3.2: Hseush-Kaiser event recognizer

PAs have problems with ambiguities [4, 32], but the greatest drawback to the Hseush-Kaiser approach is the use of the sequencing operator (;) instead of Lamport's happens-before concept ($\rightarrow$). To render $a \rightarrow b \rightarrow c$ in a DPE requires the syntax

$$a; (a \oplus c \oplus X)*; b; (a \oplus b \oplus X)*; c$$

where X denotes the exclusive selection of all other primitive events which occur anywhere in the DPE. Automatic generation of DPEs is, of course, possible for the $\rightarrow$ relation, but it is not feasible for the Haban-Weigel *between* operator [32]. Thus, PAs cannot achieve the level of expressivity required to meet all of the goals set forth in Section 3.1.2.

3.2.6 Garg-Waldecker Weak Conjunctive Predicates

Garg and Waldecker [17] recognized the importance of the ability to detect unstable predicates, and further observed that a feasible detection algorithm must make use of Lamport's happens-before relation. In the interest of simplicity (and, hence, tractability), they defined a limited global predicate as follows:

Definition 14 (WCP)

*A **weak conjunctive predicate** consists of one or more primitive events e_i "conjoined"[4] by the concurrent composition operator ($\|$); i.e., a WCP is true iff all of its primitive events are concurrently true:*

$$WCP := e_1\ [\| e_i]*$$

In order to detect the occurrence of WCPs, the authors proposed the following strategy: all events (and their corresponding timestamps) are sent to a central "checker" process, which then scans for occurrences of the specific primitive events, and determines whether they are concurrent in the Lamport sense. They provided an algorithm for this checker, and also demonstrated how the process can be divided among a hierarchy of checkers.

Like the various Miller-Choi predicates, the expressiveness of WCPs is extremely limited; they can be used to verify mutual exclusion, but little else. For example, they do not offer any provision for sequential relationships ($\rightarrow$) within a predicate.

3.2.7 Chiou-Korfhage Event-Normal-Form Predicates

Chiou and Korfhage [10, 11, 12] built on Garg and Waldecker's [17] work, attempting to expand it and reduce its limitations. They adopted the weak conjunctive predicate, albeit under the new name **concurrent event string (CES)**, and then defined a **sequential event string (SES)** to be the sequential composition of one or more CESs, i.e.,

[4]Note that, while Garg and Waldecker use the term "conjunctive" and the traditional symbol for logical conjunction ($\wedge$) in their WCPs, they assign to it the meaning of concurrent composition; this is but one of many examples of the current confusion over terminology in the field.

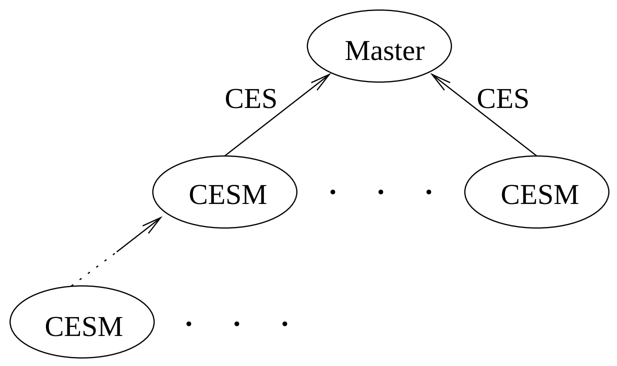


Figure 3.3: Chiou and Korfhage's distributed ENF-predicate recognizer

$$\text{SES} := \text{CES} \; [\rightarrow \text{CES}]*$$

Finally, their approach allows the combination of sequential event strings with the disjunction operator.

Definition 15 (ENF)

An event-normal-form predicate consists of one or more SESs combined with logical disjunction operators:

$$ENF := SES \; [\vee \; SES]*$$

As with Garg and Waldecker's WCPs, the detection of concurrent event strings can be distributed throughout the system with a hierarchical structure of "checker" processes, or, as Chiou and Korfhage term them, **concurrent event string monitors (CESMs)**. Each instance of a CES is eventually detected and passed to a central master process, which checks for the occurrence of SESs and, eventually, ENF predicates (Figure 3.3).

While event-normal-form predicates offer a greater degree of expressiveness than WCPs (in addition to detecting violations of mutual exclusion, they can also locate

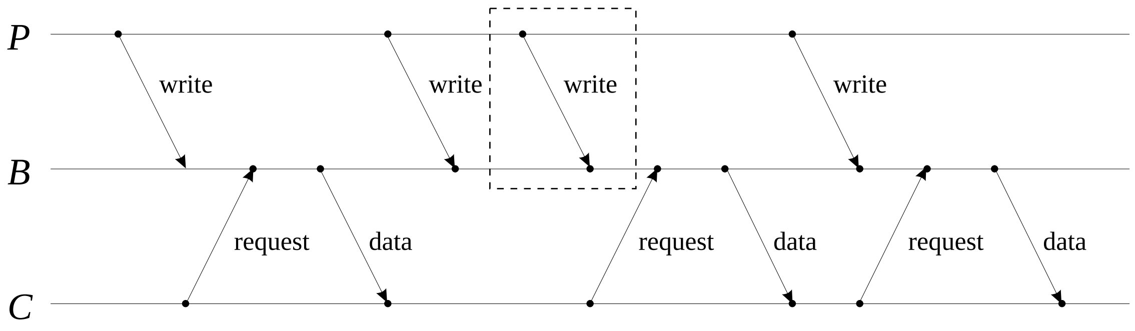


Figure 3.4: Producer-consumer race problem

occurrences of some subroutine calls), their sequential-composition operator ($\rightarrow$) still provides only a limited amount of control and, in particular, cannot verify communication symmetry. Consider the following problem:

Example 5

> Consider the situation depicted in Figure 3.4. A producer P and a consumer C are writing to and reading from, respectively, a bounded buffer B of size one. Some form of control logic (its details are unimportant, and are not discussed here) is in place to regulate the reads and writes in order to prevent the buffer from overflowing. There is an error in this control logic, and, at one point (inside the dashed-line box), the producer goes too fast, thus overflowing the buffer.

What is needed in order to detect this error condition is a method, such as Haban and Weigel's *between* operator, through which one may ascertain how "closely" two sequential events (a, b) are related (including, in particular, what intermediate events e_i may lie in-between, such that $a \rightarrow e_i \rightarrow b$). Because ENF predicates lack such a facility, they are inherently unable to detect an error condition like that in Example 5.

3.2.8 Basten's PLR Parsing

Basten [3, 4] recognized that, since Lamport's happens-before relation ($\rightarrow$) yields a partial order, an event trace from a distributed execution may be viewed as a partially ordered multiset ("pomset").

Conventional LR parsing, practised, e.g., in modern compilers, deals with totally ordered multisets ("tomsets"). (Aho *et al.* [1, Section 4.7] is the definitive introduction to this area.) Basten extended this mechanism to pomsets, and provided an algorithm to generate tables for, and parse, PLR grammars. His algorithm relies on first producing a linearization of the pomset (i.e., an arbitrary total ordering which is compatible with the partial ordering); the events are then fed to the parser in this (artificial) sequence, and the parser attempts to afford each event a position within the structure defined by the PLR grammar. Unfortunately, Basten's algorithm deals only with the recognition of entire pomsets, and ignores the problem of detecting subpomsets within the event-trace input.

Cormack [13] provided an extension to the LR parsing of tomsets which allows it to recognize substrings within a larger input context. His algorithm augments the traditional closure operator to include arbitrary prefix information, so that the resulting parse-action tables cause a substring y to be recognized within the context vy, for any possible prefix v. Cormack's goal was to provide graceful error-recovery during program compilation, and his approach is quite successful in this regard. However, his parser must reset itself whenever it encounters a token that does not fit into the grammar; this strategy does not work with PLR parsing, since a linearization may leave multiple extraneous events within a valid instance of a subpomset.

While an efficient subpomset PLR parsing algorithm may be possible, its form is far from obvious. On a more subjective note, context-free parallel grammars ("CFPGs") provide a method of predicate specification which, while powerful, is complex and difficult to grasp. Drawing an analogy with the totally ordered input of modern compilers, one can observe that formal grammars are the overwhelming tool of choice among compiler writers, who require a detailed understanding of the entire input; in general, however, programmers employ much simpler pattern specifications, such as regular expressions, when searching through source code during the construction and debugging phases. What is needed seems to be a practical compromise between the time required to construct a query and the precision of the result.

3.2.9 Seuren Communication Patterns

Seuren [34] sought to provide an automated mechanism to abstract away "units of work" within an execution trace. With this goal in mind, he restricted his objective to deal only with sets of events that form a totally connected subgraph, which he termed **communication patterns**. In this context, he presented an efficient search algorithm based on the Boyer-Moore substring-matching approach.

Example 6

> What does—and does not—qualify as a *communication pattern*: Seuren's patterns must be totally connected, and may not have any non-member events participating in the connections. Thus, in the event trace of Figure 3.5, $\{a, b, d, e\}$ can constitute a pattern, but $\{a, b, d, f\}$ and $\{a, i\}$ cannot.

While Seuren's restrictions on the composition of patterns are reasonable in the context of automated event abstraction, they limit the expressiveness of the resulting specification mechanism to an extent that severely hampers its ability to

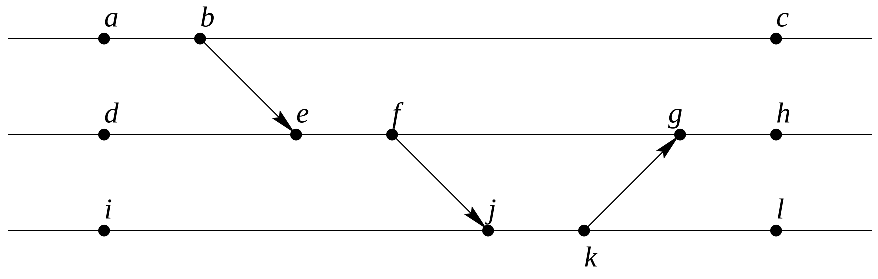


Figure 3.5: Sample *communication patterns*

constitute a reasonable predicate-specification system in a debugging or monitoring context. Regrettably, Seuren's algorithm is intrinsically linked to this restriction, and is not applicable in a monitoring or trace-searching setting.

3.3 Proposed Predicate-Specification Mechanism

The most common stumbling block among previous attempts at event recognition is the lack of a solid definition for compound precedence; of the above algorithms, only Basten's (theoretical) PLR parsing, and Seuren's (for this application, overly limited) communication patterns allow predicates to be constructed as a hierarchy of successive abstractions while correctly preserving a causality relationship.

Starting with the rich Haban-Weigel syntax, this thesis adds a few refinements to create a new, more expressive predicate-definition language built on the carefully-defined compound-precedence relationships described in Chapter 2.

3.3.1 Syntax

The basic *component* of a predicate is a 3-tuple describing a single primitive event: the textual name of the *process* in which the event occurs, the *type* (e.g., process start, asynchronous send) of the event, and an additional *text*ual comment which may have been included either automatically (e.g., in the case of a remote procedure call, the name of the procedure being invoked) or manually (e.g., a programmer-defined call in the source code could generate an event indicating that a certain checkpoint had been reached in the code's execution). Each of these individual entries can, at the user's option, be left blank (in which case it is treated as a "match-anything" wildcard) or filled with a regular expression. If an entry matches more than one *process*, *type* or *text*ual comment, then all matches are considered in the course of a backtracking search for occurrences of the overall predicate.

A communication "*event*" can be identified by joining two *components*, representing the send and the receive, with a period (i.e., *send.receive*).

These *events* can be composed by using the standard boolean operators ($\wedge$, $\vee$), parentheses, and the precedence relationships discussed in Chapter 2. To provide a greater degree of expressiveness, **limited** versions of the happens-before relationships (similar to the Haban-Weigel *between* operator) are introduced.

Definition 16 (Limited $\rightarrow$)

$$A \overset{B}{\rightarrow} C \textit{ is satisfied iff } \exists A, C :\ (A \rightarrow C)\ \wedge\ (\nexists B :\ A \rightarrow B \rightarrow C)$$

Definition 17 (Limited $\rightsquigarrow$)

$$A \overset{B}{\rightsquigarrow} C \textit{ is satisfied iff } \exists A, C :\ (A \rightsquigarrow C)\ \wedge\ (\nexists B : A \rightsquigarrow B \rightsquigarrow C).$$

The grammatical structure of this specification language is summarized in Figure 3.6.

$$
\begin{array}{rcl}
\mathit{predicate} & ::= & \mathit{term} \\
\\
\mathit{term} & ::= & \mathit{term} \rightarrow \mathit{term} \\
 & | & \mathit{term} \parallel \mathit{term} \\
 & | & \mathit{term} \rightsquigarrow \mathit{term} \\
 & | & \mathit{term} \wr\!\wr \mathit{term} \\
 & | & \mathit{term} \overset{\mathit{term}}{\rightarrow} \mathit{term} \\
 & | & \mathit{term} \overset{\mathit{term}}{\rightsquigarrow} \mathit{term} \\
 & | & \mathit{term} \wedge \mathit{term} \\
 & | & \mathit{term} \vee \mathit{term} \\
 & | & (\ \mathit{term}\ ) \\
 & | & \mathit{event} \\
\\
\mathit{event} & ::= & \mathit{component}\ .\ \mathit{component} \\
 & | & \mathit{component} \\
\\
\mathit{component} & ::= & [\ \mathit{process}\ ,\ \mathit{type}\ ,\ \mathit{text}\ ]
\end{array}
$$

Figure 3.6: A generating grammar for the proposed predicate language

3.3.2 Expressivity

To assess the degree of expressive control which is offered by this proposed language, consider the problem classes enumerated in Section 3.1.2, all of which an ideal predicate-specification system should be able to detect.

3.3.2.1 Phase Transitions

Certain types of transitions can be detected quite easily using this system. For example, if a predicate conjunctively specifies a *thread-terminate* message for each individual thread, then it is satisfied iff the entire distributed computation has terminated. Other transitions, e.g., deadlock detection, remain a daunting task; an astute user could, however, use this mechanism to specify a predicate to indicate when deadlock may be present, and then define a more in-depth (and, presumably, more costly) analysis using another tool, to be invoked whenever the initial predicate was satisfied.[5]

Example 7 (Termination Detection)

Consider the execution in Figure 3.7. The text beside each event indicates its event type. The predicate

$$[\ \mathrm{A}\ ,\ \mathrm{thread\_end}\ ,\ ]\ \wedge\ [\ \mathrm{B}\ ,\ \mathrm{thread\_end}\ ,\ ]\ \wedge\ [\ \mathrm{C}\ ,\ \mathrm{thread\_end}\ ,\ ]$$

is satisfied iff the computation has terminated.

[5]Alternatively, the user could define a non-exhaustive set of assertions whose failure (i.e., detection) would indicate the presence of a deadlock.

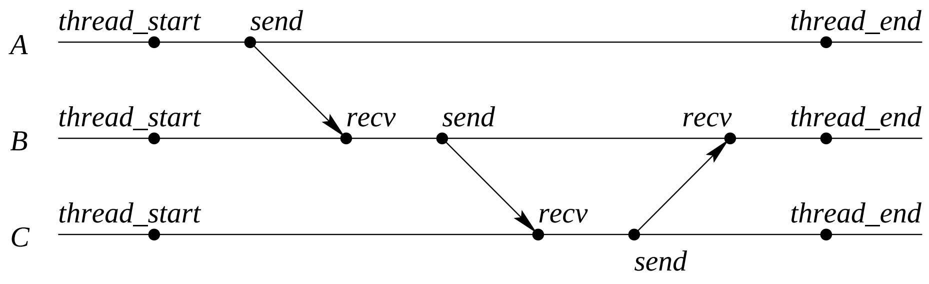


Figure 3.7: Output trace discussed in Example 7

3.3.2.2 Mutual-Exclusion Violations

These are trivially specifiable through the use of the concurrent-composition operator.

Example 8

> Let A and B specify two events which must not be allowed to occur concurrently. Then the predicate $A \parallel B$ is satisfied iff the exclusion is violated.

3.3.2.3 Finding Subroutines

Most event-collection systems include enough information in the trace data that the users are able to specify predicates that locate calls into, and returns from, remote subroutines. If the existing trace data are not specific enough for this purpose, users can always augment the available information by hand-coding instrumentation at the entry to and exit from the relevant procedures.

Example 9

> Consider a program written in the μC++ language [7], and then traced with the default POET instrumentation. A request to lock a semaphore produces

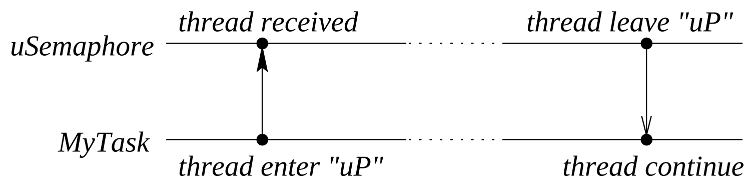


Figure 3.8: μC++ trace fragment referred to in Example 9

the event-trace fragment shown in Figure 3.8. (The task *MyTask* makes a routine call to lock the semaphore. The comment beside each event is the event type; textual "decorations" are appended in quotation marks, where appropriate.) The predicate

$$A := [\text{ MyTask , thread enter , uP }] \,.\, [\text{ uSemaphore , thread received , }]$$

matches all calls from *MyTask* to lock the semaphore,

$$B := [\text{ uSemaphore , thread leave , uP }] \,.\, [\text{ MyTask , thread continue , }]$$

matches all returns from the above call, and

$$A \overset{A}{\rightarrow} B$$

matches each entire call-and-return sequence.

3.3.2.4 Symmetry in Communication

The limited $\rightarrow$ and $\rightsquigarrow$ operators provide a degree of control that enables the user to detect the presence of simple communication-symmetry problems quite easily.

Nonetheless, the definition of predicates to check complex symmetry patterns can be quite tedious and confusing; some of the extensions sketched in Section 3.3.3 might help to alleviate this problem.

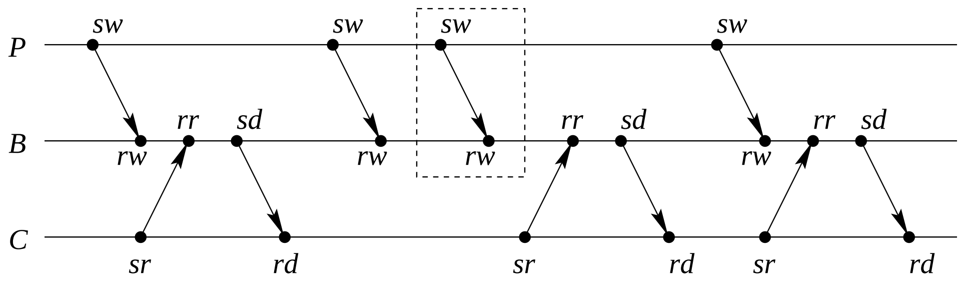


Figure 3.9: Producer-consumer race revisited

Example 10 (Producer-consumer race revisited)

Recall the situation mentioned in Example 5: a producer and a consumer are communicating via a bounded buffer of size one. Examining Figure 3.9, notice that the send and receive portions of each message are separately designated: *sw* = *send write*, *rw* = *receive write*, *sr* = *send request*, *rr* = *receive request*, *sd* = *send data*, and *rd* = *receive data*. As in the earlier figure, a dashed line surrounds the extraneous write which indicates a synchronization problem and causes the buffer to overflow. Using this notation, the predicate

$$[B, rw,] \overset{[B,sd,]}{\rightarrow} [B, rw,]$$

detects the buffer-overflow problem depicted in the figure.

3.3.2.5 Identifying Bottlenecks

As is clear from the bounded-buffer example, this class of problem is actually a slight variation on the previous one. Perhaps unsurprisingly, the proposed system has the flexibility and degree of control required to detect simple bottlenecks. Moderately large-scale situations, however, require an extension to the syntax; this is dealt with in Section 3.3.3.

Example 11

Once again, consider a simple semaphore, used to lock access to a particular feature which can only be accessed by one thread of control at a time. As has been demonstrated in Example 9, the μC++ language and POET instrumentation translate a request by a process, say *Fred*, to lock the semaphore into the communication pair

[Fred , thread enter , uP] . [uSemaphore , thread received ,]

and the corresponding lock-achieved message into the pair

[uSemaphore , thread leave , uP] . [Fred , thread continue ,]

Likewise, the request-release / release-achieved messages would be

[Fred , thread enter , uV] . [uSemaphore , thread received ,]

and

[uSemaphore , thread leave , uV] . [Fred , thread continue ,]

Now, consider the situation where a second process, say *Jane*, also requests a lock on the same semaphore while *Fred* has it locked. Then the (conceptual) thread of control blocks inside the *uSemaphore* trace until it becomes possible to grant *Jane* the lock; this can be seen in Figure 3.10. (Dashed triangles indicate the points at which *Fred* and *Jane* receive confirmation that the semaphore has been locked (∇) or unlocked (Δ).) To detect all situations where a process is waiting for a semaphore to become available, it is sufficient to specify the single-event predicate

A := [uSemaphore , thread block ,]

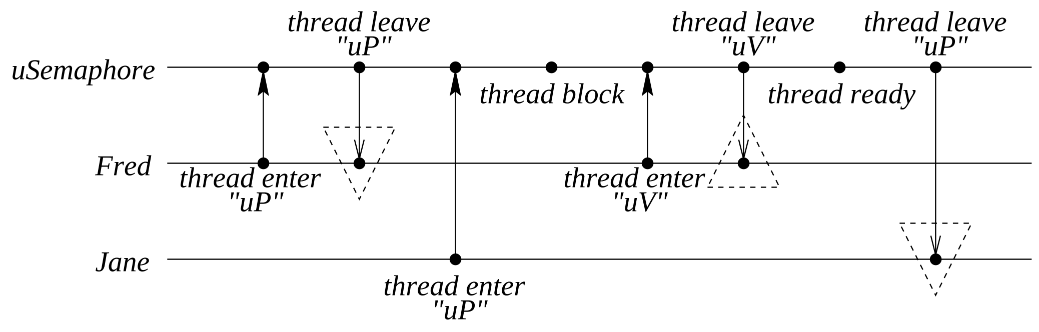


Figure 3.10: A thread blocks while waiting to acquire a lock

For notational convenience, define the predicate

$$B := [\text{ uSemaphore , thread ready , }]$$

Then the predicate

$$A \overset{B}{\rightarrow} A$$

(two consecutive blocks with no intervening ready) can be used to detect the more complex situation where two or more processes are blocked simultaneously.

This approach does not, however, extend to the detection of situations when three or more processes are simultaneously blocked. The predicate

$$A \overset{B}{\rightarrow} A \overset{B}{\rightarrow} A$$

might be expected to achieve this effect. Consider, however, the situation in Figure 3.11: tasks are blocked (dashed circles) four times, and unblocked (dashed square) once; at the end of the diagram, tasks 1, 4, and 3 are simultaneously blocked, but the above-mentioned predicate is not satisfied. What is needed is a predicate with a functionality equivalent to the following:

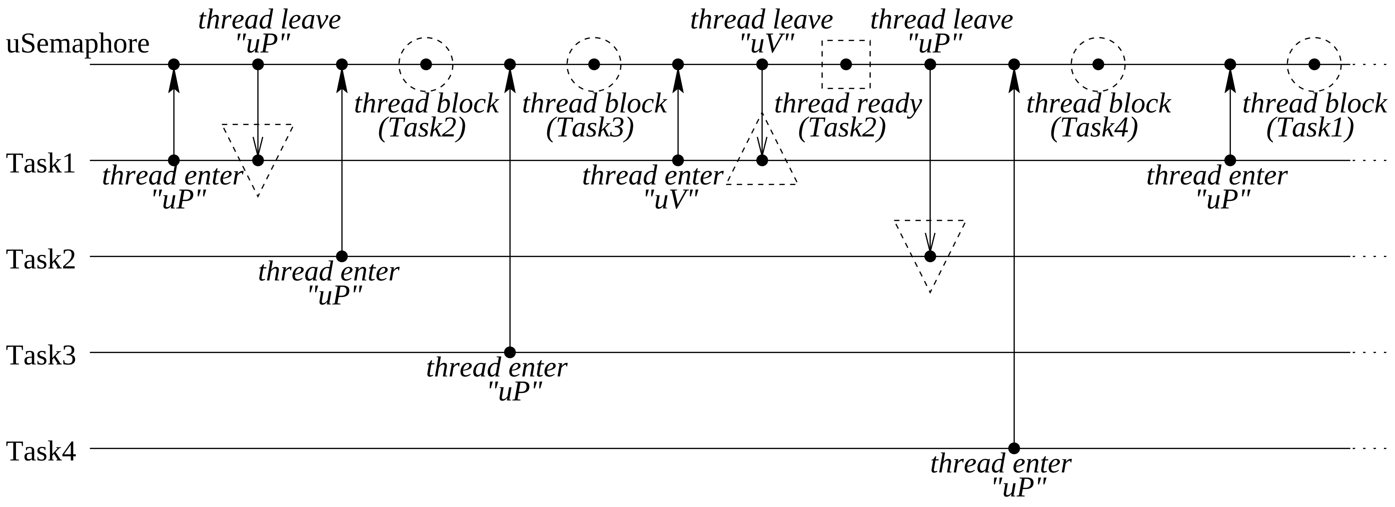


Figure 3.11: Three threads are simultaneously blocked

$$
\begin{gathered}
C := A \vee B; \\
(A \overset{C}{\rightarrow} A \overset{C}{\rightarrow} A) \bigvee \\
(A \overset{C}{\rightarrow} A \overset{C}{\rightarrow} B \overset{C}{\rightarrow} A \overset{C}{\rightarrow} A) \bigvee \\
(A \overset{C}{\rightarrow} A \overset{C}{\rightarrow} B \overset{C}{\rightarrow} A \overset{C}{\rightarrow} B \overset{C}{\rightarrow} A \overset{C}{\rightarrow} A) \bigvee \\
\ldots
\end{gathered}
$$

In other words, some partial-order versions of the regular-expression repetition $(*)$ operator are needed. This point is addressed in Section 3.3.3.

3.3.3 Possible Extensions

Thus, the syntax proposed above is sufficient to meet the first four goals set forth in Section 3.1.2, and to partially fulfill the fifth. Nevertheless, the previous work discussed above suggests some further extensions whose addition to the predicate-definition language could prove to be profitable.

Repetition Operators

The elusive fifth goal can be met in a more satisfactory manner if the syntax is extended to allow arbitrary repetitions of compound events, joined by $\rightarrow$ and $\rightsquigarrow$ operators.

Definition 18 (Repetition Operator)

$A * B$ *is satisfied iff* $B \bigvee (A \rightarrow B) \bigvee (A \rightarrow A \rightarrow B) \bigvee \cdots$

Definition 19 (Alternative Repetition)

$A @ B$ *is satisfied iff* $B \bigvee (A \rightsquigarrow B) \bigvee (A \rightsquigarrow A \rightsquigarrow B) \bigvee \cdots$

Definition 20 (Limited $*$)

$A \overset{B}{*} C$ *is satisfied iff* $C \bigvee (A \overset{B}{\rightarrow} C) \bigvee (A \overset{B}{\rightarrow} A \overset{B}{\rightarrow} C) \bigvee \cdots$

Definition 21 (Limited @)

$A \overset{B}{@} C$ *is satisfied iff* $C \bigvee (A \overset{B}{\rightsquigarrow} C) \bigvee (A \overset{B}{\rightsquigarrow} A \overset{B}{\rightsquigarrow} C) \bigvee \cdots$

Example 12

Recall the predicate, sketched in Example 11, to identify the eventuality that three or more processes are simultaneously blocked waiting for a specific semaphore.

$$\begin{gathered}(A \overset{C}{\rightarrow} A \overset{C}{\rightarrow} A) \bigvee \\ (A \overset{C}{\rightarrow} A \overset{C}{\rightarrow} B \overset{C}{\rightarrow} A \overset{C}{\rightarrow} A) \bigvee \\ (A \overset{C}{\rightarrow} A \overset{C}{\rightarrow} B \overset{C}{\rightarrow} A \overset{C}{\rightarrow} B \overset{C}{\rightarrow} A \overset{C}{\rightarrow} A) \bigvee \\ \cdots\end{gathered}$$

This can be re-written as

$$A \overset{C}{\rightarrow} A \overset{C}{\rightarrow} (B \overset{C}{\rightarrow} A) \overset{C}{*} A$$

Similarly, the following predicate could be used to detect whether four or more processes are ever simultaneously blocked:

$$A \overset{C}{\rightarrow} A \overset{C}{\rightarrow} (B \overset{C}{\rightarrow} A) \overset{C}{*} A \overset{C}{\rightarrow} ([B \overset{C}{\rightarrow} A] \vee [B \overset{C}{\rightarrow} (B \overset{C}{\rightarrow} A) \overset{C}{*} A]) \overset{C}{*} A$$

"Not Yet" Operator

Although wildcards combined with the limited $\rightarrow$ and $\rightsquigarrow$ operators almost subsume its functionality, there are still some situations wherein Haban and Weigel's simple *negation* operator can be useful.

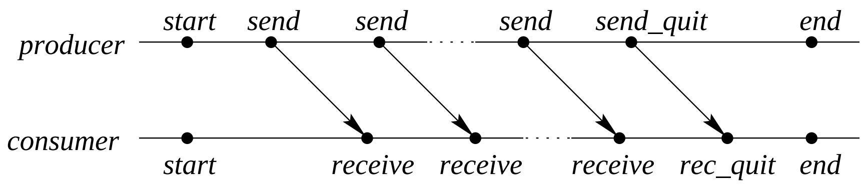


Figure 3.12: Producer-consumer shutdown

Example 13

Consider the situation depicted in Figure 3.12. A *producer* sends a succession of data items to a *consumer*; when the *producer* is finished, it sends a *quit* message to the *consumer*, and both processes terminate. The *consumer* should not terminate before the *producer* has sent the *quit* message; this error condition could be detected using a "not yet" operator ($\vdash\!\!\rightarrow$).

$$[\text{ consumer , end , }] \vdash\!\!\rightarrow [\text{ producer , send\_quit , }]$$

Named Subblocks

As has informally been used in some of the examples above, the ability to assign names to building blocks within a predicate can simplify the process of defining certain predicates.

Example 14

The predicate

$$[\text{ semaphore , block , }] \overset{[\text{semaphore,ready,}]}{\longrightarrow} [\text{ semaphore , block , }]$$
$$\overset{[\text{semaphore,ready,}]}{\longrightarrow} [\text{ semaphore , block , }]$$

could be re-written as

$$A := [\text{ semaphore, block },] ;$$
$$B := [\text{ semaphore, ready },] ;$$
$$A \overset{B}{\rightarrow} A \overset{B}{\rightarrow} A$$

Binding

The ability to bind specific instances of matches to sub-patterns with instance numbers or variables would allow the specification of predicates that can otherwise only be described in a roundabout manner, if at all.

Example 15

Consider the search for a child process whose first action, immediately following startup, is to spawn another child (see Figure 3.13). This sequence can be specified using wildcards with the limited $\rightarrow$ relation

$$[\text{ , spawn , }] \; . \; [\text{ child.}* \text{ , start , }] \overset{[\text{child.}*,,]}{\longrightarrow} [\text{ child.}* \text{ , spawn , }]$$

or, equivalently, by binding instances

$$\{1\} := \text{“child.}*\text{”} ;$$
$$[\text{ , spawn , }] \; . \; [\; \{1\} \text{ , start , }] \overset{[\{1\},,]}{\longrightarrow} [\; \{1\} \text{ , spawn , }]$$

If, however, the search is for a child that spawns another child after an indeterminate number of intervening events, then this capability is indispensable. The predicate

$$\{1\} := \text{“child.}*\text{”} ;$$
$$[\text{ , spawn , }] \; . \; [\; \{1\} \text{ , start , }] \rightarrow [\; \{1\} \text{ , spawn , }]$$

cannot be specified without the ability to bind instances.

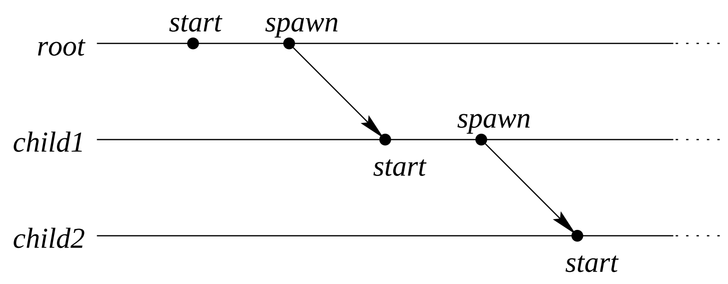


Figure 3.13: A child (immediately) spawns another child

Real-Time Information

The ability to add real-time restrictions to predicates would be particularly helpful for performance analyses. For example, the situation wherein a user is blocked, waiting for a lock, for more than 5 seconds, might be specified as follows:

$$\{1\} := \text{“user.}*\text{”} \; ;$$
$$[\ \{1\}\ ,\ \text{block}\ ,\ ]\ \xrightarrow{>5sec.}\ [\ \{1\}\ ,\ \text{unblock}\ ]$$

Chapter 4

Implementation

4.1 POET

4.1.1 Motivation

The Partial-Order Event Tracer (POET) is an event-visualization tool developed by the Shoshin research group at the University of Waterloo [38, 40]. There are several reasons why POET provides a good base for event-detection investigations:

- **Solid (tested) foundation.** The POET system has been in use for four years, and the algorithms to timestamp events and display process-time diagrams are reasonably fast and largely bug-free.

- **Target-independent and abstraction-capable.** POET is capable of dealing with a number of different target (debuggee) operating environments, and can apply several abstractions to its process-time diagrams. The presence of these features in POET allows (or, perhaps, encourages) the development of an event-detection / monitoring mechanism that can cope with such an environment.

- **Appreciable user base.** Since distributed-system programming is still in its infancy, no distributed debugging tool is in widespread use. Nonetheless, POET is the focus of an active research effort at the University of Waterloo, where it is also employed by undergraduates in the introductory concurrent-programming course; as well, its development has been, and continues to be, actively supported by IBM. The objective of this thesis is to provide a prototypical implementation of an event-detection mechanism so that practical experience with it can shed further light on the ideal form for such a tool; POET seems to have the user base that is a prerequisite for such an iterative-refinement process.

4.1.2 Architecture

POET is target-system independent [41], and has been adapted for use with Hermes [38], OSF DCE [43], PVM [25], ABC++ [33], μC++ and, naturally, to debug itself. Various custom parameters for each of these environments are stored in target-description files; the relevant environment is identified when the first debuggee starts, and the event tracer then configures itself accordingly.

Instrumentation is achieved in several different ways, depending on the target environment; these include modifications to code-generators, run-time libraries, the system interpreter, and/or the use of system-specific status information. While the exact information collected varies from one target-environment to the next, attempts are generally made to trace inter-process communication and other significant intra-process events, e.g., process creation, termination, blocking and unblocking. Support is also provided for the client programmer to hand-instrument sections of the source code being examined through the inclusion of calls into the event-collection mechanism. The POET system generates modified Fidge timestamps (extended for synchronous communication as described in Section 2.4.1, with the additional change that each clock is intialized to the zero vector) for the event-trace data, and can use this information to construct a scrollable process-time diagram.

The POET system is, itself, composed of a number of processes. The *event server* receives event data from the target-specific collection mechanism, and stores this in an event-data file. The *debug-session* process manages an interactive graphical interface which includes a scrollable process-time diagram. Finally, the *checkpoint* process produces checkpoints of timestamp information at regular intervals throughout the event-data file; this performance enhancement allows the *debug session* to quickly scroll to another location and redraw the diagram, computing the relevant timestamps starting at a checkpoint instead of at the beginning of the file. Figure 4.1 outlines the communication relationships between these various components.[1]

4.1.3 Display

Different symbols (circles and squares, solid and unfilled) are used to display various classes of event types; this helps to maximize the amount of information provided by the limited available screen real-estate. As well, the process-lines are drawn in different styles (solid, dashed, or empty) depending on the current state of the process (e.g., executing, blocked, and non-existent). The exact meanings ascribed

[1]This figure is adapted from [38].

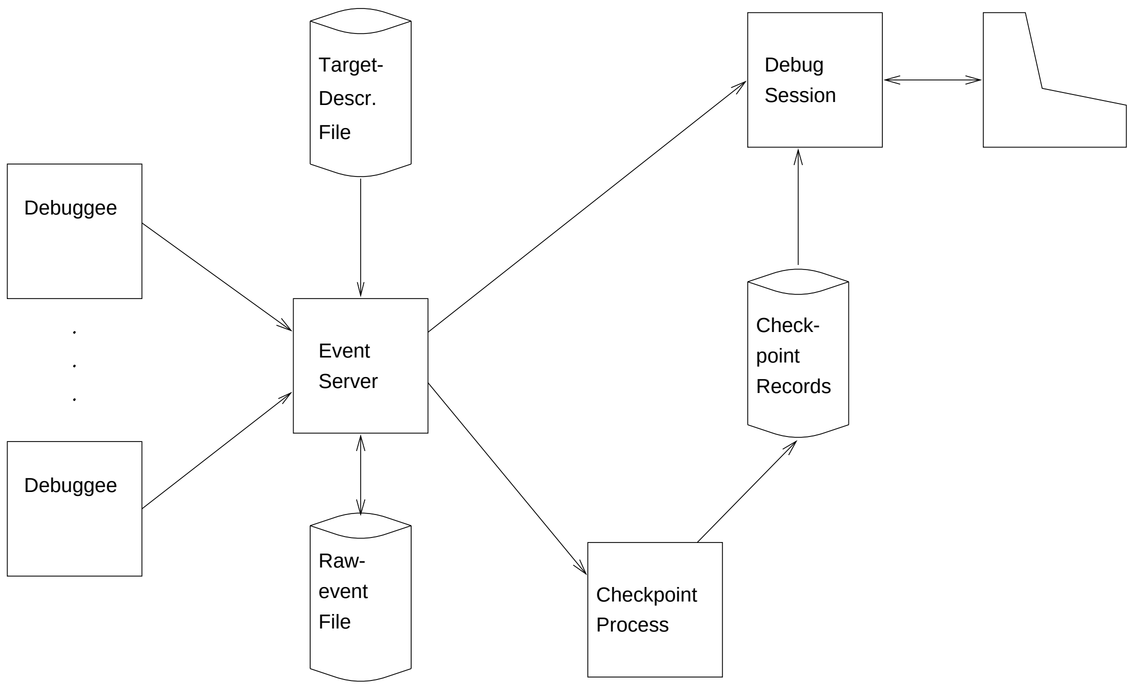


Figure 4.1: The POET architecture

to each symbol and line-style are target-specific. For each individual event, the user can pop up an information box which includes the event's type and any descriptive text.

To assist in the understanding of large event traces, processes can be clustered together (thus hiding events internal to the cluster) [39], and primitive events can be grouped together into **abstract** events, either manually or through the application of automatic pattern-detection mechanisms [24, 34]. Cluster traces are drawn on a shaded background. Abstract events are denoted by a rectangular box, which contains a filled square where it crosses each trace from which a member of the abstraction is drawn.

Example 16

> Figure 4.2 presents a sample event-trace display. The trace output depicts the behaviour of a sample μC++ AlarmClock program. Here, the *Clock* process periodically calls into the *Alarm* to indicate that another *tick* has occurred. Each *SampleUser* process, on startup, makes a request to the *Alarm* process to let it *sleep_for* some random number of seconds, at which point the *Alarm* process unblocks the *SampleUser*, which then requests to sleep for a further random interval. After its second awakening, the *SampleUser* process terminates itself. To decrease the amount of information in the display, several *SampleUser* processes are clustered together into the *Other Users* trace; notice that the intra-cluster events (in this case, the various events upon thread startup) are not displayed. Also, the *sleep_for* calls for several *SampleUser* processes inside the the *Other Users* cluster, and some of the communication between the *Clock* and *Alarm* have been grouped into two abstract events.

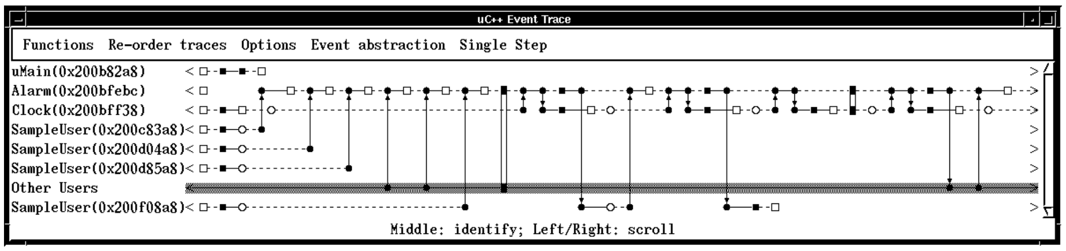


Figure 4.2: POET display with clustering and event abstraction

4.2 Implementation Notes

I have implemented an event-predicate-detection feature as an extension to the *debug-session* process of the POET system. This section provides an overview of the implementation, including some of the algorithms and the theory behind them.

4.2.1 ASCII Representation

Some of the notation[2] introduced in Section 3.3.1 does not have an obvious plain-text encoding. To avoid the need for a separate, graphical, predicate-editing tool, the meaning-equivalent representations described in Table 4.1 are introduced.

4.2.2 Parsing

An $LR(1)$ parser, based on the grammar described in Section 3.3.1, is implemented with the *lex* / *yacc* parser-generation tools [22]. As the predicate is parsed, a standard-style expression parse tree is built. Most relational operators allowed by the grammar are binary ($A \rightarrow B$, $A \parallel B$, $A \leadsto B$, $A \wr\wr B$, $a.b$, $A \wedge B$, $A \vee B$) and

[2]A comprehensive list of the notation adopted in this thesis is given on page 118.

Formal	ASCII	Meaning
$\rightarrow$	`-->`	sequential composition
$\parallel$	`||`	concurrent composition
$\leadsto$	`~~>`	alternative $\rightarrow$
$\wr$	`{}`	alternative $\parallel$
$A \overset{B}{\rightarrow} C$	`A-(B)->C`	limited $\rightarrow$
$A \overset{B}{\leadsto} C$	`A~(B)~>C`	limited $\leadsto$
$\wedge$	`AND`	conjunction
$\vee$	`OR`	disjunction

Table 4.1: ASCII equivalences for predicate notation

use the obvious binary-tree representation; an example is presented in Figure 4.3. The ternary operators ($A \overset{B}{\rightarrow} C$, $A \overset{B}{\leadsto} C$) are also stored in a binary representation; see Figure 4.4 for an example.

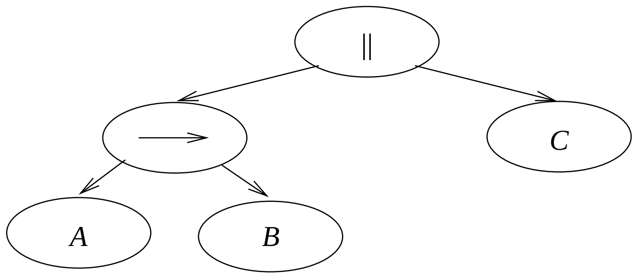


Figure 4.3: A simple parse tree, representing the expression $(A \rightarrow B) \parallel C$

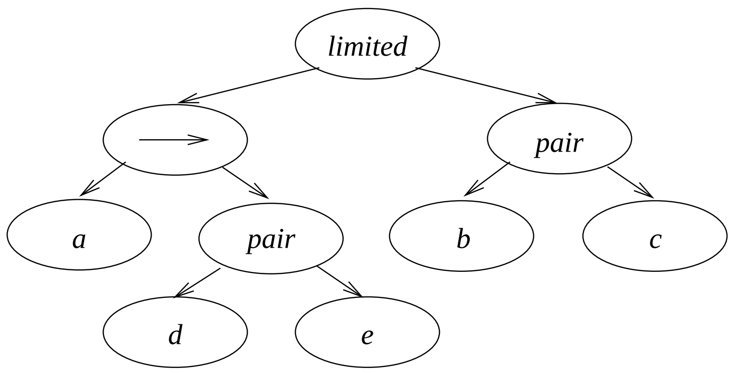


Figure 4.4: Parse tree for the expression $a \xrightarrow{b.c} d.e$

4.2.3 Searching

A depth-first search is employed, with extensions for backtracking to locate all possible predicate matches. The search procedure, SEEKNEXTMATCH(), operates recursively, starting at the root of the parse tree. Because backtracking is employed, each node may be visited multiple times in the course of the search. The usual stack-based local-variable scheme would fail to preserve search information on subsequent visits to the same node; thus, each node's local search-progress information is stored in designated fields within the parse-tree's node structure. Naturally, the algorithm's specific behaviour on each node depends upon its type.

Merging Timestamps

Before SEEKNEXTMATCH returns a match for a non-leaf node, it generates front and metron timestamps for the node, based on the front and metron timestamps of its children. The algorithm used is a direct application of the following theorem:[3]

[3]The actual implementation does not maintain back timestamps, since it is based on the modified Fidge primitive-event timestamping mechanism; back timestamps are merely mentioned

Theorem 7 (Merging Timestamps)

Given two compound events B, C, construct a new compound event $A := (B \cup C)$. Then, $\forall i \in \{1, \ldots, n\}$, the timestamps for A can be calculated as follows:

$$\begin{aligned} T_{A.front}[i] &:= \min\{T_{B.front}[i], T_{C.front}[i]\} \ ; \\ T_{A.back}[i] &:= \max\{T_{B.back}[i], T_{C.back}[i]\} \ ; \\ T_{A.metron}[i] &:= \max\{T_{B.metron}[i], T_{C.metron}[i]\} \ ; \end{aligned}$$

Proof:

The first two calculations follow directly and obviously from the definitions of their respective timestamps. The correctness of the *metron* calculation is, however, somewhat less obvious and deserves further consideration. For each component i, $1 \leq i \leq n$, of the vector timestamp, one of the following conditions must be true:

Case 1: $T_{B.metron}[i] \ = \ T_{C.metron}[i] \ = \ -1$
$\Rightarrow T_{A.metron}[i] \ := \ -1$ which is correct, since $\nexists e \in A$ s.t. $proc(A) = i$.

Case 2: $T_{B.metron}[i] \ = \ -1 \ \wedge \ T_{C.metron} \ \neq \ -1$
$\Rightarrow T_{A.metron}[i] \ := \ T_{C.metron}$ which is correct.

Case 3: $T_{B.metron}[i] \ \neq \ -1 \ \wedge \ T_{C.metron} \ = \ -1$
This follows, by symmetry, from case 2.

Case 4: $T_{B.metron}[i] \ \neq \ -1 \ \wedge \ T_{C.metron} \ \neq \ -1$
$\Rightarrow T_{B.metron}[i] = \left(\max\limits_{b \in B,\ p_b = i} T_b[i]\right) \ \wedge \ T_{C.metron}[i] = \left(\max\limits_{c \in C,\ p_c = i} T_c[i]\right)$

here for the sake of completeness.

$\Rightarrow T_{A.metron}[i] := \left(\max_{a \in (B \cup C),\ p_a = i} T_a[i] \right)$
which is the value stipulated by Definition 11. □

All leaf nodes must be single (as yet unpaired) primitive events. Recall that a primitive event is specified by a 3-tuple:

[*process-name descriptor* , *event type* , *event text*]

If any position in this specification is left blank, it is treated as a match-anything wildcard.

Algorithm 2 (Searching for a Single, Primitive Event)

```
int $p$ ;   /* process number */
int $e$ ;   /* event number */
for $p$ := 1 to NumProcesses
    if ProcessName($p$) matches descriptor
        for $e$ := FirstRelevant($node$) to LastRelevant($node$)
            if EventType($p$, $e$) matches descriptor
            $\wedge$ EventText($p$, $e$) matches descriptor
                return($p$, $e$) ;
return(NoFurtherMatches) ;
```

Note that, in the interests of efficiency, an attempt is made to restrict the primitive events under consideration to that range which is *relevant* within the context of those match candidates that have already been located, instead of blindly considering all events on each process as candidates for every leaf node. The discussion of these relevancy restrictions is deferred until Section 4.2.7.

The *process-name descriptor* may be plain text (in which case it must match the ProcessName exactly), or it may be a more complex regular expression.

Regular-expression parsing and matching operations are implemented using the `regexp` routines included with the standard C library.[4]

4.2.4 Scrolling to a Match

Once a match has been located, it is desirable to scroll the partial-order display in such a manner that as many of its component primitive events as possible are simultaneously visible to the user. In cases where the available screen space cannot accommodate the entire match at once, it is reasonable to display the earlier components of the match at the expense of those which happened thereafter. To this end, the ScrollToMatch algorithm first finds the **front** of the compound event to be displayed.

Definition 22 (Front of a compound event)

> *The* ***front*** *of a compound event A, front(A), is the set of primitive events $a \in A$ such that $\nexists b \in A : b \rightarrow a$.*

Example 17

> Consider a compound event A which contains all of the primitive events depicted in Figure 4.5. The front of that event is $\{a, d\}$, i.e., the two events which are circled in the figure.

[4]See the UNIX system manual page for `regexp(3)`, or the include file `regexp.h`, for further details.

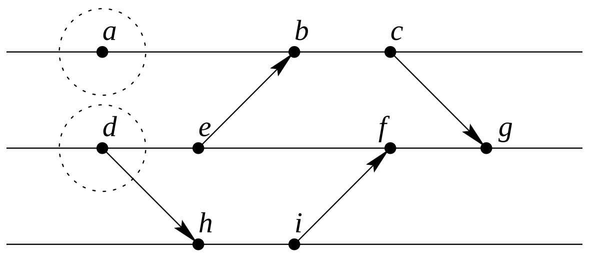


Figure 4.5: The **front** of a compound event

Algorithm 3 (Finding the front of a compound event)

```
/* Initialize the array */[5]
for $i$ := 1 to NumProcesses
        $first[i] := \perp$ ;
/* Find the first event on each trace */
for each $a \in A$
        $first[p_a] :=$ Earliest( $a$, $first[p_a]$ ) ;
/* Throw out extraneous events */
for $i$ := 1 to NumProcesses
        for $j$ := 1 to NumProcesses except $i$
                if $first[j] \rightarrow first[i]$
                        $first[i] := \perp$ ;
                        break ; /* exit inner loop */
/* front = union of all non-$\perp$ entries in first[] */
return(first) ;
```

[5]Note that $\perp$ is used to denote the 'undefined,' or 'nil' value.

The scroll algorithm is then instructed to place the events in $front(A)$ as close to the left edge[6] of the display as possible. (Synchronous communication lines and/or event abstractions may force the algorithm to display some of these events slightly away from the edge, displaced toward the centre of the display).

4.2.5 Colouring a Match

In addition to scrolling the display to render (as much as possible of) the match visible, it is also desirable to highlight the constituent events in some fashion which makes their location obvious to the user. This issue is rendered somewhat problematic by the functionality which has already been incorporated within the POET display: the event-tagging facility highlights the background behind those events which it affects, and the event decorator[7] (when employed) draws circles, squares, and various arcs thereof, surrounding events. The approach adopted here is to draw the symbols that form part of the match with their usual shape and background, but to change the foreground colour.[8]

[6]Or, if the display is being built in the vertical mode, wherein time runs from top to bottom, the top edge is used.

[7]This optional feature decorates the entry points for similar remote-procedure calls with the same addition to the event symbol. Depending on the target-environment being used, the decoration to be used may be hard-coded for that specific event type, or derived from a heuristic analysis of the surrounding events.

[8]The default *match colour*, "HotPink," is configurable through the POET resource file.

4.2.6 Process Clustering and Event Abstraction

Recall that process clustering (the combination of event information from several traces into a single display-trace unit) hides all events on the constituent traces except for those involving communication with an external trace. If all of the constituent events forming a match are hidden in this manner, then the match cannot be displayed without unclustering at least some of the traces; if this is the case, then the search algorithm assumes that the user is not interested in that match (because s/he has abstracted away all of the relevant detail) and ignores it, searching for a subsequent match. If only part of the match is hidden, then the remainder is displayed in the normal manner.

If one or more components of the match are hidden within an abstract event, then that abstract event is treated in the same manner as a primitive event by the scroll-to-match and event-colouring algorithms (the entire event is coloured in the "match" colour). It is possible for consecutive matches to be hidden within the same set of abstract events; if that is the case, then only the first such match is displayed, and repeats are rejected in the quest for a subsequent match.[9]

4.2.7 Relevancy Restrictions

As has been mentioned above (see Section 4.2.3), when searching for a match to a leaf node (i.e., a primitive-event specification), it is often desirable to limit the range of event data under consideration.

[9]It would be a good idea to report the number of rejected (hidden) matches to the user; this enhancement may be implemented at some point in the future.

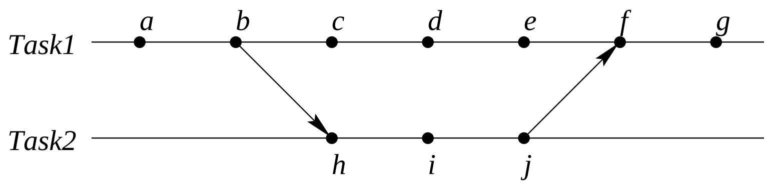


Figure 4.6: Relevancy example

Example 18

Consider the process-trace diagram in Figure 4.6, and the predicate

$$[\ \textit{Task2}\ ,\ \alpha\ ,\ ]\ \|\ [\ \textit{Task1}\ ,\ \beta\ ,\ ]$$

and let event i be of type α. Then, once the search algorithm has identified event i as a match to the left child of the ($\|$) node, it embarks on a search for a match to the right child, i.e., an event of type β in task *Task1*. The **relevant** events are $\{c, d, e\}$, because no others in *Task1* could possibly be concurrent with the candidate (event i) which has already been identified.

The current system performs relevancy restrictions for those operators that rely on the recommended precedence relation, i.e., $\rightarrow$ (unlimited and limited) and $\|$. (The existential quantifier involved in the definition of the alternative precedence relationship, used by $\leadsto$ and $\wr$, makes the determination of search-space restrictions problematic and limits their possible effectiveness.) The principles that form the basis of the search restrictions are listed in Table 4.2; each restriction is helpful in determining either the first relevant event (type $\vdash$) or the last ($\dashv$).

The relevancy-restriction feature could be implemented by employing a straightforward binary-search algorithm to enforce each of the conditions in Table 4.2. However, because the POET system uses modified Fidge timestamps, certain optimizations are possible.

Operator	Event	Type	Restriction	Timestamp Comparison
$A \rightarrow B$	$b \in B$	$\vdash$	$A \rightarrow b$	$T_{A.metron} \overset{act}{<} T_b$
$A \overset{C}{\rightarrow} B$	$b \in B$	$\vdash$	$A \rightarrow b$	$T_{A.metron} \overset{act}{<} T_b$
	$c \in C$	$\vdash$	$A \rightarrow c$	$T_{A.metron} \overset{act}{<} T_c$
	$c \in C$	$\dashv$	$c \rightarrow B$	$T_c[p_c] < T_{B.front}[p_c]$
$A \parallel B$	$b \in B$	$\vdash$	$\neg(b \rightarrow A)$	$\neg(T_b[p_b] < T_{A.front}[p_b])$
	$b \in B$	$\dashv$	$\neg(A \rightarrow b)$	$\neg(T_{A.metron} \overset{act}{<} T_b)$

Table 4.2: The basis for relevancy restrictions

Theorem 8

The index of an event a on process p_a is equal to $(T_a[p_a] - 1)$.

The actual algorithms to locate the first and last relevant events are outlined below. Each BinSearch operation performs a standard binary search on the current process within the specified range, and LAST returns the last event on the specified process.

Algorithm 4

```
function FirstRelevant
        switch(restriction type)
        case $\rightarrow$:
                if $p_b \in active(A)$
                        $range := T_{A.metron}[p_b] \ldots \text{LAST}(p_b)$ ;
                else
                        $range := 0 \ldots \text{LAST}(p_b)$;
                return BinSearch($range$ for first $b$ s.t. $T_{A.metron} \overset{act}{<} T_b$) ;
        case LIMITED:
                /* B was taken care of by the case above, */
                /* so I only worry about the limiter, C */
                if $p_c \in active(A)$
                        $range := T_{A.metron}[p_c] \ldots (T_{B.front}[p_c] - 1)$ ;
                else
                        $range := 0 \ldots (T_{B.front}[p_c] - 1)$ ;
                return BinSearch($range$ for first $c$ s.t. $T_{A.metron} \overset{act}{<} T_c$) ;
        case $\|$:
                return $(T_{A.front}[p_b] - 1)$ ;
        default:
                return 0 ;
        end switch
```

```
function LASTRELEVANT($node$)
    switch($restriction\ type$)
    case LIMITED:
        return $(T_{B.front}[p_c] - 1)$ ;
    case $\|$:
        if $p_b \in active(A)$
            $range := (T_{A.front}[p_b] - 1) \ldots (T_{A.metron}[p_b])$ ;
        else
            $range := (T_{A.front}[p_b] - 1) \ldots$ LAST$(p_b)$ ;
        return BINSEARCH($range$ for first $b$ s.t. $\neg T_{A.metron} \overset{act}{<} T_b$) ;
    default:
        return LAST$(p_b)$ ;
    end switch
```

4.3 A Sample Session

Recall the alarm-clock application discussed in Example 16. This section traces through the steps required to answer the following question:

Does the Alarm process ever receive three or more ticks from the Clock process without awakening a SampleUser process in-between ticks?

In my formal notation, this would be expressed as follows:

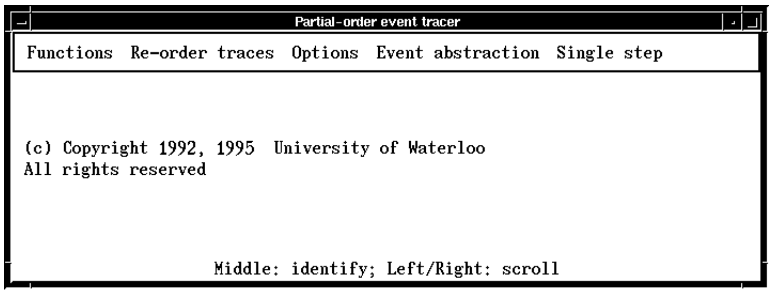
$$A := [\text{Clock}(.*), \text{thread enter, tick}]\ .\ [\text{Alarm}(.*), \text{thread received}, .*]\ ;$$
$$B := [\text{Alarm}(.*), \text{thread leave, sleep\_}.*]\ ;$$
$$C := A \vee B\ ;$$
$$A \overset{C}{\rightarrow} A \overset{C}{\rightarrow} A$$

1. The user launches the POET environment, and sees the startup screen shown in Figure 4.7.
2. Selecting the *Run program* option from the *Functions* menu (Figure 4.8), the user enters the command-line for the compiled μC++ application.
3. The POET system identifies the target environment, collects event-trace information, and displays a process-time diagram for the execution (Figure 4.9).
4. Examining events in the trace (and, with the middle mouse button, identifying the event types and accompanying text, where appropriate), the user constructs a predicate specification with a text editor:

```
[Clock(.*),thread enter,tick].[Alarm(.*),thread received,]

-(
[Clock(.*),thread enter,tick].[Alarm(.*),thread received,]
OR
[Alarm(.*),thread leave,sleep_.*]
)->

[Clock(.*),thread enter,tick].[Alarm(.*),thread received,]

-(
[Clock(.*),thread enter,tick].[Alarm(.*),thread received,]
OR
[Alarm(.*),thread leave,sleep_.*]
)->

[Clock(.*),thread enter,tick].[Alarm(.*),thread received,]
```

5. Having saved the above specification as `3ticks.pred`, the user selects the *Find predicate* option from the *Functions* menu, and chooses this specification (Figure 4.10).

6. The search algorithm locates an occurrence of the predicate, scrolls the display to reveal it, and then colours the components of the match appropriately. (Because of the constraints of black-and-white reproduction, the match colouring has been rendered as a light grey in Figure 4.11.) Thus, the user's initial question is answered in the affirmative.



```
(c) Copyright 1992, 1995  University of Waterloo
All rights reserved

Middle: identify; Left/Right: scroll
```

Figure 4.7: The POET display on startup

7. The user selects *Find again* to discover whether there is another situation in the event trace that also satisfies this predicate. POET locates a second match and displays it in the same manner.

8. Having selected *Find again* a second time, the user is informed that there is no third match (Figure 4.12).

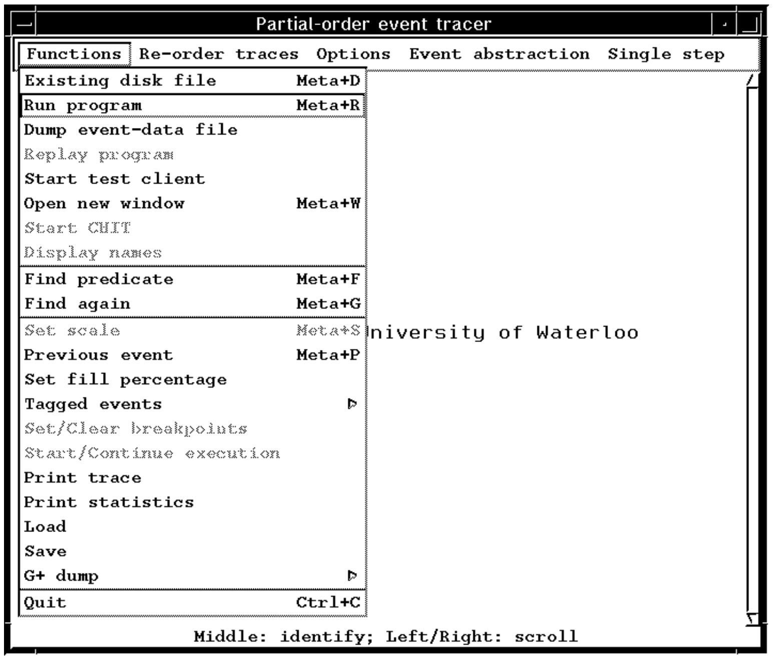


Figure 4.8: The *Functions* menu

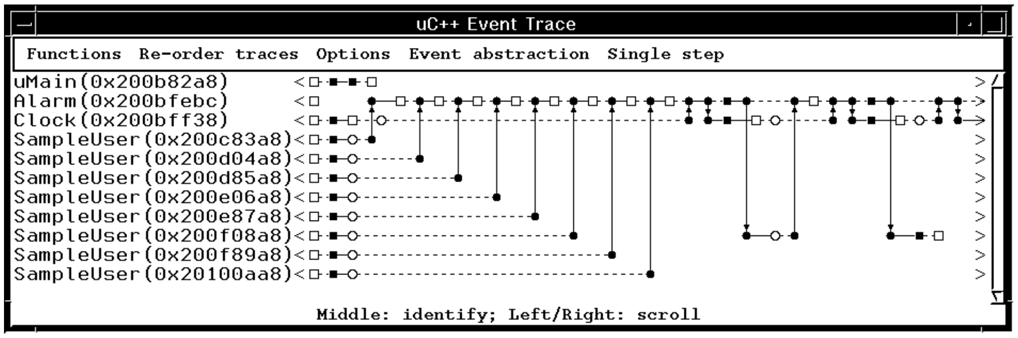
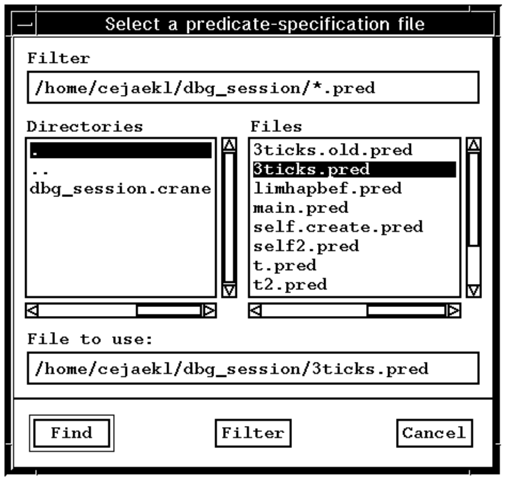


Figure 4.9: Execution trace for the AlarmClock program



```
/home/cejaekl/dbg_session/*.pred
```

Directories:

```
.
..
dbg_session.crane
```

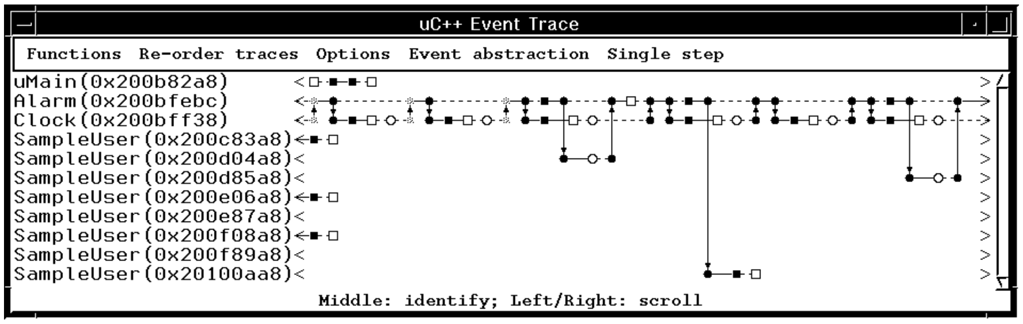
Files:

```
3ticks.old.pred
3ticks.pred
limhapbef.pred
main.pred
self.create.pred
self2.pred
t.pred
t2.pred
```

File to use:

```
/home/cejaekl/dbg_session/3ticks.pred
```

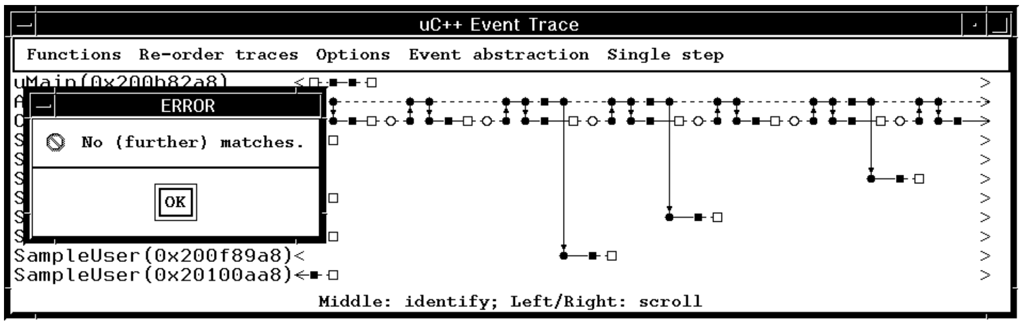
Figure 4.10: Selecting a predicate-specification file



```
uMain(0x200b82a8)
Alarm(0x200bfebc)
Clock(0x200bff38)
SampleUser(0x200c83a8)
SampleUser(0x200d04a8)
SampleUser(0x200d85a8)
SampleUser(0x200e06a8)
SampleUser(0x200e87a8)
SampleUser(0x200f08a8)
SampleUser(0x200f89a8)
SampleUser(0x20100aa8)
```

Middle: identify; Left/Right: scroll

Figure 4.11: The first match



No (further) matches.

Figure 4.12: After the last match

Chapter 5

Performance

As is the case with any tool designed to search through a large amount of data, the time and storage space required to seek a match are serious concerns. This chapter examines the factors that affect performance and some techniques which have been, or could be, applied to improve the speed of searches.

5.1 Space Used

The search facility, as it is currently implemented, requires reasonably little memory.[1] As the pattern-specification file is parsed, a tree is built wherein each node consists of the 3706-byte[2] structure detailed in Tables 5.1 and 5.2. There is a cer-

[1]The POET system uses a significant amount of secondary-storage space to record the event-trace information. A discussion of the trade-offs and rationales for the actual event-trace format is beyond the scope of this thesis; the interested reader may wish to consult [24] and/or the POET source code.

[2]Some compilers will add an extra 2 bytes of padding, extending the structure to the next 4-byte boundary.

tain amount of space wasted by this system, which could be recovered in return for some additional code complexity and execution time:

- *local.trace*, *local.event*, and *local.tse* are only used in leaf nodes; if this memory were dynamically allocated, 1248 bytes could be replaced with a 4-byte pointer.
- In leaf nodes, *local.front* = *local.metron* = *local.tse.ts*. Some of these 1200-byte vector timestamps could be reconstructed from a single source when required.
- *local.trace* = *local.tse.event.e_etrace* and *local.event* = *local.tse.event.e_evcnt*. Eight bytes could be saved here; there might be a minor increase in code and time because of additional pointer-dereferencing.
- The vector timestamps currently assume a (maximal) 300 processes in the execution trace.[3] These could be dynamically allocated.
- When a regular expression (instead of a simple text string) is encountered, its compiled version is allocated a 200-byte buffer; some attempt could be made to tailor the size to that actually required by the expression.
- *num_traces* is included only for the sake of convenience; it could be passed between functions as an additional parameter, saving 4 bytes per node.

I have chosen, however, not to incur the (often minor) performance penalties associated with the above refinements, because the memory requirements to process a reasonable predicate are already low enough to pose no practical problems.

[3]This static limit is governed by a compile-time constant.

Name		Bytes	Purpose
ntype		4	type of node ($\|$, $\rightarrow$, etc.)
left, right		2×4	pointers to left, right children
pat_proc		4	pointer to textual pattern for *process*
pat_type		4	pointer to textual pattern for *event type*
pat_text		4	pointer to textual pattern for *event text*
regex_proc		4	ptr. to compiled regexp for *process*
regex_type		4	ptr. to compiled regexp for *event type*
regex_text		4	ptr. to compiled regexp for *event text*
local	in_progress	2	search on this node currently incomplete?
	front	300×4	front timestamp for this (sub)tree
	metron	300×4	metron timestamp for this (sub)tree
	trace	4	process index (used only for a leaf node)
	event	4	event index (used only for a leaf node)
	tse	1240	TS_EVENT (timestamped event struct.)
restrict	first_type	4	type of restriction for first relevant event
	last_type	4	type of restriction for last relevant event
	first, last	2×4	pointers to timestamps for restrictions
num_traces		4	number of processes when search started
Total		3706	

Table 5.1: Contents of a parse-tree node

Name		Bytes	Purpose
event	e_etype	2	event type
	e_flag	2	miscellaneous flags
	e_trace	4	source trace (process)
	e_ptrace	4	partner's trace (if applicable)
	e_evcnt	4	source event index (position within process)
	e_sevcnt	4	partner's event index (if applicable)
	abstr_base	4	base abstraction level in use
	abstr_cur	8×1	current abstraction level for 8 possible clients
	e_rtime	8	"real" time when event occurred
ts		300×4	vector timestamp
Total		1240	

Table 5.2: Contents of a TS_EVENT structure

Example 19

> Consider the fairly complex predicate discussed in Section 4.3. This produces a 23-node parse tree, so the basic node structures occupy $23 \times 3706 = 85238$ bytes. Including terminating *null* characters, the textual patterns for the 12 leaf nodes occupy a total of $(5 \times (28 + 27)) + (2 \times 32) = 339$ bytes. There are a total of 14 regular expressions (1 for each *process* descriptor, and 2 in *event_text* descriptors); their compiled versions are stored in buffers which total $14 \times 200 = 2800$ bytes. The two $\rightarrow$ operators enforce restrictions on the *last* relevant event for 6 nodes; the *restrict.last* timestamps are dynamically allocated and occupy $11 \times 4 = 44$ bytes each, for a total of $6 \times 44 = 264$ bytes. There are also two-sided restrictions (imposed on the limiting terms) for 10 nodes; these total $10 \times (2 \times 44) = 880$ bytes.
>
> Thus, the total amount of memory requested for the complete parse tree is
>
> $$85238 + 339 + 2800 + 264 + 880 = 89521$$
>
> bytes, or roughly 87 KB. Of course, the granularity (and overhead) of memory allocation doubtless causes the actual amount of memory used to be slightly higher.

In the course of the search, some stack space is also required (as the parse tree is recursively descended), and some additional temporary storage is allocated from the heap. However, the amount of stack and heap space required to perform a search is also quite small.

Generalization

Thus, as a rough approximation, one may consider the amount of memory required to search for a predicate to be $4n$ KB, where n is the number of nodes in the predicate's parse tree. The space requirement scales linearly as the size of the predicate increases, and is independent of the amount of event-trace data to be examined; also, the memory required is insignificant compared to the other space requirements of any event-tracing system.

5.2 Time Used

Unlike its space requirements, the search algorithm's running time may prove to be problematic.

As mentioned in Section 3.1.2, it is impossible at this point to accurately predict the size, architecture, and complexity of the common distributed systems which programmers will eventually need to debug; it is, likewise, unclear how the complexity of these systems will correlate with the speed and number of processors available for debugging them. Thus, it is difficult to guess at the threshold which will separate acceptable performance from that which will render a search tool too slow to be usable.

Bearing that caveat in mind, this section presents a general outline of the factors which appear to influence search times.

5.2.1 The Overriding Performance Factor

It is inherent in the nature of the problem that the event trace data, because of its size, needs to reside in secondary storage, whereas the expression being searched for, and progress/result information, is small enough to fit in primary memory. This is the case in the POET implementation. Thus, it seems reasonable to expect that:

Conjecture 1

The limiting factor in a predicate search's execution is the time required to fetch primitive events from secondary storage for examination.

Example 20

Here, a simple investigation is performed using a relatively short execution trace. Because all components of the POET system and the application being examined are run on the same machine, no data is transmitted between processors; because the event-trace file is small, it effectively resides in the operating system's disk cache and no significant amount of time is spent accessing the disk. Under these "ideal" conditions, the vast majority of the time spent on event retrievals is caused by the processing overhead associated with inter-process communication. Thus, in this situation, the amount of compute time consumed is an accurate indicator of the real (elapsed) time.[4]

Consider, yet again, the sample search session detailed in Section 4.3; this event trace contains 323 events, distributed as shown in Table 5.3. In order to ensure that the majority of the time spent by the POET system is a direct result of searching for a predicate, steps 5 through 8 are repeated twenty times before the *quit* option is selected. As can be seen in Figure 5.2, the amount of processing time spent by the POET system on startup is insignificant compared with the time spent searching for the predicate. (The `poet` process is the *event server.*)

[4]Even if the event-trace file is much larger, the difference between compute time and elapsed time remains quite small; see Section 5.2.2 and, in particular, Figure 5.3. Compute time is used here because the nature of the user interface, and the small size of the event-trace file, make it quite difficult to obtain an accurate measurement of the elapsed time.

Trace Name	Number of Events
uMain(0x200b82a8)	4
Alarm(0x200bfebc)	114
Clock(0x200bff38)	125
SampleUser(0x200c83a8)	10
SampleUser(0x200d04a8)	10
SampleUser(0x200d85a8)	10
SampleUser(0x200e06a8)	10
SampleUser(0x200e87a8)	10
SampleUser(0x200f08a8)	10
SampleUser(0x200f89a8)	10
SampleUser(0x20100aa8)	10
Total	323

Table 5.3: AlarmClock program: traces and events

The interesting portion of the *debug session* process' function-call profile,[5] generated with `prof(1)`, is depicted in Figure 5.1. Of the functions listed, `get_ts_event`, `get_tse`, `do_get_event`, and `mf_get_event` are all directly involved in the fetching of primitive events from the *disk server* and timestamping them. This means that more than

$$3.76 + 1.62 + 1.08 + 0.68 = 7.14 \text{ seconds}$$

or $\frac{7.14}{29.25} \approx 24\%$ of the *debug session*'s processing time is spent fetching primitive events.

In the same profile, `merge_timestamps`, `advance`, and `step` are readily identifiable as part of the in-memory manipulation of the parse tree's contents. As well, the majority of the `strcmp` calls are related to this same manipulation. Thus,

$$(4.64 + 0.76 + 0.60) + (0.77 - \varepsilon) < 6.77 \text{ seconds}$$

or roughly $\frac{6.77}{29.25} \approx 23\%$ of the *debug session*'s time is directly attributable to manipulating in-memory structures during the search process.

Of course, this is not the full story—as is clear from Figure 5.2, the *event server* (the `poet` process) spends almost as much time servicing requests for

[5]There are several irregularities in the data reported in Figure 5.1, including duplicate entries for the `__mcount` function (which is part of the profiling mechanism), a total compute time (29.25 seconds) which does not match that reported by `ps`, and a disproportionately large slice of time credited to `collect_exit`, `mf_rename_trace` and `disk_init` (these are short functions, called once or not at all, which should definitely not be using the amount of time attributed to them). These appear to be the result of bugs in the AIX profiling-tool suite. It is this author's belief that the execution times attributed to `merge_timestamps`, `get_ts_event`, `get_tse`, `do_get_event`, `strcmp`, `advance` and `step` are, nevertheless, reasonably accurate reflections of reality.

```
Name                  %Time    Seconds   Cumsecs  #Calls  msec/call
.merge_timestamps      15.9       4.64      4.64   31800     0.1459
.get_ts_event          12.9       3.76      8.40  133894     0.0281
.__mcount              11.1       3.24     11.64
._moveeq                9.6       2.80     14.44
.get_tse                5.5       1.62     16.06  160161     0.0101
.__mcount               5.0       1.46     17.52
.do_get_event           3.7       1.08     18.60  232479     0.0046
.update_found_displa    3.2       0.95     19.55     161     5.90
.disclaim_free_y        2.8       0.82     20.37
.strcmp                 2.7       0.78     21.15
.advance                2.6       0.76     21.91  151700     0.0050
.mf_get_event           2.3       0.68     22.59  232490     0.0029
.step                   2.1       0.60     23.19   81960     0.0073
.collect_exit           2.0       0.59     23.78       1   590.
.free_y                 1.8       0.53     24.31   68204     0.0078
.mf_rename_trace        1.0       0.28     24.59
.malloc_y               0.9       0.25     24.84   76341     0.0033
.disk_init              0.8       0.23     25.07       1   230.
.write                  0.7       0.20     25.27   61800     0.0032
[...]
.redraw                 0.0       0.00     29.25      87     0.0
```

Figure 5.1: (Partial) output from 'prof dbg_session'

After Startup:

```
cejaekl  7006 29530   0 20:59:00 pts/18  0:00 ts_chkpt
cejaekl 29530 33448   0 20:58:58 pts/18  0:00 poet
cejaekl 31068 29530   0 20:59:00 pts/18  0:01 dbg_session
```

After Twenty Complete Searches:

```
cejaekl  7006 29530   0 20:59:00 pts/18  0:00 ts_chkpt
cejaekl 29530 33448   2 20:58:58 pts/18  0:41 poet
cejaekl 31068 29530   3 20:59:00 pts/18  0:50 dbg_session
```

Figure 5.2: (Partial) output from the 'ps' command

primitive events as the *debug session* does on its entire search process. Additionally, if these two POET components are running on different machines, then there could also be significant communication delays which would slow down requests to retrieve primitive events.

The time required to process a predicate search appears, indeed, to be overwhelmingly influenced by the time required to fetch primitive events from secondary storage. This result is compatible with Conjecture 1.

In the future, the amount of data that the *event server* stores per event will probably be reduced (perhaps from the current 40 bytes to 20). This would improve performance somewhat, because a greater number of primitive events could be fetched in a single (block) request; it is, nonetheless, likely that the time spent fetching events will remain the dominant performance factor.

5.2.2 Test Runs

As discussed above, one would expect the search time to be proportional to the number of events which are fetched from secondary storage.

In reality, the situation is more complex. In order to reduce the number of (expensive) event requests that must be issued, it is common to (and POET does) fetch events from secondary storage in blocks, returning a group of events, in sequence, on the requested trace. This optimization means that events can be retrieved more quickly if they are requested in sequence, but there is a significant performance decrease when random-access requests are made. As well, requests for timestamps can have a large effect on the time required to complete a search. Because of the expense involved in calculating timestamps, POET makes a substantial effort to cache and re-use them; however, if too many timestamps (or even a few timestamps for events which are widely separated) are requested, then the time required to compute the timestamps affects substantially the running time of the search.

The following test runs provide empirical evidence which supports this analysis.

Predicates

The seven test predicates are listed in Table 5.6. The first two locate what happens to be the last event recorded for the *event server* and *checkpoint* processes respectively. Predicate III scans the traces for both *debug session* processes, one after the other, and locates the process termination events which are the last events recorded for both processes.

Predicate IV attempts to find an *exit* event in the *event server* process which is concurrent with a *start* event in the *checkpoint* process; no such combination exists, so the search must fail.

Number	Predicate
I	[poet,exit,]
II	[ts_chkpt,exit,]
III	[dbg_session,exit,]
IV	[poet,exit,] ‖ [ts_chkpt,start,]
V	[poet,create,] ‖ [dbg_session,exit,]
VI	[dbg_session,exit,] ‖ [poet,create,]
VII	[dbg_session,exit,] ‖ [ts_chkpt,wait for events,]

Table 5.4: Predicates used for performance analysis

There are three *create* events in the *event server* process, each of which corresponds to the spawning of one of the other three processes. None of these is concurrent with the *exit* event for either *debug session* process, so Predicates V and VI are both unsatisfiable.

Slightly less than half of the events recorded for the *checkpoint* process are of type *wait for events*. None of these, however, are concurrent with either *debug session*'s *exit* event. Thus, while the second component of Predicate VII has many matches, the complete predicate cannot be matched.

Methodology

All of these tests were performed on an otherwise unloaded 66 MHz IBM RS/6000 250, based on a self-debug event file[6] which contains 175187 events divided among four processes as shown in Table 5.5. Note that each process begins with an event

[6]One instance of POET can be used to examine another instance's behaviour. The `poet` process described here is the event server.

Trace Name	Number of Events
poet	87593
dbg_session	7108
ts_chkpt	79598
dbg_session	888

Table 5.5: Test data: traces and events

of type *start*, and terminates with an event of type *exit*.

After the startup of POET and the loading of the event file, the *checkpoint* process was allowed to timestamp all of the events and save checkpoints for later use by the *debug session*'s timestamping algorithm; this required 20 CPU seconds. Then, a search was performed for the first predicate and the relevant statistical information recorded. The search was repeated a further four times; the values reported below are averages over five trials.[7] In order to provide a clearer view of the algorithm's behaviour, each search was examined with the relevancy restriction feature (see Section 4.2.7) disabled, and again with that feature enabled.

Miscellaneous Overhead

Table 5.6 lists the observed CPU time used by the POET system and the corresponding real elapsed time (all figures are in seconds, and represent the average over five trials); the first set of data is the result of searches without relevancy restric-

[7]Almost no variation was observed among searches for the same predicate, with the exception of an occasional and minor improvement in the number of timestamps retrieved from caches on the final four iterations.

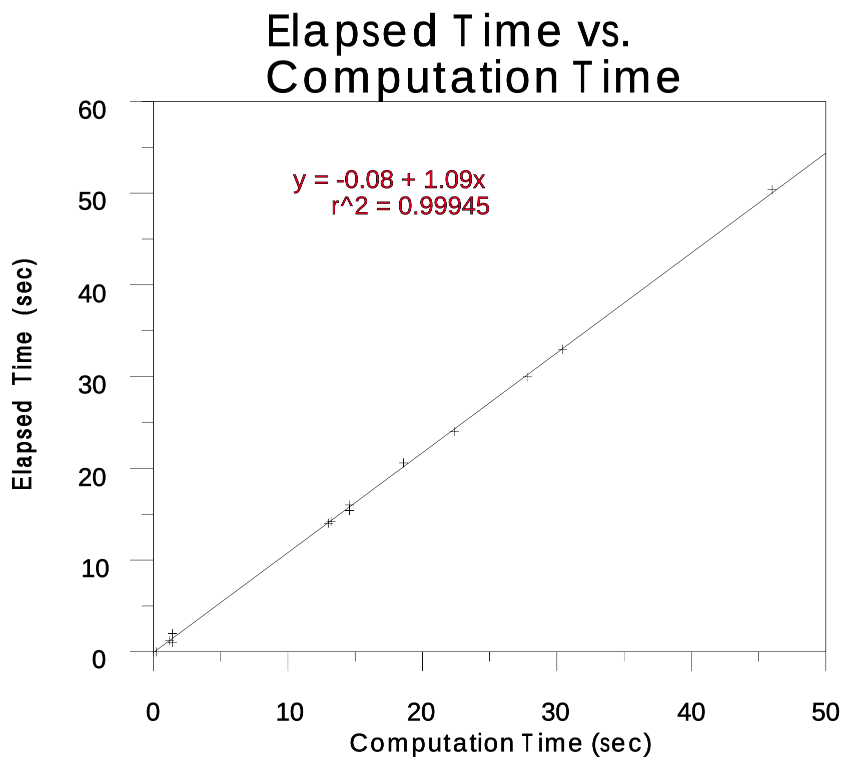


Figure 5.3: Observed execution times correlate well with CPU seconds

tions, and the second set reflects the same searches with these restrictions enabled.[8] As the graph and linear-regression curve-fit in Figure 5.3 make clear, the overhead involved in waiting for inter-process communication and other operating-system tasks is nearly constant and roughly 9%. The good correlation ($r^2 \approx 0.99945$) provides some measure of assurance that outside influences are constant between searches for different predicates, and may be safely excluded from further consideration in the analysis of these test runs.

[8]The reasons for the difference between these two sets of numbers are discussed below.

(Relevancy Restrictions Disabled)

Predicate	Compute time (*event server*)	Compute time (*debug session*)	Compute time (total)	Elapsed time
I	2.0	12.6	14.6	15.4
II	1.6	11.4	13.0	14.0
IIIa	0.2	1.2	1.4	1.0
IIIb	0.0	0.0	0.0	0.0
IV	3.6	24.2	27.8	30.0
V	2.6	16.0	18.6	20.6
VI	4.2	26.2	30.4	33.0
VII	7.8	38.2	46.0	50.4

(Relevancy Restrictions Enabled)

Predicate	Compute time (*event server*)	Compute time (*debug session*)	Compute time (total)	Elapsed time
I	1.8	12.8	14.6	15.4
II	1.8	11.4	13.2	14.2
IIIa	0.2	1.0	1.2	1.2
IIIb	0.0	0.2	0.2	0.0
IV	1.8	12.8	14.6	16.0
V	4.0	18.4	22.4	24.0
VI	0.4	1.0	1.4	2.0
VII	0.2	1.2	1.4	2.0

Table 5.6: CPU seconds used and real time elapsed

Performance Data

Tables 5.7 and 5.8 list the observed statistical information with relevancy restrictions disabled and enabled, respectively. Each table contains two rows for Predicate III, one for the time to find the first match, and another for the time required to continue the search until the second match is located.

Generating a timestamp can be a time-consuming operation, involving, in the worst case, the retrieval of checkpoint information and a significant number of primitive events that are causally between the checkpoint and the event whose timestamp is desired. The *debug session* process maintains a large, multi-level cache in an attempt to avoid as many timestamp calculations as possible. Thus, the *Timestamp requests* column indicates the number of primitive events whose timestamps were requested,[9] whereas the *Timestamp calculations* column indicates only the number of such requests that could not be satisfied from the cache.

The number of requests to fetch a primitive event is listed in the *Event requests* column. For Predicates I, II and III, one would expect this to be the number of primitive events in the appropriate process' trace, because the search algorithm scans each trace from beginning to end, and only stops on a match at the very end of the trace. Predicates IV, V, VI and VII all involve concurrent composition, and all fail to match any set of events within the execution trace; in the absence of relevancy restrictions, the search algorithm examines every event in all processes appropriate for the first component and then, for every match thereto, examines every event in all processes appropriate for the second component. Thus, for example, the search

[9]Some of these requests occur during the display algorithm's "update" of the process-trace diagram which follows a search, successful or otherwise; the actual number of timestamps requested by the search algorithm is less than or equal to this number.

Predicate	Timestamp requests	Timestamp calculations	Event requests	Expected event req.s	Compute time (sec)
I	13.0	0.4	87603.6	87593	14.6
II	13.0	0.0	79606.0	79598	13.0
IIIa	16.2	0.0	7118.4	7108	1.4
IIIb	17.0	0.0	899.8	888	0.0
IV	2.0	0.0	167192.0	167191	27.8
V	9.0	0.2	111625.6	111581	18.6
VI	8.0	0.0	183186.0	183182	30.4
VII	79260.0	79252.2	591580.0	167192	46.0

Table 5.7: Performance with relevancy restrictions disabled

Predicate	Timestamp requests	Timestamp calculations	Event requests	Expected event req.s	Compute time (sec)
I	13.0	0.2	87603.4	87593	14.6
II	13.0	0.0	79606.0	79598	13.2
IIIa	16.2	0.0	7118.4	7108	1.2
IIIb	17.0	0.0	899.8	888	0.2
IV	2.0	0.0	87594.0	$\leq$ 167191	14.6
V	66.0	32.0	172721.0	$\leq$ 111581	22.4
VI	5.0	0.0	8011.0	$\leq$ 183182	1.4
VII	4.0	0.0	8000.0	$\leq$ 167192	1.4

Table 5.8: Performance with relevancy restrictions enabled

for Predicate V should generate

$$87593 + 3(7108 + 888) = 111581 \text{ event requests.}[10]$$

The introduction of relevancy restrictions should reduce the number of events examined. When the number of events requested is significantly above expectations, the difference is caused by the actions of the timestamping algorithm.

Conjecture 2

> *The time required to perform a search is proportional to the number of primitive events examined.*

Is this conjecture a valid reflection of the algorithm's performance? The graph in Figure 5.4 (which ignores Predicate VII) suggests that it is, and that the time required to complete a search is roughly 0.18 seconds per 1000 events examined. However, none of the searches included in this graph requires any significant calculation of timestamps. The algorithm requests only one timestamp (that of the primitive event which is matched) for Predicates I, II, IIIa and IIIb, two timestamps (one for each of the unique matches to the components) for Predicate IV, and four timestamps for Predicates V and VI. In the above cases, the multi-level caching which the POET system employs is virtually able to eliminate the need to calculate timestamps from scratch.

Now, consider the graph in Figure 5.5, which reflects the entire contents of Table 5.7. The result for Predicate VII is obviously not on the line of best fit from Figure 5.4. The factor that differentiates this case from the others is the number of timestamps that must be calculated; because the component

[10] Recall that there are three *create* events in the *poet* trace. Note that reordering the terms in a predicate can affect the search time (this will be discussed in Section 5.3, *q.v.*).

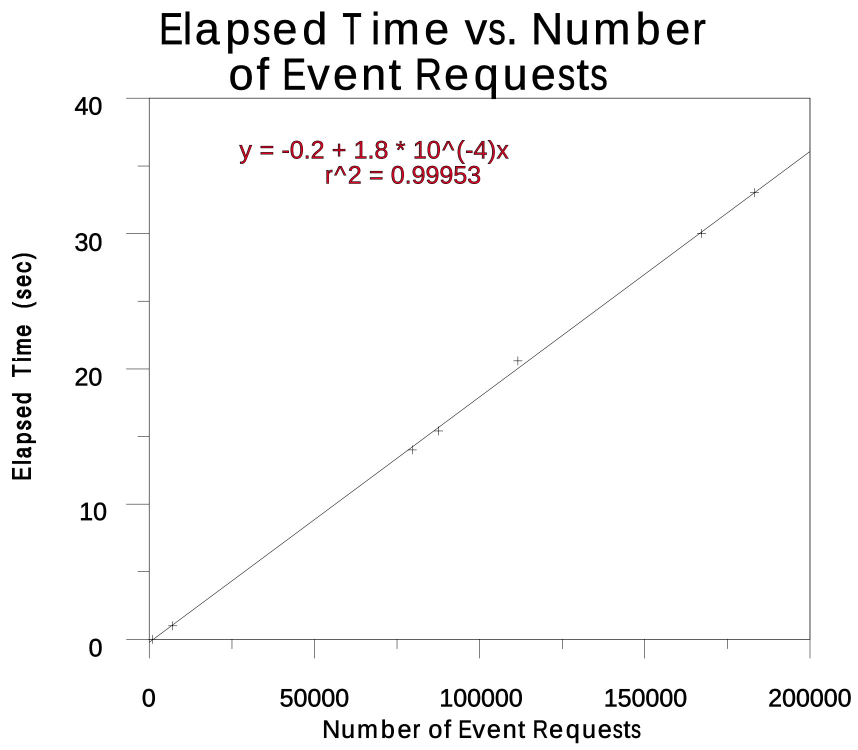


Figure 5.4: Predicates I through VI, relevancy restrictions disabled

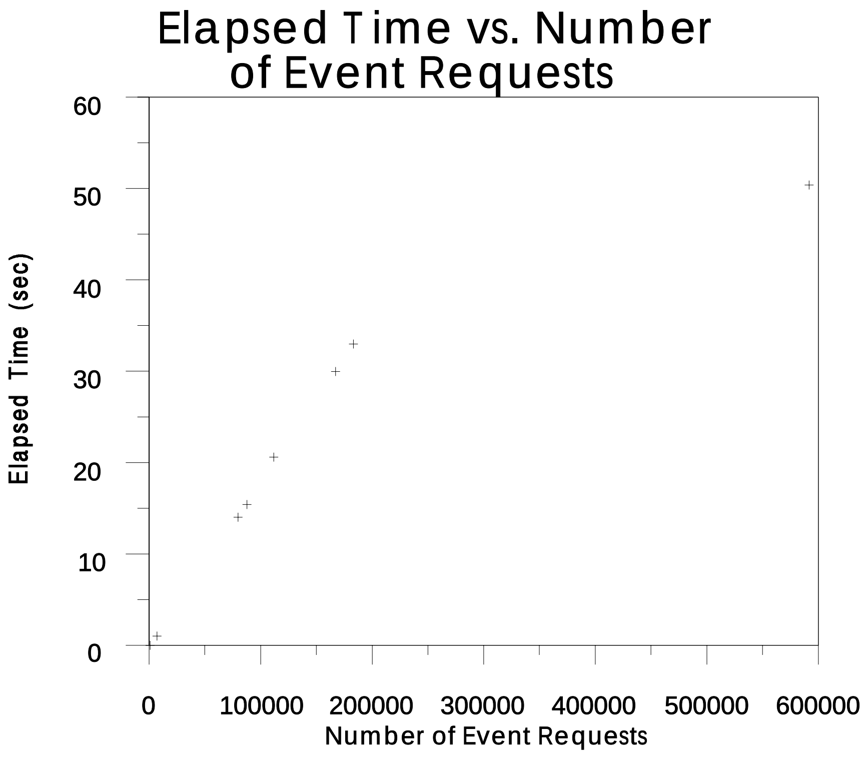


Figure 5.5: Predicates I through VII, relevancy restrictions disabled

[ts_chkpt,wait for events,]

matches many events, many timestamps are requested, and the resulting timestamp calculations, in turn, examine a significant number of primitive events. Perhaps because each timestamp calculation requests events that are in close proximity to one another, the bulk processing and caching features of the event-retrieval mechanism are able to provide a certain performance improvement.

Finally, consider what happens when the relevancy-restriction feature is enabled (see Figure 5.6). For the simple predicates which involve only a single event (I, II and III), the search algorithm operates in essentially the same manner, and performance is similar. For the more complex predicates which involve a concurrent

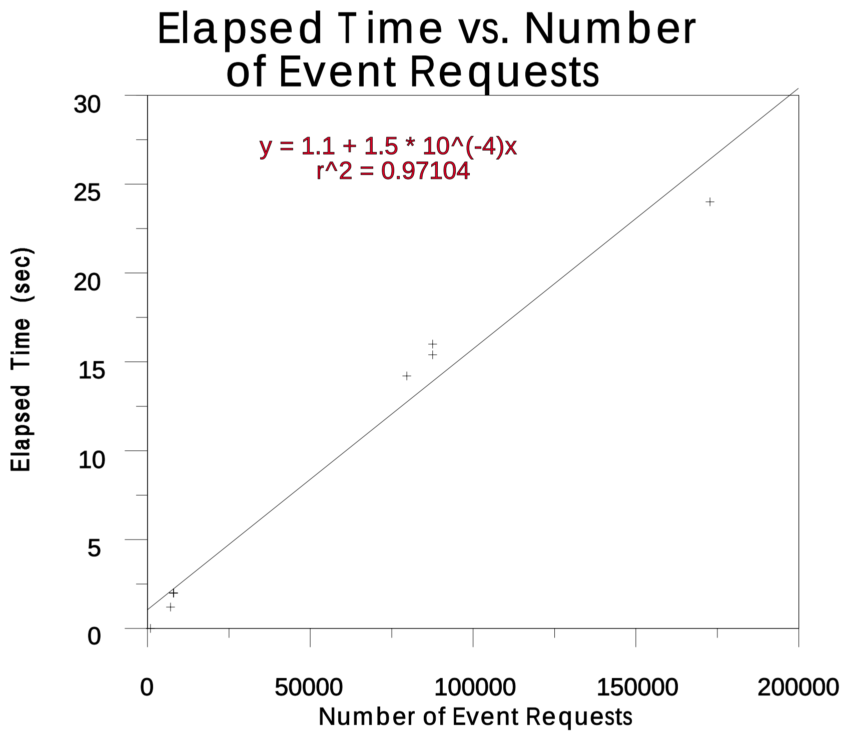


Figure 5.6: Predicates I through VII, relevancy restrictions enabled

composition of events (IV, V, VI and VII), there is a reduction in the number of primitive events that the search algorithm requests in its attempts to match the second component of the predicate. However, because the relevancy-restriction feature may perform a binary search which requires timestamps from vastly disparate regions of the event trace, the timestamp-caching mechanism can be largely circumvented, and the resulting timestamp calculations may precipitate a huge increase in the number of primitive events actually examined. Thus, while the relevancy restrictions can result in a speedup (e.g., Predicates IV, VI, and VII), they can also result in an overall slowdown of the search, (e.g., Predicate V).

As is the case without relevancy restrictions, the search which involves a sub-

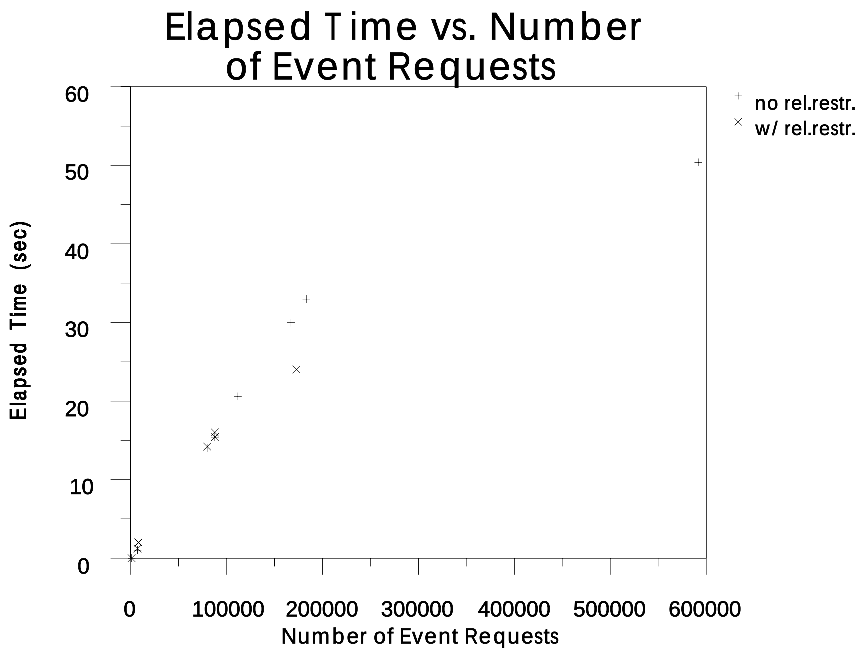


Figure 5.7: Combined data, with and without relevancy restrictions

stantial amount of timestamp-calculation requires somewhat less time per event request issued. This disparity is visible in Figure 5.7. Those searches which do not require a significant amount of timestamp-calculation (in this graph, all searches except for those which took 22.4 seconds and 46.0 seconds to execute) take roughly 0.18 seconds per thousand event requests, while the rest take less time per request because some of those requests are for timestamp calculations.

5.2.3 Conclusions

The above data suggest that the time required to complete a search is equal to $c_1 f + c_2 g$, where f is the expected number of primitive events to be examined, g is the number of timestamp requests issued, and c_1 and c_2 are constants such that

$c_1 \ll c_2$. Thus,

Conjecture 3

The running time of the search algorithm is $O(f + g)$.

Since the employment of relevancy restrictions can both decrease f and increase g, it may either improve or degrade performance.

The values for f and g are both $O(n^t)$, where n is the size of the event-trace file and t is the size of the predicate. It is important to note, however, that this exponential time bound can be substantially avoided through the judicious construction of predicates.

5.3 Improving Running Time

As the discussion above suggests, it may be necessary to speed up the search process. This section sketches some techniques which might assist in the attainment of that goal.

- **Index by event type.** If a sufficient amount of reasonably fast storage is available, a linked list could be maintained for each event type within each process, so that the occurrences of that type of event could be quickly enumerated. This would, of course, require a pre-processing pass over the event-trace data. In case the memory requirements were too severe, a compromise approach could employ indexing on some event types and/or processes in conjunction with a query engine which would search for indexed terms before turning to other terms. Alternatively, just recording the first and last occurrence of each event type on each trace could provide a significant speedup without much additional space.

- **Employ special knowledge about event types.** All of the predicates used in the test runs described above are trivially verifiable by an informed human, because people know that *start* events can only occur as the first event recorded for a process, and that *exit* events must be the last. Knowledge of the special qualities that accompany events like these could be put to good use by the search algorithm.

- **Search for less-frequent terms first.** In the test runs above, Predicate VII

  [dbg_session, exit,] || [ts_chkpt, wait for events,]

  can be searched for in a reasonable length of time. On the other hand, a search for the predicate

  [ts_chkpt, wait for events,] || [dbg_session, exit,]

  fails to return an answer after twenty minutes. In the first case, the trace for *debug session* is searched once and then the trace for the *checkpoint* process is also searched for a *wait for events* event which is concurrent with the one *exit* event in the trace for *debug session*. In the second case, the trace for the *checkpoint* process is searched once and, for each of the many thousands of events of type *wait for events*, the *debug session* trace is searched for an *exit* event which is concurrent with that *wait for events* event. Obviously, some heuristic attempt to identify which terms occur less frequently and then search for matches to those terms first could yield an impressive performance improvement.

These suggestions can be grouped under the general title of **query optimization**. Like its homonym in database retrieval, the possible techniques are numerous and the details of their application complex; a full discussion of this area is beyond the scope of this thesis.

Chapter 6

Conclusions and Future Directions

6.1 Summary of the Problem

While the idea of building distributed systems is not a new one, recent developments in hardware technology suggest that their use will soon be both a sound economic move and a performance-driven necessity. Developing software for such systems remains a significant stumbling-block to their widespread acceptance; this is largely because of the difficulty of debugging distributed applications. In the past few years, substantial progress has been made toward the provision of a distributed extension of the traditional sequential-debugging tool suite; the development of a solid event-predicate-detection facility would fill a large gap that remains.

6.2 Thesis Contributions

To provide a solid foundation for event-detection predicates, precedence relations ($\rightarrow$ and $\rightsquigarrow$) are defined for compound events and, for the recommended relation

($\rightarrow$), two alternative vector-timestamp-calculation algorithms and corresponding precedence tests are developed and proven correct.

Five goals, which an ideal predicate-detection system should meet, are enumerated: detecting phase transitions and violations of mutual exclusion, locating subroutines and asymmetrical communication patterns, and identifying performance bottlenecks. These provide a new framework for comparison within which the previous work in this area is evaluated. Based on this evaluation, a new predicate-specification syntax is introduced, heavily influenced by that of Haban and Weigel [20], but with the important extensions of event-specification wildcards and send-receive pairing.

A prototypical implementation has been completed, and various issues which were dealt with in the course of its construction are discussed. Finally, the factors affecting the performance of this prototypical algorithm are investigated, and several enhancements which could improve performance are suggested.

6.3 Future Work

As is discussed in Chapter 5, the time complexity of the search algorithm may prove to be a problem; it would be useful to pursue a closer investigation of the trade-offs involved with the various performance-enhancing techniques sketched in Section 5.3. Consideration should also be given to the issues (performance and otherwise) involved in employing this predicate-recognition engine as an online monitor in addition to its current use as a post-mortem analysis tool.

There is much that could be done to improve the expressive control and ease of specification provided by the predicate-definition language. The additions sketched in Section 3.3.3 (repetition operators, a "not yet" operator, named subblocks, binding and real-time information) may provide a good starting-point. In addition, the construction of "libraries" of commonly-used subblocks could ease the development of specifications in much the same way as "include files" facilitate the task of programming with a macro-assembler.

More fundamentally, it can be argued that the current definition language, which uses single, primitive events as its fundamental building blocks, is at too low a level to permit the construction of complex predicates with a reasonable amount of effort. While the system presented in this thesis provides a useful foundation, it might be appropriate to construct a higher-level language on top of it, much as many modern, high-level languages are built on top of an assembly language.

6.4 A Final Word

This is an exciting time in the computer-systems field; the world is embracing a rapid shift in favour of distributed architectures, thus placing new strains on the software developers. It is this author's hope that the prototypical predicate-recognition tool, and the associated background information presented in this thesis, will provide a basis for fertile discussions between programming professionals and those researching debugging techniques. Perhaps, after a suitable iterative-refinement process, this tool may provide an effective resource in the development of distributed applications.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Massachusetts, 1988.

[2] John Backus. The history of FORTRAN I, II, and III. *Annals of the History of Computing*, 1(1):21–37, 1979.

[3] A. A. Basten. Event abstraction in modeling distributed computations. In K. Ecker and M. Krämer, editors, *Workshop on Parallel Processing, Proceedings*, pages 46–65, Lessach, Austria, September 1993.

[4] A. A. Basten. Hierarchical event-based behavioral abstraction in interactive distributed debugging: A theoretical approach. Master's thesis, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands, 1994.

[5] T. Basten, T. Kunz, J. P. Black, M. H. Coffin, and D. J. Taylor. Time and the order of abstract events in distributed computations. Computer Science Note 94/06, Eindhoven University of Technology, Department of Mathematics and Computing Science, Eindhoven, The Netherlands, February 1994.

[6] Peter C. Bates and Jack C. Wileden. High-level debugging of distributed systems: The behavioural abstraction approach. *The Journal of Systems and Software*, 3(4):255–264, December 1983.

[7] Peter A. Buhr. Introduction to concurrent programming using μC++. Forthcoming.

[8] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[9] Wing Hong Cheung. *Process and Event Abstraction for Debugging Distributed Programs*. PhD thesis, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada, 1989.

[10] Hsien-Kuang Chiou and Willard Korfhage. Detecting ENF event predicates in distributed systems. Unpublished manuscript. Computer Science Department, Polytechnic University, Brooklyn, NY 11201.

[11] Hsien-Kuang Chiou and Willard Korfhage. Enhancing distributed event predicate detection algorithms. Unpublished manuscript. Computer Science Department, Polytechnic University, Brooklyn, NY 11201.

[12] Hsien-Kuang Chiou and Willard Korfhage. Efficient global event predicate detection. In *14th International Conference on Distributed Computing Systems*, pages 642–649, 1994.

[13] Gordon V. Cormack. An LR substring parser for noncorrecting syntax error recovery. *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation, SIGPLAN Notices*, 24(7):161–169, 1989.

[14] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *Proceedings of the 11th Australian Computer Science Conference*, pages 56–66, Brisbane, 1988.

[15] Colin J. Fidge. *Dynamic Analysis of Event Orderings in Message-Passing Systems*. PhD thesis, Australian National University, Department of Computer Science, Canberra, Australia, 1989.

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

[17] Vijay K. Garg and Brian Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.

[18] S. Gill. Parallel programming. *The Computer Journal*, 1(1):2–8, April 1958.

[19] Virgil D. Gligor and Susan H. Shattuck. Deadlock detection in distributed systems. *IEEE Transactions on Software Engineering*, SE-6(5):435–440, September 1980.

[20] Dieter Haban and Wolfgang Weigel. Global events and global breakpoints in distributed systems. In *Proceedings of the 21st Annual Hawaii International Conference on System Science*, pages 166–175, 1988.

[21] Wenwey Hseush and Gail E. Kaiser. Modeling concurrency in parallel debugging. *ACM SIGPLAN Notices*, 25(3):11–20, 1990.

[22] S. C. Johnson and M. E. Lesk. Language development tools. *The Bell System Technical Journal*, 57(6):2155–2175, July-August 1978.

[23] Marc Khouzam. Single stepping in event visualization tools for distributed applications. Master's thesis, Department of Computer Science, University of Waterloo, Ontario, Canada, 1996.

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

[25] Thomas Kunz and David J. Taylor. Visualizing PVM executions. In *Proceedings of the 3rd PVM Users' Group Meeting*, Pittsburgh, May 1995.

[26] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 27(7):558–565, July 1978.

[27] Keith Marzullo and Susan Owicki. Maintaining time in a distributed system. *Operating Systems Review*, 19(3):44–54, July 1985.

[28] F. Mattern. On the relativistic structure of logical time in distributed systems. In M. Cosnard et al., editors, *Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V., Amsterdam, North-Holland, The Netherlands, 1989.

[29] Barton P. Miller and Jong-Deok Choi. Breakpoints and halting in distributed programs. In *8th International Conference on Distributed Computing Systems*, pages 316–323, 1988.

[30] Oystein Ore. Theory of graphs. *American Mathematical Society Colloquium Publications*, 38, 1962.

[31] M. Krish Ponamgi, Wenwey Hseush, and Gail E. Kaiser. Debugging multithreaded programs with MPD. *IEEE Software*, pages 37–43, May 1991.

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

[33] Ilene Seelemann. Application of event-based debugging techniques to object-oriented executions. Master's thesis, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada, 1995.

[34] Michiel F. H. Seuren. Design and implementation of an automatic event abstraction tool. Master's thesis, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada, 1996.

[35] James Alexander Summers. Precedence-preserving abstraction for distributed debugging. Master's thesis, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada, 1991.

[36] Andrew S. Tanenbaum. *Structured Computer Organization.* Prentice-Hall, third edition, 1990.

[37] Andrew S. Tanenbaum. *Distributed Operating Systems.* Prentice-Hall, 1995.

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

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

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

[41] David J. Taylor, Thomas Kunz, and James P. Black. Achieving target-system independence in event visualization. In *CD-ROM Proceedings of the 1995 CAS Conference*, 1995.

[42] J. H. Wilkinson. The Pilot ACE. In *Automatic Computation: Proceedings of a Symposium held at the National Physical Laboratory*, pages 5–17, March 1953.

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

[44] Jeffrey S. Young. *Steve Jobs: The Journey is the Reward.* Lynx Books, 41 Madison Avenue, New York, New York, 10010, 1988.

[45] Ivan Y. K. Yu. Integrating event visualization and sequential debugging. Master's essay, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada, 1996.

Symbol	ASCII	Meaning	see page(s)
$\neg$	`NOT`	logical negation	
$\wedge$	`AND`	logical conjunction	
$\vee$	`OR`	logical disjunction	
$\Leftrightarrow$	`<=>`	bidirectional implication (iff)	
$\backslash$		set subtraction	
$\overset{vec}{\leq}$		vector $\leq$	13
$\overset{act}{<}$		$<$ over $active()$	22
a	`a`	primitive event	
A	`A`	compound event	17
$\vert A\vert$	`|A|`	cardinality of A	25
p_a		$proc(a)$	10
T_a		timestamp	11, 13, 15, 17
$T_{A.front}$		front timestamp	19
$T_{A.metron}$		metron timestamp	22
$T_{A.back}$		back timestamp	19
$\rightarrow$	`-->`	sequential composition	9, 19
$\Vert$	`||`	concurrent composition	9
$\rightsquigarrow$	`~~>`	alternative $\rightarrow$	24
$\wr\wr$	`{}`	alternative $\Vert$	24
$A \overset{B}{\rightarrow} C$	`A-(B)->C`	limited $\rightarrow$	45
$A \overset{B}{\rightsquigarrow} C$	`A~(B)~>C`	limited $\rightsquigarrow$	45
$x*$	`x*`	repetition of x	33, 54
$A@$	`A@`	alternative repetition	54
$A \overset{B}{*}$	`A{B}*`	repetition with limiter	55
$A \overset{B}{@}$	`A{B}@`	alternative repetition with limiter	55