Online Monitoring of Distributed Systems Using Causal Event Patterns

by

Sukanta Pramanik

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2014

© Sukanta Pramanik 2014

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Event monitoring and logging, that is, recording the communication events between processes, is a critical component in many highly reliable distributed systems. The event logs enable the identification of certain safety-condition violations, such as race conditions and mutual-exclusion violations, as safety is generally contingent on a specific causally ordered pattern of process communication. Previous efforts at finding such patterns have often focused on offline techniques, which are unable to identify operational problems as they occur. Online monitoring tools exist but they are often restricted to identifying a specific violation condition, such as a deadlock or a race condition, using dedicated data structures. We address the more general problem of detecting causally related event patterns that can be used to identify various undesired behaviours in the system.

The main challenge for online pattern matching is the need to store the partial matches to the pattern, as they may combine with future events to form a complete match. Unlike pattern matching in most other domains, causally ordered patterns can span a potentially unbounded number of events and efficiently searching through this large collection poses a significant challenge.

We present an efficient online causal-event-pattern-matching framework that bounds the number of partial matches it stores by reporting only a representative subset of pattern matches. We define a subset of matches as representative if it has at least *one occurrence of each event in the pattern on each process*, which is applicable for a large class of distributed applications. Our first pattern-matching algorithm, OCEP introduces a backtracking algorithm to efficiently find a representative subset from the history of events. An evaluation of the framework shows that OCEP is capable of handling several frequently occurring violation patterns at the event rates of some representative distributed applications.

Our second algorithm, Ananke, introduces causality-based rules in the search pattern that can be used to specify the removal of an event from the maintained history. We used some of the most frequently occurring types of concurrency bugs in real-world applications to show that the desired causal order of events can be utilized to specify such removal rules. More importantly, these rules are able to maintain a finite history and still report a representative set of matches within a millisecond in most cases.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, David Taylor. David has worked for a while on partial-order relationship among events and how it is a more accurate representation of causality than the real-time of event occurrences. I found it fascinating that we can use it to detect various types of faults rather than a specific one. This work is the result of my discussions with him about the various aspects of causal event patterns and their use in monitoring of distributed systems. I would also like to thank him for his endless research inspiration and guidance. I can recall numerous occasions when I would present to him some roadblock along with my failed attempts to circumvent it and he will point me to the right direction after asking a few questions. I have enjoyed every moment of working with him and will miss our Friday meetings when research talks moved into contemporary topics, symphony, theatre, and sometimes cooking. He provided just the right mix of guidance and independence to allow me to come into my own as a researcher.

I would like to thank the members of my dissertation committee – Paul Ward, Tim Brecht, and Martin Karsten – for their invaluable comments and helpful suggestions. I would also like to thank Robbert van Renesse for agreeing to serve as the external examiner of my thesis.

I would also like to thank past and present members of the Shoshin group and the SyN group (formerly NDS) for their support and comments. Shoshin and NDS talks are a wonderful place to receive early comments and suggestions about one's work and I have used it often to float premature ideas. I would especially like to thank Paul Ward, Tim Brecht, Bernard Wong, and Srinivasan Keshav whose suggestions I have used in my work. The Shoshin lab has been part of my home for a while and I will never forget the delightful memories here. In particular, I have had conversations about various topics both related and unrelated to research with Ahmad, Omar, Hao, Thomas, Jakub, Akshay, Sharon, Li, Jack, Sukhbir, Masoomeh, Jim, Reaz, and Carol.

I am eternally grateful to my wonderful family for loving and supporting me through my PhD life. I know this was *the* only way to fulfill my parent's dreams as I did not go into the Medical profession. But I only received encouragement from them without any pressure to succeed. My brother, who is also one of my best friends, could always take my mind out of complicated thoughts with his talks about new stamps, coins, and animes. And of course, my wife Sudeshna, who now barely laughs while reading PhD Comics. She has been a pillar in my life for a while without whom I could never have finished this endeavour. Lastly, my two little angels Sohaum and Shomik, who made my life more challenging but also so much more joyful that it helped me remain sane. Although I do not show it often, I love all of you more than I can ever say.

Finally, I would like to acknowledge the financial support that I received from the University of Waterloo in the form of various scholarships, awards, and teaching opportunities.

Table of Contents

Table of Contents ix

List of Figures xi

1 Introduction **1**
1.1 Main Contributions 2
1.2 Thesis Outline 4

2 Background and Related Work **5**
2.1 Monitoring Distributed Systems 5
2.2 System Model 7
2.3 Global Property Detection 12
2.4 State-Based Detection 13
2.5 Event-Based Detection 15
2.6 Related Work 18

3 OCEP Algorithm **23**
3.1 Pattern Language 23
3.2 Online Event-Pattern Monitoring 28
3.3 OCEP Framework 29
3.4 Performance Evaluation 40
3.5 Conclusion 50

4 Subset-Based Algorithm **53**
4.1 System Model and Pattern Language 54
4.2 Bottom-Up Search 55
4.3 Minimizing the Event-History Size 57
4.4 Compact Subset 60

4.5 Finite Automaton 63
4.6 Compound Events 66
4.7 Conclusion 76

5 Removal by Rules **79**
5.1 Pattern Language 80
5.2 Pattern Search Algorithm 81
5.3 Managing History 82
5.4 Building the Patterns 92
5.5 Performance Evaluation 96
5.6 Conclusion 103

6 Conclusion **105**
6.1 Future Work 107

References **109**

List of Figures

2.1 Process-time diagram for visualizing a distributed system 7
2.2 Observability issues in monitoring 8
2.3 Causal ordering of events using vector timestamps 12
2.4 Process-time diagram with *consistent* and *inconsistent cuts* 14
2.5 Using state-lattice for state-based detection 14

3.1 Causality operators for patterns 25
3.2 Process-time diagram showing some example patterns 25
3.3 Grammar for specifying a pattern 26
3.4 Choosing a representative subset for $A \rightarrow B$ 29
3.5 The structure of the pattern tree 30
3.6 Restricting domain of e_i with respect to e 35
3.7 Using causality to update domain when backtracking 36
3.8 Search example – pruning the search space in $goForward$ 38
3.9 Search example – resolving conflict in *goBackward* 39
3.10 Architecture of POET: Server with one local viewer 40
3.11 POET screenshot 42
3.12 Execution time for detecting deadlock 44
3.13 Execution time for detecting message races 45
3.14 Execution time for detecting an atomicity violation 47
3.15 Execution time for detecting an ordering bug 48
3.16 Detailed execution time for test cases 49

4.1 A grammar for specifying a simple pattern language 54
4.2 Pattern tree for a bottom-up search stores match history at all nodes 55
4.3 Parameters for the complexity analysis 55
4.4 Compound patterns cannot be matched by storing n matches per event 59
4.5 Determination of coverage using timestamps 60

4.6 Completeness problem in the tree-based algorithm 62
4.7 Tracking incomplete matches with a finite automaton 63
4.8 Completeness problem with finite automaton 65
4.9 A single state in a finite automaton with all its transitions 66
4.10 Causal relations in *incomplete matches* 67
4.11 Communication events adding new matches to coverage set 69
4.12 Using *Prec-Trail* for individual events in an incomplete match 71
4.13 Communication events removing existing matches from $Cov(\alpha, e)$ 73

5.1 Grammar for specifying a pattern with removal rules 80
5.2 Pattern tree and the use of trigger events along with the removal rules 81
5.3 Using rules for removing events 83
5.4 Removal decisions for concurrency 83
5.5 A new communication event is used to update *gpTrace* 86
5.6 The first stored event on each trace is used to calculate *firstIndex* 87
5.7 *earliestGP* is calculated from the GPs to the existing matches 89
5.8 Some deadlocks may remain hidden because of network buffering 93
5.9 The events that follow the desired program behaviour can often be removed 95
5.10 Boxplots for execution time with Ananke 97
5.11 Cumulative distribution of execution time for deadlock pattern 98
5.12 Cumulative distribution of execution time for message-races pattern 98
5.13 Cumulative distribution of execution time for atomicity-violation pattern 98
5.14 Cumulative distribution of execution time for ordering-bug pattern 99
5.15 Summary statistics for the execution time for all the test cases 100
5.16 History size and individual execution time for deadlock and message races 101
5.17 History size and individual execution time for atomicity violation and ordering bug . 102

Chapter 1

Introduction

The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.

Brian W. Kernighan in the paper Unix for Beginners (1979)

The explosive growth of the Internet in the 1990s has moved distributed systems beyond their traditional application areas to industrial automation, defense, telecommunications, and into nearly all domains, including e-commerce, health care, government, emergency services, and entertainment. In the past few years we have also seen a growing popularity of large-scale Web services that rely on distributed application backends. These distributed backends are designed to scale horizontally to utilize the vast quantity of resources available in modern datacenters. Supporting such large-scale deployments, however, introduces additional uncertainty and complexity to these distributed applications, which already have complicated communication patterns to support sophisticated and demanding Web services. These factors make it incredibly difficult to reason about the correctness of a modern distributed application.

One common approach to help with understanding the runtime behaviour of a distributed application is to track its execution over time in order to capture its runtime-state information. The collected information is then used to detect whether a property is satisfied or violated in the global state. This approach of *global-predicate* detection is a well studied problem and is based on building a lattice of global states [Cooper and Marzullo, 1991], which is known to be NP-complete [Mittal and Garg, 2001].

The semantics intended by the programmer for an application is also closely related to a *linearized history* of the system [Herlihy and Wing, 1990]. A *linearized history* is a finite sequence

of events that consists of either invocation of operations or responses to operations. Software behaviour can then be analyzed by monitoring for a pattern of events that represent some undesired behaviour, such as bugs, misuse, or intrusions. For example, in a traffic-light system, a correctness condition is that lights in only one direction may be green in the global state. Alternatively, this problem can be modeled as a sequence of events between the lights. An event-matching-based approach monitors the events e_i that denote light i has turned green and then searches for a pattern that represents two events e_i and e_j happening concurrently. A match to this pattern signifies that the system is in an unsafe state. Similar patterns can also be used to identify mutual-exclusion violations in a distributed application even if the actual global state observed by the system is correct [Schwarz and Mattern, 1994].

In this work, we introduce an efficient online framework for detecting faults in a distributed application using causal event-patterns. We define an event to be the act of sending or receiving a message between two components of a monitored application or the occurrence of another relevant state transition in a component (where "relevant" is a matter of subjective judgment for each application environment). A causal event-pattern then represents a complex interaction as a distributed sequence of causally-ordered events. Matching a causal event-pattern allows us to reason about the correctness of many distributed systems without requiring the global state.

1.1 Main Contributions

We use generic causal-patterns that can represent different types of undesired behaviours such as deadlock, race condition, atomicity violation, etc. The pattern language that we use for specifying patterns of partial-order events is derived from existing work [Xie, 2003; Kunz, 1994]. We have added *variable binding* to the attributes of the monitored events, which increases the expressiveness of the search patterns.

The major challenge that we faced for reporting these matches is that it requires storing a substantial amount of intermediate information, representing partial pattern matches, to match arbitrary causal relationships. This affects the runtime performance in two different ways. Firstly, when monitoring a set of arbitrarily long-running processes, the amount of memory required to store the partial matches may also become arbitrarily large. Secondly, the monitoring algorithm needs to traverse this large collection of partial matches on each event in order to extend them, potentially, to a total match.

Our first algorithm, OCEP, handles the second challenge by utilizing the vector timestamps and the causality relations among the events in the pattern to prune the search space. We also define a subset of matches that is *representative* of the set of all matches. We evaluate OCEP using some of the most frequently found concurrency-bug patterns in real-world applications.

OCEP successfully reports a *representative subset* of matches to the pattern in each application within a millisecond in almost all cases. Thus OCEP can handle an event rate of one thousand (or usually more) events per second. It should be noted that this is not the rate of arrival for all events, nor even all events matching some component of a pattern, but only *terminating events* (defined in Section 3.3.2), which can potentially complete a pattern match. Thus, in most situations, an overall event-arrival rate much higher than one thousand events per second can be handled.

We have proven that for some patterns an exponential (in the pattern-length) number of partial matches must be stored in order to report a match, if any exists. Thus, a search algorithm that maintains a polynomially-bounded subset of partial matches may fail to report some matches. Our second algorithm, Ananke, introduces user-defined rules in the search pattern that allow the use of application-specific knowledge for removing redundant events. Ananke uses these rules along with the causality among the events in the pattern to determine when an event can no longer generate a unique match. We show, using the same concurrency-bug patterns, that Ananke is able to maintain a finite history while still being able to report a representative subset of matches.

Specifically, this thesis provides the following contributions which sets it apart from the related work in this field.

- Our pattern-detection algorithms are *online*, i.e., they report matches to the pattern as the application is executing.
- Our causal patterns are generic and can represent various types of undesired behaviours.
- A reported match to the pattern contains one match for each individual event in the pattern. Information for each event also includes the participating process and the real-time when it occurred.
- Our reported matches may span the entire execution time, i.e., we do not restrict the search domain of our pattern search to a finite window of past events.
- We have defined a *subset of matches* that is *representative* of all the matches during the entire execution time. The subset is representative in the sense that if there are many matches, only a subset is reported, which contains at least one match with an event occurring on each trace (if it exists). This definition also ensures that we always report a unique match. A more precise definition of *representative subset* is provided in Section 3.3.1.
- We use some of the most frequently found concurrency-bug patterns in real-world applications to show that our pattern-detection algorithms successfully report a representative subset of matches for each application within a millisecond in almost all cases. Thus they can provide an essentially immediate response for an event rate of thousands of terminating

events per second. For a higher event rate, the slowdown will mostly affect pattern detection. The application itself is only affected by the event collection as both the timestamping and pattern detection happen outside its scope.

- Ananke is the more scalable of the two algorithms as it can maintain a finite event history, using user-defined event-removal rules, in situations where the history maintained by OCEP grows without bound.

1.2 Thesis Outline

In Chapter 2 we begin with necessary background for monitoring distributed systems. We provide our system model for causality-based event-pattern detection and review related work.

The main body of the thesis is in three chapters. Chapter 3 presents our first online algorithm OCEP. We define our pattern grammar and how it is used to create a search pattern that represents a fault. We also define our reported subset of matches and explain why we consider it to be *representative*.

OCEP uses a very simple technique for removing redundant events, which is not very effective in practice. Ideally an event-removal technique should utilize the causality relationships among the events specified in the pattern. In Chapter 4, we explore the elusive idea of maintaining a finite subset that can be used to detect the matches to any generic pattern. We look at what role the causality relation plays in event removal and theoretically analyze a number of approaches that can be used.

In Chapter 5, we present our second algorithm, Ananke, that uses a rule-based approach for event removal. Ananke exploits the causality among the events in the pattern through some *default rules*. It also updates the grammar so that user-defined removal rules can be provided in the search pattern.

Finally in Chapter 6, we summarize our achievements and discuss how our work can further be extended.

Chapter 2

Background and Related Work

You know you have a distributed system when the crash of a computer you have never heard of stops you from getting any work done.

Leslie Lamport in Security Engineering

A distributed system is a collection of processes working together to accomplish some common task. These processes are typically located on multiple computers connected by a network and they coordinate their actions by passing messages [Coulouris et al., 2011]. In this chapter we discuss different aspects of distributed systems that have an effect on monitoring. We also provide our system model and the necessary background for our causality-based event-pattern detection. Additionally, we explore previous work in this field and show how it is related to our work.

2.1 Monitoring Distributed Systems

The monitoring of distributed systems can be defined as the process of dynamically extracting information about the processes and their interactions, collecting this information, and presenting it to users in useful formats [Joyce et al., 1987]. The collected information can be used for various purposes such as debugging, performance evaluation, program visualization, and monitoring of an application for error conditions.

Monitoring a distributed system is more complex compared to a system running on a single processor. Two aspects that play a key role are *distribution* and *concurrency*.

Distribution concerns the presence of multiple components that may be physically located in different places. *Concurrency* is concerned with a program having multiple independent components whose execution may overlap in time [Magee and Kramer, 2006]. A related concept is *parallelism*. A *parallel* program is one which runs on multiple processors at the same time.

The concept of concurrency is orthogonal to the concept of distribution or parallelism. It is possible to execute a concurrent program on a uniprocessor using multiple threads of control. These threads are then interleaved in an arbitrary way by the scheduler and the programmer uses some type of synchronization to ensure the desired outcome regardless of the scheduling order. It is also possible to execute a sequential program distributed across multiple machines (by using remote procedure calls) or in parallel on multiple processors or cores (by parallelizing the calculations). Thus, *concurrency* is a semantic property of a program while *parallelism* or *distribution* describe its execution environment.

Many of the implicit assumptions in a system running on a single processor become invalid when monitoring a distributed system:

Global clock Distributed systems do not have a shared global clock or synchronized local clocks. Each process executes on a machine with its own local clock and without specialized clock-synchronization hardware it is not possible to determine an exact real-time ordering of events happening across multiple machines. We can only observe a partial order between events that occur in processes on different machines [Lamport, 1978].

Reproducibility A distributed system that has concurrent components has an inherent nondeterminism, because a set of concurrent or causally independent events may occur in any order [Schwarz and Mattern, 1994]. The result is that the errors that are found in one execution of the application may not be repeated in the subsequent ones.

Observability Monitoring tools that need to check a global property must collect the locally observed states from all of the processes to construct a global view. It is very difficult to obtain a consistent global state as the order in which the states are collected at the monitor may be inconsistent or even incorrect because of unpredictable communication delays [Fidge, 1996].

Probe effect When auxiliary code is added to an application for monitoring, this may prevent certain erroneous computations from occurring or may introduce new errors that were not part of the original program [Fidge, 1996]. Also, the monitoring system may itself compete for resources with the system being observed and thus modify its behaviour.

Data size The amount of information in a large system can easily swamp the monitoring application, thus necessitating filtering of the information.

Our work focuses on distributed systems that also have concurrent components. Concurrent programs, however, are inherently nondeterministic whether their execution is local or distributed. For example, if we are executing a program with two concurrent components on a uniprocessor, the observed events from these components can be totally ordered using the local clock. This temporal causality is, however, inaccurate as two independent events may occur in any order in a single execution.

2.2 System Model

We model our distributed system as a finite set of n sequential processes $P_1, P_2, \ldots, P_n$ communicating only by message passing. The processes do not share memory and furthermore there is no global clock or perfectly synchronized local clocks. Each process P_i executes a local algorithm that controls its behaviour by changing its *local state* s_i. The process's state includes the values of all the variables within it as well as the values of any local objects it affects, such as files. We assume a new process P_i can start at any point, however, such a P_i must be causally related to one of the existing processes.

The local algorithm executing at process P_i takes a series of actions, each of which is either an operation that transforms P_i's state or a message *send* or *receive* operation. The occurrences of these actions performed by the local algorithm are called *events*. The coordinated execution of all these local algorithms running concurrently is what we call a *distributed computation*.

The *global state* of a distributed computation is defined as the collection of the local states of all processes as well as the state of the communication channels on which a message is in transit at *a certain instant of time*.

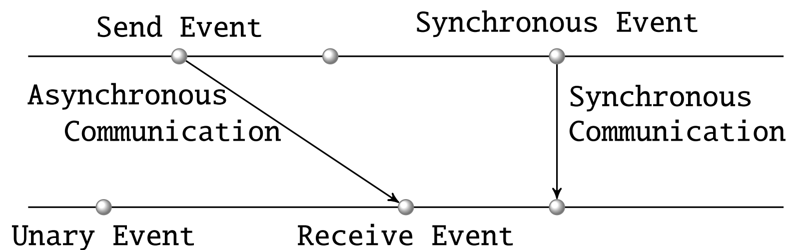


Figure 2.1: Process-time diagram for visualizing a distributed system

A convenient way of visualizing a distributed computation is by means of a process-time diagram, as in Figure 2.1. Each process is represented by a line in which time moves from left to right. For our model, it suffices to distinguish three types of events: send events, receive events,

and unary events (state transitions that do not communicate with other processes). An event is shown as a solid dot on the process lines while a message is represented by a directed arrow connecting a send event with its corresponding receive event. Communication events (send and receive) can be of two types: *synchronous* and *asynchronous*. A *synchronous* communication event blocks the process until the operation is complete and is indicated by a vertical line. An *asynchronous* communication is non-blocking and is represented by a diagonal line.

2.2.1 Causal Ordering of Events

The essential problem in monitoring the global state is the absence of global time. We can observe the succession of states in an individual process as the events that occur on a single process are totally ordered. We, however, need to collect the local states of all the processes at *a certain instant of time* to detect the actual global state of the system. Since there is no global clock, all the processes cannot readily agree on a time at which to record their local states.

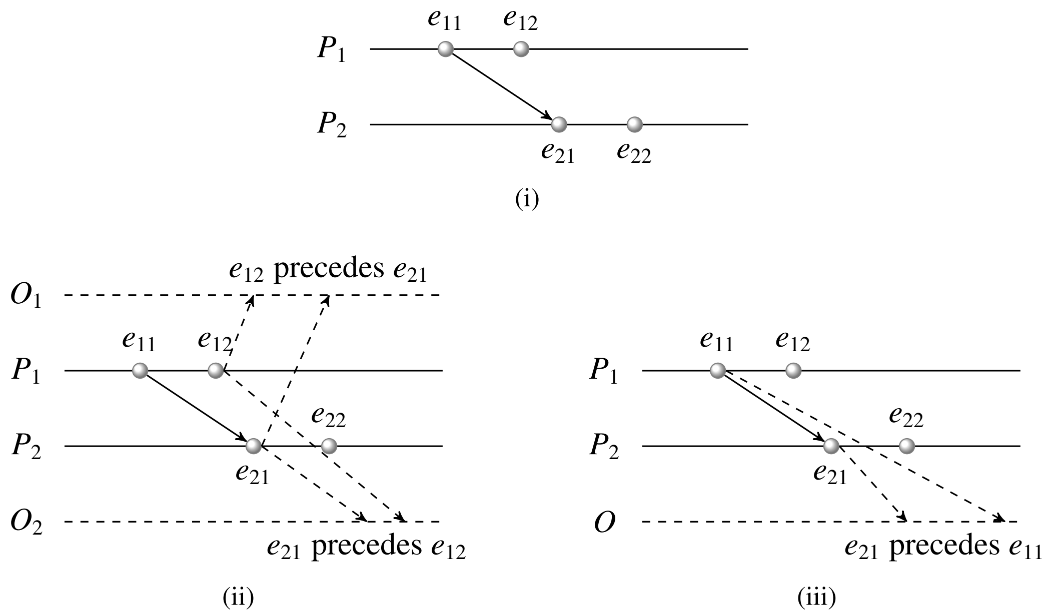


Figure 2.2: Observability issues in monitoring: i) A distributed computation, ii) Multiple observers see different orderings, iii) Observer assumes incorrect ordering [Schwarz and Mattern, 1994]

Another problem is that distributed systems have an inherent nondeterminism introduced by varying message delays. A set of concurrent or causally independent events may occur in any

order, possibly yielding different results in each case [Schwarz and Mattern, 1994]. Because of this, an observer who relies on the arrival time of notifications to determine event orderings may assume arbitrary ordering between unrelated events. Figure 2.2 shows this *observability* problem, presenting two different kinds of difficulty that may occur when observing the computation in Figure 2.2(i). In Figure 2.2(ii) two observers O_1 and O_2 see different orderings of events e_{12} and e_{21} for the same program. In Figure 2.2(iii) observer O assumes an incorrect ordering of events e_{11} and e_{21}.

In a distributed system the ordering of events defined by totally ordered clocks will provide an incomplete view of causality [Fidge, 1996]. In order to have a more intuitive description of concurrent events the ordering of events occurring on different nodes has to be a *partial order* [Peled, 2001].

A *(strict) partial order* is a binary relation ($<$) over a set A that satisfies the following three properties

1. *Irreflexivity*: $a \nless a$ for all $a \in A$
2. *Transitivity*: If $a < b$ and $b < c$ for any $a, b, c \in A$ then $a < c$
3. *Asymmetric*: If $a < b$ then $b \nless a$ for any $a, b \in A$

A total order is a partial order that satisfies a fourth property known as *comparability* which is defined as, for any $a, b \in A$ with $a \neq b$ either $a < b$ or $b < a$. Thus a partial order is an order defined for some, but not necessarily all, pairs of items.

If we view the process-time diagram as a directed acyclic graph, it is clear that an event e_{il} on process i can only affect another event e_{jm} on process j if there exists a directed path from e_{il} to e_{jm}. An alternative is to view it as a set of events with a partial order where causality is defined by the *happened-before* relation [Lamport, 1978].

Definition 1: *Happened-before*
A *happened-before* relation, denoted by $\rightarrow$, on the set of events of a system is the smallest relation satisfying the following three conditions:

1. if a and b are events in the same process, and a comes before b then $a \rightarrow b$.
2. if a is the sending of a message by one process and b is the receipt of the same message by another process, then $a \rightarrow b$.
3. if $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

Based on the *happened-before* relation, the *concurrency* relation ($\|$) can then be defined as follows:

Definition 2: *Concurrent*
Two distinct events a and b are said to be *concurrent*, denoted by $a \| b$, if and only if $a \nrightarrow b$ and $b \nrightarrow a$.

For example in Figure 2.2(i), $e_{11} \rightarrow e_{12}$ and $e_{21} \rightarrow e_{22}$ according to rule 1, $e_{11} \rightarrow e_{21}$ according to rule 2, and $e_{11} \rightarrow e_{22}$ according to rule 3 (Definition 1). Finally because of the absence of any other inter-process communication, $e_{12} \| e_{21}$ and $e_{12} \| e_{22}$ (Definition 2).

An *observation* of a distributed computation is a linearization of the partial order defined by the causality relation. A linearization of a partial order $\rightarrow$ on a set X is a total order on X such that any x occurs before x' whenever $x \rightarrow x'$. As shown in Figure 2.2, a single execution of a distributed program allows its observers to obtain different *observation*s.

Thus, the notion of *consistency* in an *observation* of distributed computation is basically correctly reflecting causality. Typically measures are taken to preserve the causal consistency in an observed event sequence. In general, many different observations, which are causally consistent by definition, of a single execution exist, as illustrated in Figure 2.2(ii).

2.2.2 Logical Clocks and Vector Timestamps

Lamport also introduced the concept of a *logical clock*, which is an integer counter attached to each process, and a timestamping algorithm based on it. The basic idea is that each process i has an associated logical clock C_i, which is incremented after each event. Any event e occurring in process i gets a logical time $C_i(e)$. The global time for event e, denoted by $C(e)$, is defined as $C(e) = C_i(e)$ if e occurs in process i. A *send* event on process i will carry its logical time $C_i(s)$ along with the message. The receiving process j will update its logical clock $C_j = max(C_j, C_i(s) + 1)$. This will ensure that if a can causally affect (i.e., happened-before) b then $C(a) < C(b)$ [Lamport, 1978].

Although the total ordering generated by Lamport's logical clock is consistent with the intended happened-before relation, the scalar logical timestamp fails to generate the correct partial ordering. This is because, with Lamport's clock $C(a) < C(b)$ does not necessarily imply $a \rightarrow b$. Therefore, it is still not possible to determine the happened-before relation using Lamport's logical timestamps.

Fidge and Mattern simultaneously and independently introduced the concept of a vector timestamp that can determine the causality relation between any two events in constant time. Each process P_i maintains a vector clock C_i of size n, where n is the number of processes in the sys-

tem. Each C_i is initialized to a *zero vector* at the beginning of P_i. When an event e occurs on P_i, the vector clock C_i is changed and a timestamp T_e is assigned to e according to the following algorithm [Mattern, 1988; Fidge, 1991].

1. If e is a unary event we simply increment the 'local' entry in the timestamp.

$$C_i[i] := C_i[i] + 1;$$

$$T_e := C_i;$$

2. If e is an asynchronous send event, the local effect is identical to a unary event.

$$C_i[i] := C_i[i] + 1;$$

$$T_e := C_i;$$

3. If e is an asynchronous receive event on process P_i corresponding to the send event d on process P_h, in addition to incrementing the 'local' entry in the timestamp, we must (roughly) take an element-wise maximum with the timestamp of the send event.

$$C_i[i] := C_i[i] + 1;$$

$$\forall p \in [1 \ldots n] \wedge p \neq h, C_i[p] := max(C_i[p], C_h[p]);$$
$$C_i[h] := max(C_i[h], C_h[h] + 1);$$
$$T_e := C_i;$$

For synchronous events, we use an improvement to Fidge/Mattern timestamps proposed by Cheung that allows us to use the same precedence checking for both synchronous and asynchronous events [Cheung, 1989].

4. If e is a synchronous event on processes P_i and P_j,we take an element-wise maximum of the two clock values and, after assigning timestamps increment the 'partner' entry in each clock.

$$C_i[i] := C_i[i] + 1;$$

$$C_j[j] := C_j[j] + 1;$$

$$\forall p \in [1 \ldots n], C_i[p] := C_j[p] := max(C_i[p], C_j[p]);$$

$$T_e := C_i;$$

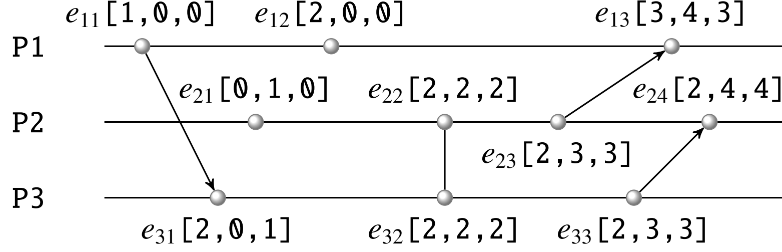


Figure 2.3: Causal ordering of events using vector timestamps

Finally, the local vector clocks on processes P_i and P_j are updated as follows:

$$C_i[j] := C_i[j] + 1;$$

$$C_j[i] := C_j[i] + 1;$$

Figure 2.3 shows an example of the application of the timestamping algorithm.

If a and b are two distinct events, on processes P_i and P_j respectively, and their vector timestamps are denoted by T_a and T_b then the following is a fundamental property associated with the vector timestamps [Mattern, 1988].

$$a \rightarrow b \iff T_a(i) < T_b(i)$$

Thus vector timestamps allow the quick determination of the causality relation between two events using a single integer comparison as defined above. Given two events a and b and their timestamps, we can check whether a happens before b or b happens before a with at most two integer comparisons. If neither of these two relationships holds, two more integer comparisons between process numbers and event numbers are needed to determine equality or concurrency.

2.3 Global Property Detection

Monitoring a distributed computation often involves checking whether a certain property holds at a particular instant of its execution. These properties can be some desired behaviour (*liveness property*) or some undesired misbehaviour (*safety property*) of the system under test. Typically a global property is specified using a *global predicate* and the monitoring application checks whether this predicate is satisfied during runtime. Thus global property detection is not a good fit for determining liveness properties.

One way to detect a global predicate is to monitor the *global state* of the system which is defined as a collection of the *local states* of all the processes at that particular instant. We call this a *state-based approach*.

Alternatively, we may monitor the *state transitions* or *events* rather than the actual states. The global predicate can then define the relative causal order in which certain events occur in the system. We call this an *event-based approach*.

2.4 State-Based Detection

The state-based approach monitors a system's global state over time and detects whether a particular state occurs in an actual execution. One simple technique is to use a *snapshot algorithm* [Chandy and Lamport, 1985] to collect the global state and send it to the monitor process for detection. A global snapshot collected in this way only captures a single observation.

Alternatively the monitor process can collect the events denoting global state transitions from the observed processes. Mathematically, we can take any set of local states from the individual processes to form a global state. A local state is the effect of all the events occurring on that particular process at a particular time. So collecting a global state in this way can be compared to *cutting* the individual processes at a time-instant and gathering event history on all the processes.

Typically, timestamps are used to guarantee causal order in the observed event sequence. A *cut* that includes the *effect* but not its *cause* is an *inconsistent* cut. A cut is considered *consistent* if for each event it contains, it also includes all the events that happened-before it. In Figure 2.4 the cuts C_1 and C_2 are consistent but C_3 is inconsistent as it includes the receive event e_{14} but not the corresponding send event e_{23}.

When causal order is maintained, at every point of an observation, the set of events that have been observed so far forms a consistent cut. Hence, every observation induces a totally ordered sequence of consistent global states. The validity of a global predicate φ is then closely related to the set of observations on which it is valid. Cooper and Marzullo introduced the notion of qualifiers which can be used to refer to the set of observations that satisfies a predicate [Cooper and Marzullo, 1991].

possibly φ is true if there exists at least one *observation* which satisfies φ.

definitely φ is true if every observation satisfies φ.

The set of all consistent cuts of a distributed computation forms an *n*-dimensional state lattice [Cooper and Marzullo, 1991] as shown in Figure 2.5(ii). Each vertical line of the state

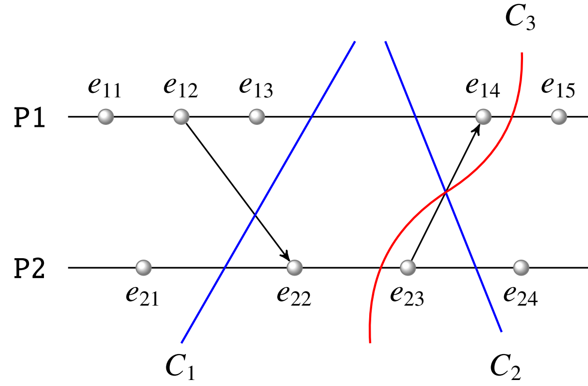


Figure 2.4: A process-time diagram showing *cuts* of which C_1 and C_2 are *consistent* while C_3 is *inconsistent*

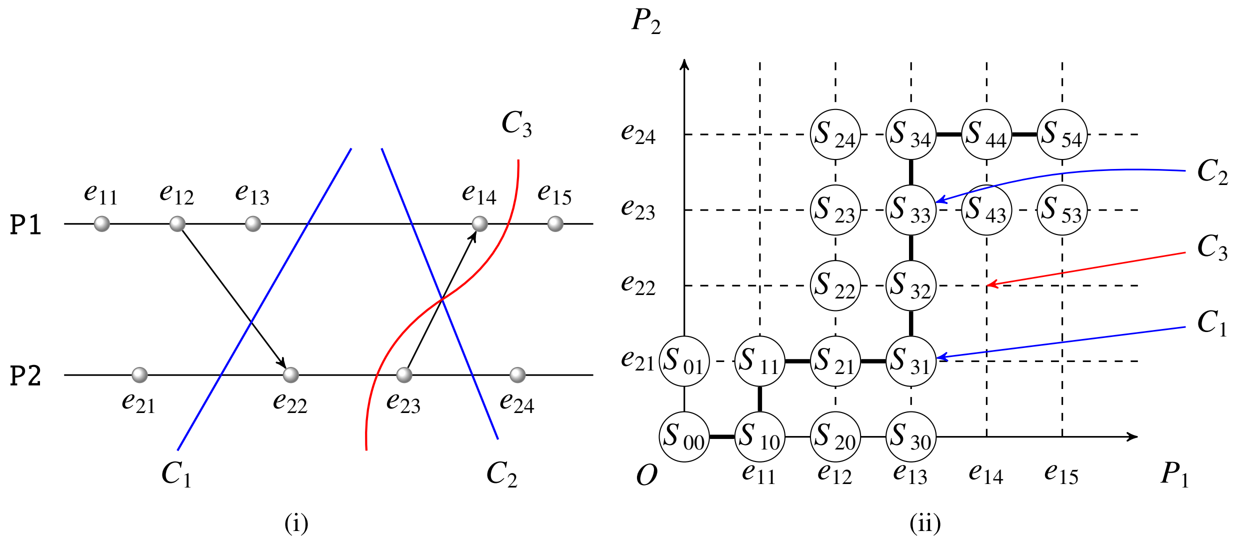


Figure 2.5: i) A process-time diagram, ii) Corresponding state lattice and a path representing an observation

lattice corresponds to an event in P_1 and each horizontal line represents an event in P_2. An intersection point $p = [e_{1i}, e_{2j}]$ of two event lines denotes a *cut* and is represented by the set $\{e_{11}, \ldots, e_{1i}, e_{21}, \ldots, e_{2j}\}$. All intersections denoting consistent cuts are marked by global states (shown as circles with the state labels). The two subscripts of a state label denote the number of events before the cut on each process. For example, in Figure 2.5(ii), S_{31} represents the cut

$C_1 = \{e_{11}, e_{12}, e_{13}, e_{21}\}$. The intersection points corresponding to an observed event sequence form a path in the lattice diagram.

So possibly φ is true if there exists a path in the state lattice and an intersection point p on it such that φ is true at p. To evaluate possibly φ, the monitor starts at the initial state and navigates through the reachable consistent cuts in the lattice, searching for an intersection point at which φ evaluates to True. Definitely φ is true if every path in the state lattice has an intersection point p on it such that φ is true at p. It thus requires inspecting all consistent states in the worst case.

The complexity of detecting both types of predicate is linear in the number of global states, but unfortunately the number of possible global states is $O(M^n)$, where M is the maximum number of local events per process and n is the number of processes [Cooper and Marzullo, 1991]. Thus using this approach for online detection of a global predicate is intractable.

It is possible, however, to restrict the predicate so that it can be detected efficiently. Garg and Waldecker used global predicates which are conjunctions or disjunctions of local predicates and the validity of these local predicates can be detected in isolation. In that case we can restrict the search to only one dimension in the lattice until the local predicate is satisfied or until a causality constraint forces us to continue on a different process [Garg and Waldecker, 1994]. The computational complexity of the detection algorithm then reduces to $O(Mn)$. This approach involves blocking all the processes but one in order to obtain the sequential execution and thus aggravates the *probe-effect* on the system's normal behaviour.

2.5 Event-Based Detection

The state-based approach tries to restrict the expressibility of predicates in order to make the algorithm tractable. As a result, it cannot detect global predicates in which the local components are causally related. Alternatively, we may focus our attention on the *events* representing the *state transitions* rather than the actual *states*. Each event, in effect, causes a transition from one global state to another. A global predicate can then be defined by the relative causal order in which certain events occur in the system. For many applications detecting such basic patterns of behaviour may provide us with sufficient insight into the resulting system state.

For example, in a traffic-light system, a correctness condition is that only the lights in opposing directions may be green in the global state. Alternatively, this problem can be modeled as a sequence of events between the lights. An event-matching-based approach monitors the events e_i that denote light i has turned green and then searches for a pattern that represents two concurrent events e_i and e_j whose directions intersect each other. A match to this pattern signifies that the system is in an unsafe state. Similar patterns can also be used to identify mutual-exclusion

violations in a distributed application even if the actual global state observed by the system is correct [Schwarz and Mattern, 1994].

2.5.1 Event Predicates

Early work on event-predicate detection include the *event description language* (EDL) proposed by Bates and Wileden [Bates and Wileden, 1983] and Miller and Choi's definition of a class of *distributed predicates* [Miller and Choi, 1988]. These works introduced the notion of event patterns as a way of describing system behaviour. They used constraints on the attributes of an event and grouped collections of *primitive events* to form *compound event*s that provide an abstract view of the system. Since the ordering of events is based upon the time of arrival, concurrent events cannot be expressed and the notion of precedence is limited to the order of arrival.

Haban and Weigel addressed the detection of the causal relation between arbitrary events using vector time [Haban and Weigel, 1988]. An important aspect of their algorithm is that the compound events were assigned the timestamp of the last primitive event detected as part of it. Assigning a single timestamp to a compound event essentially negates its notion of non-zero duration. This can result in ambiguous or even incorrect precedence relationships between compound events [Schwarz and Mattern, 1994].

2.5.2 Causal Ordering for Compound Events

The question of how to specify the occurrence of *compound events* (non-empty sets of primitive events) so that their causality relationship can be intuitively extended from the relations between primitive events is a central one to event-predicate detection. The causality relationship between compound events, in effect, is defined by the causal relations between their constituent primitive events. This leads to the concepts of *strong* and *weak* precedence of compound events, first defined by Lamport [Lamport, 1986]. The two definitions differ in the number of constituent primitive events that are required to be related. Strong precedence is an all-to-all relationship while weak precedence is any-to-any.

Definition 3: ***Strong Precedence***
For any compound events A and B,

$$A \rightarrow B \iff \forall a \in A, \forall b \in B : a \rightarrow b$$

This definition has some desirable properties: it is irreflexive, antisymmetric, and transitive, which are similar to the happened-before relation for primitive events. It poses a problem, however, in defining concurrency between two compound events. Concurrency between primitive

events is defined as their being unrelated by the precedence relation. If a similar definition is used for compound events, two compound events can be concurrent while some primitive events in one compound event precede some primitive events in the other, which is clearly counterintuitive. To avoid this difficulty, concurrency would need to be defined explicitly, as all primitive events in one compound event to be concurrent with all primitive events in the other. This definition, however, would leave many pairs of compound events as being neither predecessors nor concurrent [Basten et al., 1997]. This observation inspires another definition of causality among compound events.

Definition 4: ***Weak Precedence***
For any compound events A and B,

$$A \to B \iff \exists a \in A, \exists b \in B : a \to b$$

Unfortunately, weak precedence contradicts the partial-order properties because it is possible that, when it is used, a compound event happens simultaneously before and after another primitive or compound event. This problem can be avoided by placing some restrictions on which events can form a compound event. A compound event made up of a set of primitive events, E, is *convex* if and only if $\forall x, y \in E \; \forall z : x \to z \to y \Rightarrow z \in E$. If all compound events are required to be convex then most of the partial-order properties will not be violated although weak precedence will still lack transitivity [Xie, 2003; Taylor and Xie, 2004].

It is, however, still possible to create two disjoint convex events such that each happens before the other. Nichols argued that the causality framework needs to be extended to fully classify all possible pairs of compound events [Nichols, 2008].

Definition 5: ***Overlap***
For any compound events A and B,

$$A \textit{ overlaps } B \iff A \cap B \neq \phi$$

Definition 6: ***Disjoint***
For any compound events A and B,

$$A \textit{ is disjoint from } B \iff A \cap B = \phi$$

Definition 7: ***Cross***
For any compound events A and B,

$$A \textit{ crosses } B \iff (\exists a_0, a_1 \in A, \exists b_0, b_1 \in B : a_0 \to b_0 \wedge b_1 \to a_1) \wedge (A \textit{ is disjoint from } B)$$

These three definitions can be used to define a new operator ($\leftrightarrow$) to recognize *entanglement* of two compound events and to modify the definitions of precedence and concurrence.

Definition 8: ***Entangled***
For any compound events A and B,

$$A \leftrightarrow B \iff A \textit{ crosses } B \vee A \textit{ overlaps } B$$

Definition 9: ***Precedence***
For any compound events A and B,

$$A \rightarrow B \iff (\exists a \in A, \exists b \in B : a \rightarrow b) \wedge A \not\leftrightarrow B$$

Definition 10: ***Concurrent***
For any compound events A and B,

$$A \parallel B \iff \forall a \in A, \forall b \in B : a \parallel b$$

With the inclusion of *entanglement* ($\leftrightarrow$), given any two event sets, A and B, their relationship can be described by exactly one of the four relationships: $A \rightarrow B$, $B \rightarrow A$, $A \parallel B$ or $A \leftrightarrow B$. We use the framework proposed by Nichols and in this work precedence and concurrence are defined by Definitions 9 and 10 above.

2.6 Related Work

Monitoring distributed systems for analyzing program behaviour is a well-known problem. In this section we only review the most relevant works in seven categories: 1) continuous queries over unbounded data streams, 2) event monitoring for distributed systems, 3) offline analysis of program logs, 4) replay-based analysis of distributed systems, 5) dynamic analysis of program behaviour, 6) correctness checking for parallel applications, and 7) online debugging of distributed systems.

2.6.1 Stream Query Processing

A distributed application can be modeled as a set of unbounded data streams that are constantly generating new events. Traditional DBMSs are not designed for rapid and continuous loading of individual data items and thus there has been a considerable amount of research in many aspects of processing continuous queries over unbounded data streams.

Continuous queries were introduced explicitly for the first time in Tapestry with an SQL-based language called TQL [Terry et al., 1992]. Conceptually, a TQL query is implemented by executing a one-time SQL query periodically over the current snapshot of the database and the results of all the one-time queries are merged using set union.

The Stream project provides complete DBMS functionality along with support for continuous queries over streaming data [Arasu et al., 2004a]. CQL (continuous query language) is an expressive SQL-based declarative language for registering continuous queries over streams and relations [Arasu et al., 2003].

SASE is a declarative language with SQL-like syntax, which can be used to filter, correlate and transform events [Agrawal et al., 2008]. A pattern query addresses a sequence of events that occur in order (but not necessarily contiguously) in the input stream and are correlated based on the value of their attributes. The query evaluation model employs an NFA and dedicated buffers associated with the states hold the intermediate matches. Cayuga [Demers et al., 2007] is another pubsub-based SQL-like declarative language that also uses an NFA-based complex event-processing model.

At a high level, database query processing has similarities with event-based pattern search. While a typical database query searches for a set of correlated data that have equivalence between values in one or more columns, in event-based pattern search the correlation is based on precedence relationships between events or sets of events.

The conventional wisdom for stream query processing has been to restrict the pattern-matching queries so as to handle only conjunctive queries with arithmetic comparisons [Arasu et al., 2004b]. Another approach is to constrain the search space by using a sliding window and report only the matches that fall within it [Agrawal et al., 2008]. We address the problem of finding the constituent events that match a pattern limiting only the number of reported matches.

2.6.2 Composite Event Detection

GEM is a declarative rule-based language for monitoring events in communication networks and distributed systems [Mansouri-Samani and Sloman, 1997]. It monitors the system behaviour by collecting a set of *primitive* events, which represent the lowest level of observable activity. It then searches for a user-specified *composite* event that is a sequence of events with *temporal constraint*s. GEM uses *temporal* causality between events and also has a *detection window* for maintaining the event history.

Complex Event Processing explicitly represents event causality using a complex knowledge-base schema. The RAPIDE event-pattern language [Luckham, 2002] is a strongly-typed declarative computing language that provides built-in data types, basic event patterns, pattern operators,

and temporal operators. Our work does not consider semantic effects external to the system and is based on *potential causality* defined by the happened-before relationship.

2.6.3 Log-Based Analysis

A common practice for debugging distributed systems is to collect program logs during their execution and use post-mortem analysis of the collected logs. There is a large collection of work, differing in the way the expected behaviour is defined or the way the fault diagnosis is achieved.

Causal event-pattern matching represents a point of interest using a pattern of causally related events [Nichols, 2008; Xie, 2003]. An event is one of a predefined set of instrumented activities in the distributed application. As the application executes, all the occurrences of these events are logged and stored as partially ordered event data. The offline search routine explores this partially ordered event data to find the specific events that match the given pattern. We use the same system model for identifying faults but our search algorithm is online.

Another common approach is to model the behaviour of a distributed system as a collection of causal paths. Different approaches are used to identify such a path and find anomalous behaviour or analyze performance using the end-to-end latency. Project 5 [Aguilera et al., 2003] treats each distributed component as a black-box and only collects the inter-component communications. In the subsequent offline phase it uses statistical inference to identify the causal path to understand the source of latency.

Pinpoint [Chen et al., 2004] focuses on identifying faults in an e-commerce system. It tracks the client request as it travels through the system and records information about all the components that it uses. A failure detector is used to detect whether the request is successfully completed. Finally, a data-clustering technique is used to identify the components that are highly correlated with the request failures.

Magpie monitors the resource usage of a distributed application throughout various parts of a distributed system. An earlier version [Isaacs and Barham, 2002] used a unique identifier to gather related events of a single request to create its causal path. It created a stochastic workload model that can be used for performance analysis. A later version [Barham et al., 2004] removed the necessity for the unique identifier but relies on programmers to provide a *schema* that describes relationships between events.

Pip [Reynolds et al., 2006] models a causal path as an ordered series of timestamped events on one or more hosts. An application is annotated to generate events and resource measurements during its execution. The log files are post-processed with a *reconciler*, which identifies the path instances. These path instances are then checked against the programmer's expected behaviour.

X-Trace [Fonseca et al., 2007] identifies a causal path related to an application task by inserting metadata with identifiers in the request for the task. This metadata is then propagated along the resultant causal path, as the request is handled through different sub-requests in the multiple network layers involved. X-Trace-enabled devices log the relevant information associating them with their corresponding task identifier, which is then used to identify the causal path and check program behaviour.

The causal-path-based approaches, except for Pip, use statistical inference to find unusual behaviours. Thus they can miss bugs in common paths or incorrectly identify rare but valid paths. With Pip, programmers can specify a desired behaviour and then query the collected event logs to compare the actual behaviour with the desired one. Our work is similar in some sense to Pip as we also rely on programmers to specify a pattern that represents a violation. The main difference is that while we use a partial order of the collected events, all the causal-path-based approaches create a total order of them. Thus the fault detection is based on individual path instances and lacks the ability to reason about a global property, such as mutual exclusion.

We see our work as a complementary tool that can be used alongside post-mortem analysis tools. A user may identify a runtime safety violation using our tool and then restrict offline analysis, for in-depth diagnosis, to the particular components that are involved.

2.6.4 Replay-Based Analysis

Another approach for fault detection is to record the runtime instances of a distributed system and then replay the recorded instance for checking runtime properties.

Friday debugs distributed systems using low-level symbolic debuggers (such as GDB) and extends their functionality with distributed *watchpoints* and *breakpoints* [Geels et al., 2007]. It records all the non-deterministic system calls and uses them to replay the execution so that the same code path is followed. Programmers are also able to write arbitrary Python commands to view and manipulate system state.

Recon allows fine-grained instrumentation of distributed applications that is capable of exposing instruction-level information [Lee et al., 2011]. During normal execution, it records every system call, CPU instruction, or signal. It provides SQL-like queries for debugging operations. A query compiler generates a second heavy-weight instrumented program, which is then executed for replay. During the replay all the system calls are again trapped and an event-log parser is used to retrieve the corresponding events from the earlier recorded log.

The benefit of these approaches is their ability to deterministically replay the previous execution and reproduce a bug. Thus, programmers are essentially able to acquire any runtime state during the replay. Unfortunately, saving and replaying an entire execution of a large system is

prohibitively expensive, both in terms of time and storage. Also, replay-based debuggers are not required to react in a timely manner, as an online monitor needs to.

2.6.5 Dynamic Analysis

Dynamic analysis of program behaviour through instrumentation is also a well-studied field. Often it is focused on identifying specific concurrency errors, e.g., to detect deadlocks [Agarwal et al., 2005; Jula et al., 2008], data races [Savage et al., 1997], message races [Park et al., 2007], or atomicity violations [Wang and Stoller, 2006]. We address the problem of detecting causally related event patterns that are more general in nature and can be used to match various undesired behaviours in a system.

2.6.6 Monitoring Parallel Applications

Various tools have been developed to ensure the correctness of parallel applications. Tools like Marmot [Krammer et al., 2003] and Umpire [Vetter and de Supinski, 2000] intercept MPI function calls during runtime and check the correct usage of these calls and their arguments. Their detected errors fall roughly into three categories: i) violation of the MPI specification, ii) tracking resource usage such as incorrect use of communicators or groups, use of non-portable constructs, and mismatched MPI collective operations, and iii) deadlocks and race conditions. We do not address the first two categories as they are related more to the definition of MPI functions than the causal ordering of events. We can detect deadlocks and race conditions as well as any other violation that can be expressed as a pattern of causally related events.

2.6.7 Online Debugging

Online debuggers such as D^3S [Liu et al., 2008] and P2 [Singh et al., 2006] are more closely related to our work. They monitor global properties in a distributed system by collecting global snapshots. Both of them use a total-ordering based on temporal causality which may indicate a potential ordering relationship when none is present. We use vector clocks to encode potential causality between events, based on a partial order, which solves that problem [Birman, 2005].

A solution also exists for online causal-event-pattern matching that uses a sliding window for discarding the partial matches [Fox, 1998]. We only limit the number of matches that are reported, without putting any restriction on the search-domain. Our system model is also different as he used *strong precedence*, but (as discussed in Section 2.5.2) we use *weak precedence* for compound events.

Chapter 3

Online Causal-Event-Pattern-Matching[1]

The most important property of a program is whether it accomplishes the intention of its user.

C.A.R. Hoare

Detection of event patterns for monitoring distributed systems requires an anticipation of the system's appropriate behaviour. If the correct order in which the events should occur within a system can be specified, then detecting the violation of this relative causal order would also signify that the resulting system state is incorrect.

In this chapter, we first define the language that we use to build a pattern representing a set of causally related events. We discuss different approaches that can be used to build a monitoring tool and introduce our first matching algorithm, OCEP. We introduce an existing tool POET that we used for monitoring instrumented events from a target system. We evaluate the performance of OCEP using some representative concurrency-bug patterns from real-world applications.

3.1 Pattern Language

3.1.1 Specifying an Event

An *event* is an activity of interest in the monitored application. It is the smallest building block of a pattern and as such is also called a *primitive event*. We specify a class of events in our pattern language as a 3-tuple:

$$class\text{–}id := [process,\ type,\ text]$$

[1]This chapter is an extended version of our publication in ICDCS '13 [Pramanik et al., 2013].

Classes are assigned *id*s which are later used to form *compound events*. The attributes can be specified by providing the *process* on which the event occurs, the *type* of the event, and a *text* field. These attributes can be specified for an exact match, left empty as a wild-card or used as a variable to enforce equality comparison in an operator. It is possible to further extend the definition of a class by providing a list of name-value pairs specific to a target environment [Slauenwhite, 2007].

Our algorithm is built on top of an existing tool, POET (Section 3.4.1), which monitors instrumented events from a target system and can send them to a client as a linearization of the partial order. POET stores the events grouped by trace, where a trace is equivalent to any relevant entity with sequential behaviour, such as a process or a thread, but may include passive entities such as an object or a communication channel.

In order to determine the causal relationship between two events we use vector timestamps [Fidge, 1991; Mattern, 1988]. As we explained in Section 2.2.2, given two events a (on P_i) and b (on P_j) and their timestamps V_a and V_b, we can find if a happens before b using the following equation,

$$a \rightarrow b \iff V_a[i] < V_b[i]$$

At most two integer comparisons are needed to check whether a happens before b or b happens before a. If neither $a \rightarrow b$ nor $b \rightarrow a$ holds, two more integer comparisons between process numbers and event numbers are needed to distinguish between equality and concurrency.

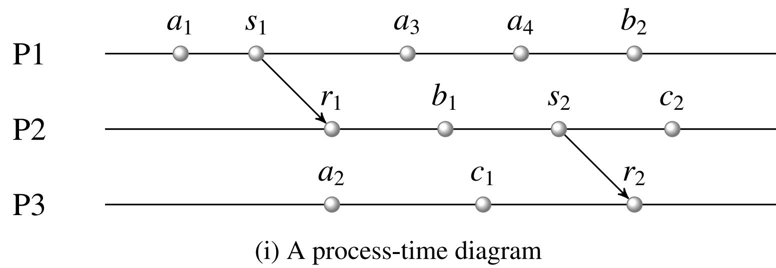
3.1.2 Specifying a Pattern

Event classes are used with *operator*s and *connector*s to build a *pattern* representing a *compound event*, which is a non-empty set of causally related primitive events. In the rest of the document we have used uppercase letters for classes of events specified in the pattern and lowercase letters for specific occurrences of matches to an event class. Compound events are represented by uppercase letters in boldface. For example, a compound event $\mathbf{M}$ can be written as a pattern $A \rightarrow B$ and can be used to find pairs a, b where a matches the specification of event class A, b matches the specification of event class B, and $a \rightarrow b$. Figure 3.1 lists all the operators in our pattern language. In the figure, a and a' match event class A and b matches event class B.

The two operators precedence and concurrence are defined by Definitions 9 and 10 in Section 2.5.2. When a pair of events indicates a synchronous or asynchronous communication between two traces, one event is called the *partner* of the other. The last operator, *limited*, is similar to precedence except it limits what type of events can occur between the two matched events. A match to $A \xrightarrow{lim} B$ must have a matched a that precedes b, but there cannot be another match to class A that happens after the matched a but before the matched b.

Operator	Meaning
$A \rightarrow B$	Event a happens before event b
$A \parallel B$	Event a is concurrent with event b
$A \,.\, B$	a and b are partner events in a point-to-point communication
$A \xrightarrow{lim} B$	a happens before b and $\nexists a' : a \rightarrow a' \wedge a' \rightarrow b$

Figure 3.1: Causality operators for patterns



(i) A process-time diagram

```
S := ['', SEND, '']
R := ['', RECEIVE, '']
A := ['', EVENT_A, '']
B := ['', EVENT_B, '']
C := ['', EVENT_C, '']
```

(ii) Event-class definition

Pattern	Matches
$A \rightarrow B$	(a_1, b_1), (a_1, b_2), (a_3, b_2), (a_4, b_2)
$B \parallel C$	(b_1, c_1), (b_2, c_1), (b_2, c_2)
$S \rightarrow R$	(s_1, r_1), (s_1, r_2), (s_2, r_2)
$S \,.\, R$	(s_1, r_1), (s_2, r_2)
$A \xrightarrow{lim} B$	(a_1, b_1), (a_4, b_2)

(iii) Corresponding example patterns

Figure 3.2: Process-time diagram showing some example patterns

Consider the event classes defined in Figure 3.2 with respect to the process-time diagram. Event class S, for example, will be matched by any *SEND* event in the system (s_1 and s_2). We have shown some example patterns and their corresponding matches in Figure 3.2(iii). The difference between precedence and the limited operator becomes clear here. There are four matches to the pattern $A \rightarrow B$ but only two matches to the pattern $A \xrightarrow{lim} B$. For example, (a_3, b_2) matches $A \rightarrow B$, but it cannot match $A \xrightarrow{lim} B$ because of the event a_4. It is tempting to use the pattern $S \xrightarrow{lim} R$ to define the *partner* operator but notice that the *limited* operator does not require the events to be two ends of a communication. In Figure 3.2(i), if another message is received (r_3) at process P_2, then (s_2, r_3), which are not partners, will match the pattern $S \xrightarrow{lim} R$.

Figure 3.3 shows the complete CFG we use for defining a pattern, which we derived from the work of Nichols. We would like to add that a richer language exists that includes other more elaborate operations, which was used by the earlier off-line algorithms [Jaekl, 1996; Nichols, 2008]. A reduced pattern language is used here because of the major challenges in performing pattern detection online.

$$
\begin{array}{lcl}
\textit{pattern-def} & \Rightarrow & \textit{defs} \;\; \textit{decls} \;\; \textit{predicate} \\
\textit{defs} & \Rightarrow & \textit{defs} \;\; \textit{class-def} \\
 & | & \textit{class-def} \\
\textit{class-def} & \Rightarrow & \textit{id} \; \mathbf{:=} \; \textit{class} \\
\textit{decls} & \Rightarrow & \textit{decls} \;\; \textit{var-decl} \\
 & | & \epsilon \\
\textit{var-decl} & \Rightarrow & \textit{id variable} \; (\mathbf{,} \; \textit{variable})* \\
\textit{predicate} & \Rightarrow & \textit{id} \; \mathbf{:=} \; \textit{pattern} \\
\textit{pattern} & \Rightarrow & \textit{term operator term} \\
 & | & \textit{term connector term} \\
\textit{operator} & \Rightarrow & \rightarrow \\
 & | & \| \\
 & | & \bullet \\
 & | & \xrightarrow{lim} \\
\textit{connector} & \Rightarrow & \wedge \\
 & | & \vee \\
\textit{term} & \Rightarrow & \textit{id} \\
 & | & \textit{variable} \\
 & | & (\textit{pattern}) \\
\textit{class} & \Rightarrow & [\mathbf{process}, \; \mathbf{type}, \; \mathbf{text}] \\
 & | & [\mathbf{process}, \; \mathbf{type}, \; \textit{variable}] \\
 & | & [\textit{variable}, \; \mathbf{type}, \; \mathbf{text}] \\
 & | & [\textit{variable}, \; \mathbf{type}, \; \textit{variable}] \\
\textit{variable} & \Rightarrow & \$\textit{id} \\
\textit{id} & \Rightarrow & \mathbf{alpha} \; (\mathbf{alnum})*
\end{array}
$$

Figure 3.3: Grammar for specifying a pattern

3.1.3 Variable Binding

A variable allows us to use a single matched event multiple times inside a pattern. Variables are declared as belonging to a specific event-class and can be used in place of the class-id.

If we build a pattern such as $(A \rightarrow B) \wedge (A \rightarrow C)$, the pattern does not specify that the two occurrences of event-class A need to match the same event a. For example, in Figure 3.2, (a_4, b_2, a_2, c_1) is a match to this pattern as a_4 and a_2 separately match the two constraints. We can use *variables* in the pattern to specify that once a matched event is *bound* to a variable, the same matched event must match at all the occurrences of that variable in the pattern.

```
class-A := [‘’, EVENT_A, ‘’]
class-A $A
B := [‘’, EVENT_B, ‘’]
C := [‘’, EVENT_C, ‘’]
pattern := ($A → B) ∧ ($A → C)
```

A pattern definition consists of class definitions, declaration of variables (if any), and then the pattern itself. For the above pattern, $\$A$ defines a variable of *class–A* and once bound to a matching event a, it remains the same for the rest of the pattern. In Figure 3.2, a_1 is the only event that satisfies both constraints and so the matches to this pattern are (a_1, b_1, c_2) and (a_1, b_2, c_2).

3.1.4 Attribute Variables

We have extended the notion of variable binding to the attributes inside the definition of an event class. The concept of binding is similar to a variable, that is, all instances of an attribute variable in a pattern must bind to the same value.

```
A := [$proc, EVENT_A, ‘’]
B := [$proc, EVENT_B, ‘’]
pattern := (A → B)
```

In the above pattern we have the process attribute bound to variable $*proc* and so a matched b must happen on the same process on which a matched a was found.

An event's *text* field can be used for additional information about the event itself. For a communication event we have used this attribute to store information about the partner process. In the following pattern, attribute variables will ensure the two matched *SEND* events are sending messages to each other. Since they are also concurrent, two blocking send messages that match the following pattern cause a deadlock.

```
Send1 := [$p1, SEND, $p2]
Send2 := [$p2, SEND, $p1]
pattern := (Send1 || Send2)
```

3.2 Online Event-Pattern Monitoring

There are two main aspects of a monitor that dictate its efficacy for online use:

Event history: It is obvious that some type of history is necessary in order to monitor system behaviour that occurs over time. Since the objective of our work is to identify the constituent events of a pattern, we must store the individual events that match the pattern.

Reporting the matches: The set of all possible matches is the intuitive choice when reporting the matches for a given pattern. For a long-running program, however, the number of matches can be substantial and may not be what a human user is looking for. A representative subset of all possible matches can give useful insights into the situation while keeping the amount of data manageable.

We first consider an algorithm that *stores all* the matched events in its history and *reports all* the matches to a given pattern. Suppose we have n processes in the monitored application and we are looking for a k-length event pattern. If the average number of matched events on each process is M, the space complexity for such a monitor is $O(Mn)$. A simple backtracking algorithm that finds all the matches will have k levels and the size of the domain for each level is M. The time complexity for finding all the matches with such an algorithm is $O(M^k)$ [Horowitz and Sahni, 1978].

Event-pattern search executes on a set of potentially unbounded processes and thus the amount of memory required to report all possible matches may also grow without bound. We also have to report these matches in a timely manner to make an effective online monitor.

In Section 2.6 we discussed existing approaches to tackle this problem by limiting the pattern's expressiveness or its event domain. A restricted pattern with aggregate operators (e.g., average, count, and maximum) can provide valuable information but it only provides a summarized view of the system. If a system requirement can be specified as a pattern of events, finding the specific set of events that violates this pattern and where they occur provides further insight into the behaviour of the system. On the other hand, the sliding-window-based approaches that restrict the event domain are susceptible to omission problems as when no matches are found it can be because a match spans multiple windows.

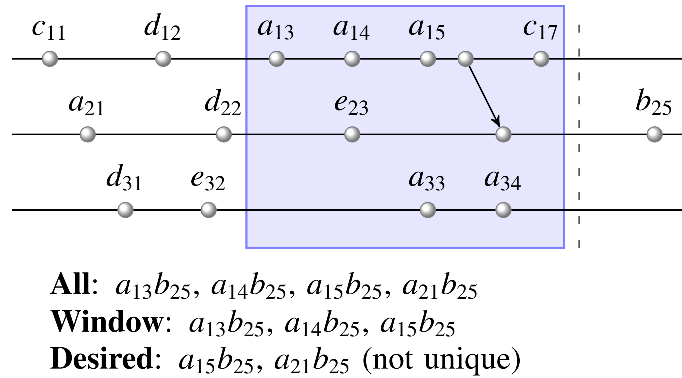
Instead of restricting the expressiveness of the pattern itself we put a limit on two aspects: event history and reported matches. Our first algorithm (*OCEP*), discussed in this chapter, defines a *representative subset* of matches and stores a subset of the event history that allows us to report this particular set of matches. Our second algorithm (*Ananke*), discussed in Chapter 5, is an improvement on OCEP that employs user-defined rules for event removal.

3.3 Online Causal-Event-Pattern Matching Framework

3.3.1 Specifying a Representative Subset

OCEP reports a representative subset of all matches that spans the entire execution time. A good representative subset should provide maximal information about the matches to the pattern. Each event in a pattern is an instrumented activity of interest occurring on a process. So it is plausible to assume that it is important for the user to know whether such an event has occurred anywhere in the system.

A representative subset should also be low in redundancy, so we can exclude multiple occurrences of similar events on the same process.



All: $a_{13}b_{25}$, $a_{14}b_{25}$, $a_{15}b_{25}$, $a_{21}b_{25}$
Window: $a_{13}b_{25}$, $a_{14}b_{25}$, $a_{15}b_{25}$
Desired: $a_{15}b_{25}$, $a_{21}b_{25}$ (not unique)

Figure 3.4: Choosing a representative subset for $A \to B$

Considering these two objectives, we propose a subset that has *at least one occurrence per process for each primitive-event class*. Figure 3.4 shows a simple process-time diagram where the dotted vertical line is the *current cut*. On arrival of the new event b_{25}, there are four matches for the pattern $A \to B$.

Figure 3.4 also shows a sliding window and the set of matches that it will report. We chose the window to have n^2 events where n is the number of processes. If we look at the reported matches, it fails to return a match that involves an event a on P_2 ($a_{21}b_{25}$). Thus the returned subset is not a proper representative of the set of all matches. Considering the pattern as a safety condition, the action that we take based on the returned subset will be incomplete as it misses the matches that span beyond the maintained window.

Our representative subset will report if any of the constituent events in the pattern has occurred on any of the processes and is part of a complete match. Thus if there is only one match, it will definitely be reported. If there are many matches, it can be proved that there exists a subset

according to our definition that has cardinality of at most kn. Here k is the number of events in the pattern and n is the number of processes.

Theorem 1. *A minimal subset defined as 'at least one occurrence per process for each primitive event' cannot have more than kn matches, where k is the number of primitive events in the pattern and n is the number of processes.*

Proof. Let S be the defined subset of matches. The first match of the pattern that is added to the subset S will ensure the *occurrence* of each primitive event on exactly one process. Each subsequent match that is added to this subset must have at least one constituent event on a process not previously included, i.e., there is no existing match in S which has the same event on the same process. After adding the first match, there are $n-1$ additional processes for each of the k primitive events. So the cardinality of the minimal subset is $k(n-1)+1$. □

3.3.2 Pattern Tree

We use a tree-based mechanism because a tree closely matches the causality structure specified in a pattern. The specified pattern is first parsed to create a *pattern tree* as shown in Figure 3.5. The leaf nodes represent the primitive events in the pattern and the internal nodes represent operators and connectors.

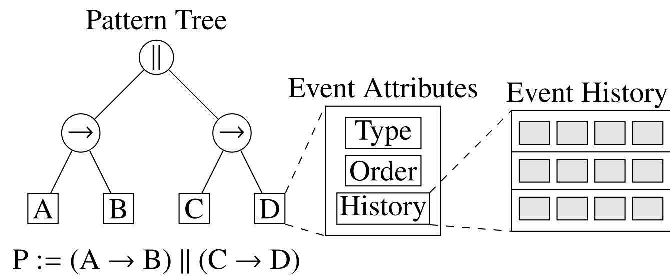


Figure 3.5: The structure of the pattern tree

Each leaf node has three main attributes:

- *Type* specifies the event class for the primitive event.
- *Order* defines the order of evaluation.
- *History* is the list of matched primitive events grouped by traces.

Every time POET reports an event it falls into one of three categories with respect to a given pattern:

1. Events that do not match the pattern.
2. Events that match the pattern but will not generate a complete match.
3. Events that can possibly generate a complete match, which we call *terminating event*s.

For the pattern $A \rightarrow B$, only a newly arrived b is a terminating event as it can generate a complete match if a causally related a is already found. In contrast, for the pattern $A \parallel B$, any newly arrived a or b is a terminating event.

When a reported event matches a leaf node of the pattern tree, it is added to the corresponding leaf node's *history* of events. This history is grouped by traces and is totally ordered for each individual trace. If the event is also a terminating event, the OCEP algorithm is triggered, which tries to build a complete match at the root in a bottom-up fashion. OCEP uses backtracking and for each terminating event it visits the other leaf nodes in a static causal order which is stored in the field *order*. Thus the execution time of the matching algorithm is only affected by the events that are actually in the pattern, not by all the events that are being monitored.

3.3.3 Maintaining Event History

A problem with partial matches is that even for simple patterns, there are cases in which it is always possible that future events can make the partial match bigger. Consider the simple predicate $(A \rightarrow B) \rightarrow C$ that could be used in monitoring safety properties. For example, the action that is being monitored occurred (a), some measures are taken (b), and an anomaly is detected (c). Thus there will potentially be numerous matches of a and b while the occurrence of c will be infrequent. All of these partial matches for $a \rightarrow b$ have to be stored in memory waiting for the arrival of a c. Unfortunately a future c can occur anywhere and thus none of those partial matches for $a \rightarrow b$ can ever be discarded.

Storing a history of all the matched events since system start-up is constrained by the available storage space. We discussed in Section 3.2 that the time complexity for finding all matches from such a history is $O(M^k)$. As we are reporting a subset of all matches it is paramount that we discard redundant events so as to limit the time and space required.

Since we do not search for all possible complete matches, we need some way to identify a subset of the matched events that covers the set of all matches in terms of our representative-subset definition. We take a very simple approach that only looks at the presence of messages

between occurrences of the same event on a trace. As we are dealing with potential causality, how an event is causally related to the other events is only affected by messages. Thus if we have multiple occurrences of the same event on a trace with no send or receive events between them, their causal relation with events on other traces is the same.

This approach is computationally simple, $O(1)$, but does not guarantee a bounded subset size. In the worst case it will still store all the events since start-up.

3.3.4 OCEP Algorithm

OCEP uses *backtracking* to find the subset of matches to the pattern, as shown in Algorithm 1. We start with the newly matched event as our initial assignment (e_1). At every stage of backtracking, the algorithm tries to extend the partial match by finding a match to the event e_i ($goForward$) so that it is causally related to the existing assignment ($e_1, e_2, \ldots, e_{i-1}$) as specified by the pattern. If a complete match is found with an event e_i on trace t or there is no unexplored match on it, the algorithm continues with matches for e_i on trace $t+1$. If there is no unexplored match on any of the traces the algorithm backtracks (*goBackward*). The variable *consistent* controls the direction of backtracking by keeping track of the return value from the two traversal functions. When a complete match is found, the function $updateSubset$ adds the match to the subset of matches that will be reported to the user and performs some bookkeeping to track the number of processes on which a match has already been found.

Algorithm 1 OCEP Algorithm

Precondition: M is a partial match of length one: $\{e_1\}$

```
1: level ← 2                          ▹ level and M are updated inside called functions
2: consistent ← true
3: while level ≠ 1 do
4:     if consistent then
5:         consistent ← goForward(M, level)
6:     else
7:         consistent ← goBackward(M, level)
8:     if M is a complete match then
9:         updateSubset(M)
```

It is obvious that this type of pattern search corresponds to a depth-first traversal of the search tree. The root of the search tree is the initial assignment and the ith level has $(i+1)$-tuple nodes which represent the partial solutions at i^{th} stage. In such a depth-first traversal, the ordering of

events that are chosen at each stage is often done in a way that constrains the later choices the most [Bitner and Reingold, 1975].

In a causal pattern, the possibility of extending a partial match depends on finding an event that is causally related with the events already in the partial solution. For example let us assume we have a pattern $A \rightarrow (B \rightarrow C)$ and our initial assignment is (c). It might happen that a seems better as the candidate event for the next stage as it constrains the choices for B more, but if there is no matching b that precedes the event c, constraining the later choices does not generate a better search space. For this reason we have used a causal static ordering of events for the backtracking stages which is stored in the *order* field at each leaf node in the pattern tree.

goForward

A very basic implementation of $goForward$ can use chronological backtracking, which will start with the latest match on a trace and chronologically go back in time. This approach is not very efficient in practice as it explores the entire search space until a solution is found or a conflict is reached. A conflict in this context is the absence of a match at level i because of some previously instantiated event.

When dealing with a causal pattern, given a partial match $(e_1, e_2, \ldots, e_{i-1})$, it is possible to use the causality relationship of the instantiated events to restrict the domain of event e_i on any trace. This technique is similar to the *forward checking* algorithm, which uses the instantiated variables to restrict the domains of all future variables [Prosser, 1993]. In contrast, we are restricting only the domain of the variable that is currently being instantiated.

The pseudo-code for the function *goForward*, which instantiates an event in each backtracking level, is given in Algorithm 2. The variable i is the passed backtracking level. The variables $trace_i$ and D_i store the trace that is currently being traversed and the domain of matched events on it. The function starts by initializing the domain D_i by restricting it using the already instantiated events.

If a conflict is found, i.e., a previously instantiated event constrains D_i to empty, *getTS* determines the desired timestamp for the event at the previous level that can possibly resolve this conflict, which is then saved in *bt* for possible use by *goBackward* (line 7). If a non-empty domain is found, i.e., all previously instantiated events conform to a non-empty D_i, it instantiates the latest event in the domain as the matched event at level i and goes forward to the next level $(i + 1)$ (line 16).

After all the traces are tried or a conflict is found, it returns $false$ and so *goBackward* is called for the next round in Algorithm 1. It uses the recorded timestamps in *bt* to *backtrack* to the closest event that can resolve one of the conflicts, if any occurred. In the presence of

Algorithm 2 Forward Algorithm

Precondition: M is a partial match of length $i-1$, i is the backtracking level, and n is the total number of traces.

```
1:  function goForward(M, i)
2:      if D_i is not initialized then
3:          while trace_i < n do
4:              for prev ← 1 ... i − 1 do
5:                  D_i ← restrictDomain(M_prev, trace_i)
6:                  if D_i = ϵ then
7:                      bt[prev][i] ← getTS(M_prev, trace_i)
8:                      break
9:              if D_i = ϵ then
10:                 trace_i ← trace_i + 1      ▹ Empty domain. Continue on next available trace.
11:             else
12:                 break                      ▹ Non-empty domain found on trace_i. Go forward.
13:     if D_i ≠ ϵ then
14:         M_i ← nextMatch(D_i, trace_i)
15:         i ← i + 1
16:         return true
17:     else
18:         return false
```

a conflict, a simple backtrack to the previous event could cause repeated failure from a single conflicting event.

restrictDomain

The function *restrictDomain* is used to constrain the domain of an event e_i on a trace (D_i) using its causal relationship to a previously instantiated event (M_{prev}). We first define two special events (GP and LS) that are used for this purpose.

Definition 11: *Greatest Predecessor*
The *greatest predecessor* of an event, e, on a trace t, denoted by $GP(e, t)$, is the most-recent event, a, on that trace that happens before e.

Definition 12: *Least Successor*
The *least successor* of an event, e, on trace t denoted by $LS(e, t)$ is the least-recent event, b, on that trace that happens after e.

An interesting property of $GP(e,t)$ is that it shows in which parts of the process t an event a can occur so as to happen before e. Similarly $LS(e,t)$ shows in which portion of the process t an event b can occur so as to happen after e. When used together, $GP(e,t)$ and $LS(e,t)$ can also identify the portions with which e is concurrent.

Suppose we are trying to instantiate the event e_i on trace l that is concurrent with event e on trace m, which is already instantiated. The earliest event on trace l that can be concurrent with e is the event that happens immediately after its GP on trace l. The latest concurrent event is the event that occurs immediately before its LS on trace l. Thus the domain of e_i on l that can extend the partial match with respect to e is given by $(GP(e,l), LS(e,l))$. The open interval denotes that the GP and LS themselves are not included in the domain. We summarize how we restrict the domain for the operators in Figure 3.6.

Causality	Domain
$e \parallel e_i$	$(GP(e,l), LS(e,l))$
$e \rightarrow e_i$	$[LS(e,l), \infty)$
$e_i \rightarrow e$	$(-\infty, GP(e,l)]$

Figure 3.6: Restricting domain of e_i with respect to e

getTS

Suppose e_i has an empty domain on trace m with respect to the instantiated event e_2 on trace l. The vector timestamps of the conflicting events can then be used to update the domain at the earlier level during backtracking. This is done in the function *getTS* and we illustrate its operation in Figure 3.7 for each of the causal relations. The two sides of a precedence pattern, $A \rightarrow B$, are related differently to each other and we name them *precedence* and *follow*. For simplicity we have shown a truncated process-time diagram and fragmented vector timestamps that only show entries for the relevant traces.

If we are looking for $e_2 \rightarrow e_i$, then the conflict happens because $LS(e_2,m)$, if it exists, happens after the latest e_i on m, Figure 3.7(i). The l-th entry in the vector timestamp of e_i (t_i^l) indicates $GP(e_i,l)$. Any matched e_2 that happens after $GP(e_i,l)$ will again lead us to the same conflict. So t_i^l is stored for later use and during backtracking, *updateDomain* uses this to restrict e_2's domain to $(-\infty, GP(e_i,l)]$.

If the conflict is for $e_i \rightarrow e_2$, $GP(e_2,m)$ happens before the earliest e_i on m, Figure 3.7(ii). Thus we can prune all the matches for e_2 on l as none of them will be able to create a complete match.

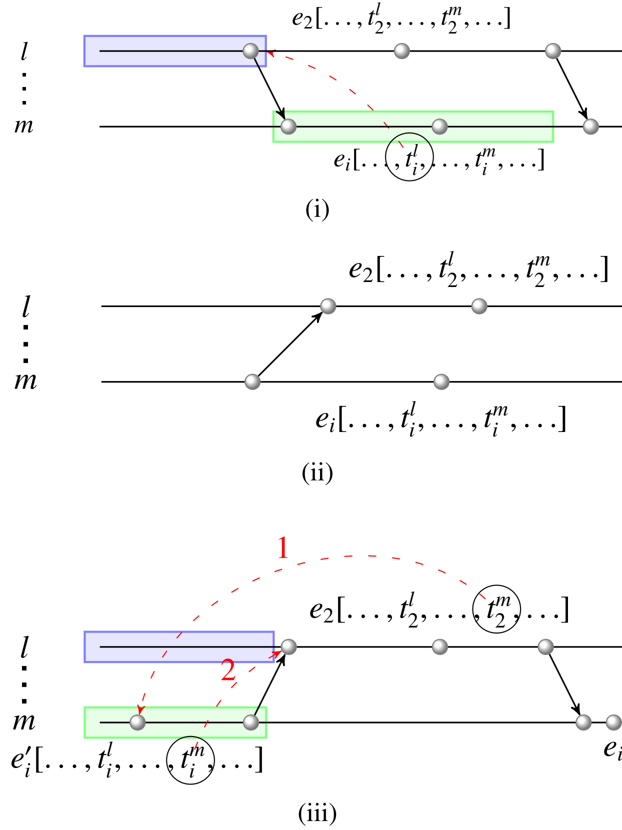


Figure 3.7: Using causality to update domain when backtracking: i) Precedence, ii) Follow, iii) Concurrency

When we have a conflict for $e_2 \parallel e_i$, Figure 3.7(iii), all the matches for e_i on m either happen after e_2 or happen before it. So e_2 cannot create a complete match with an e_i on trace m but an earlier e_2' on trace l may still create one. Then the domain of e_i on m that can be of possible interest is $(-\infty, GP(e_2, m)]$. We can use t_2^m to find $GP(e_2, m)$, say e_i'. The backtracked level then needs to restrict e_2's domain to $(-\infty, LS(e_i', l))$ and we can use t_i^m to find $LS(e_i', l)$.

goBackward

The pseudo-code for the function *goBackward* is given in Algorithm 3. It first tries to determine if any conflict with the uninstantiated events is recorded for the instantiated event at this level. Recall that any uninstantiated event will record this information while executing *goForward* at its corresponding level (Algorithm 2, line 7). If any conflict exists, the function *getClosest*

Algorithm 3 Backward Algorithm

Precondition: M is a partial match of length $i - 1$ and i is the backtracking level

```
1: function goBackward(M, i)
2:     jumpTS ← getClosest(i, bt[i])
3:     if jumpTS exists then
4:         D_i ← updateDomain(i, jumpTS)
5:     else
6:         i ← i − 1
7:     return true
```

finds the one that will make it backtrack to the latest match in the current domain. It extracts the required timestamp and uses it to restrict its domain. If no conflict was found, it simply backtracks to the previous level.

The following helper functions are used by the main search algorithm, but are simple enough not to require presentation as pseudocode.

nextMatch

This function is called at backtracking level i when there is a non-empty domain D_i on $trace_i$ (Algorithm 2, line 14). At every call, it returns the next matched event in the existing domain.

getClosest

For any backtracking level i, $bt[i][j]$ will contain the return value of $getTS$ that can help resolve the conflict between the levels i and j. When we call the function $getClosest$ at level i (Algorithm 3, line 2), we want to backtrack to the latest match in the current domain D_i that can resolve one of the conflicts. Thus, it simply returns the maximum value in $bt[i]$.

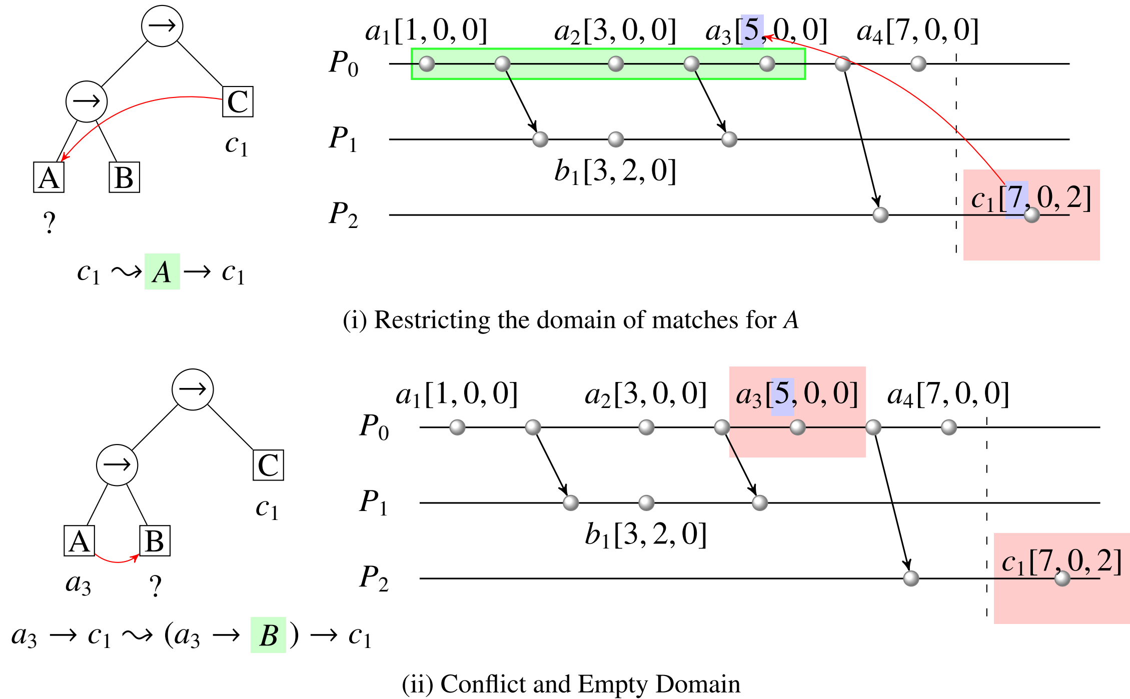
updateDomain

This function is used to update the domain D_i for a backtracking level i (Algorithm 3, line 4). Recall that $jumpTS$ is essentially a return value of $getTS$ and so we need to find the latest event in the domain D_i that has its timestamp $T[trace_i]$ less than or equal to $jumpTS$. If such an event exists, it should be the right end of the new domain for the level i. As the events on a single trace are totally ordered by their timestamps, we use binary search to find such an event.

3.3.5 A Simple Example

We use a simple pattern to show the steps of our backtracking algorithm. Suppose we are trying to match the pattern $(A \to B) \to C$. Not every newly arriving matching event will generate a complete match. For example, in this pattern, a new event a cannot generate a complete match as we still have not seen any of its causal successors. Since b and c may be concurrent, a newly arriving b or c can generate a complete match. Thus b and c are *terminating events* for this pattern and the search algorithm is invoked when OCEP receives an instance of b or c.

In Figures 3.8 and 3.9 we show the steps of the OCEP algorithm for a simple process-time diagram with three processes. We have so far seen all the events up to the vertical dotted line and c_1 is the newly arriving terminating event. So OCEP will go through the other leaf nodes in the relative causal *order* stored at the node C. Each level of the backtracking algorithm will try to find a match to the next event in this causal order that conforms to the causality relation specified in the pattern.



(i) Restricting the domain of matches for A

(ii) Conflict and Empty Domain

Figure 3.8: Search example – pruning the search space in *goForward*

At the first level, OCEP tries to find an a that precedes c_1 as shown in Figure 3.8(i). The domain at each level is all the matches stored in a leaf. However, it is possible to use the causal

relation and vector timestamp to prune this search space. If we look at the vector timestamp of c_1, the 0-th entry tells us that among the four matches on P_0, only the first three precede c_1. So we restrict the domain for A to these three events, choose the latest match in the domain for A, and move to the second level.

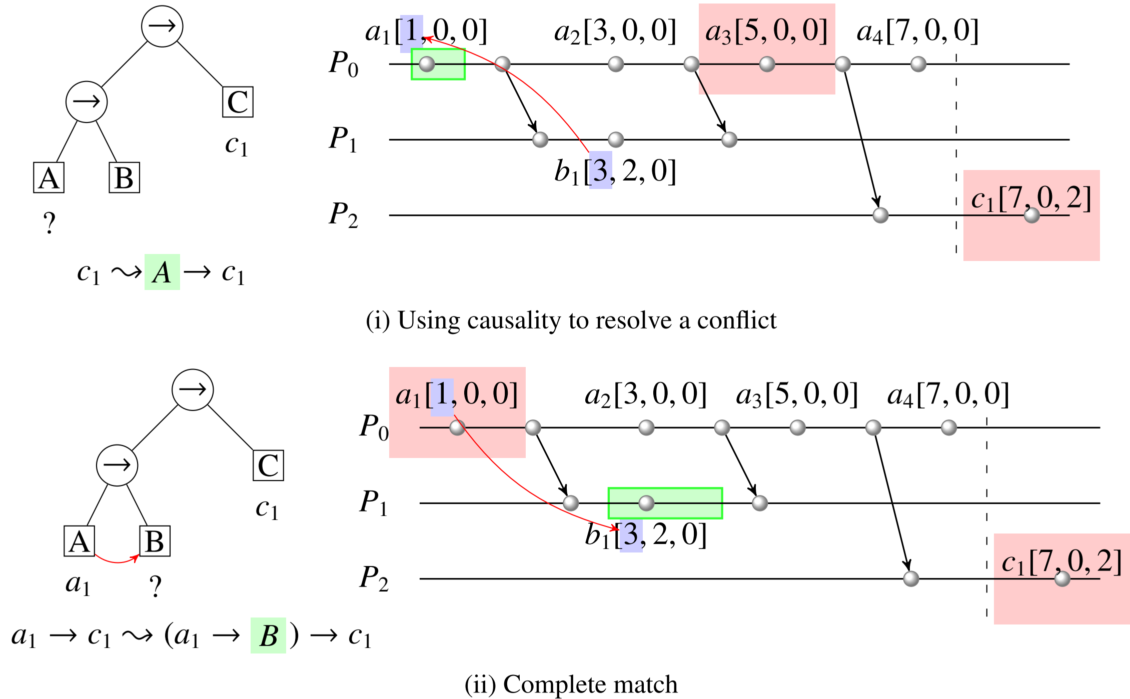


Figure 3.9: Search example – resolving conflict in *goBackward*

We now have instantiated two events in the pattern a_3 and c_1 and at the second level we will try to find a b that follows a_1 but does not follow c_1. The latter condition is required as otherwise $a_3 \rightarrow b$ will be entangled with c_1. If no conforming matches are found in a level, OCEP backtracks and selects a different event at the earlier level. Indeed, we find that the only match for B on P_1 does not conform to these relations. Thus, we have an empty domain at level two and we must backtrack and choose a different match for A.

A simple backtrack would go to the previous level, choose a_2 and then will face the same conflict. Alternatively we can use b_1's timestamp to tell the earlier level how to fix this conflict as shown in Figure 3.9(i). The 0-th entry in the timestamp of b_1 tells us that any match on P_0 that precedes b_1 must have a smaller value in the 0-th entry of its timestamp. This can be used to further restrict the domain for A to $[a_1]$.

When we backtrack to the first level, we now choose the event a_1 as the new match for A. We will then move forward to the second level and this time a_1 will have a non-empty domain for B and the final match will be $(a_1 \to b_1) \to c_1$.

3.4 Performance Evaluation

3.4.1 Partial-Order Event Tracer

This work is built on top of the various techniques and algorithms in an existing tool, the Partial-Order Event Tracer (POET) [Kunz et al., 1997; Taylor, 1999]. This tool allows a user to instrument a distributed application and collect information about it. POET itself is a distributed application and its architecture is shown in Figure 3.10 [Taylor, 1999]. POET is target-system independent, which means that it can collect data from a wide variety of execution (target) environments with a minimum amount of effort in most cases [Taylor et al., 1996].

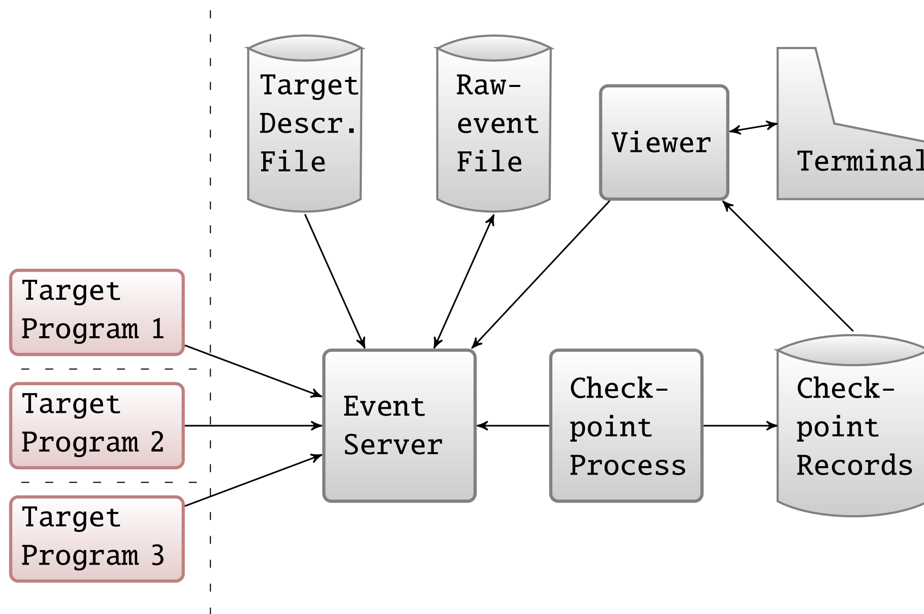


Figure 3.10: Architecture of POET: Server with one local viewer

The occurrence of a POET event indicates that one of a predefined set of important actions has occurred in an application that is part of the system. This set of important actions is entirely dependent on the target environment being used. For example, a TCP-socket target environment would define the important actions to be bind, send, and receive socket calls, among others. In

this thesis, we often use the terms *process* and *trace* interchangeably, but the events collected by POET are actually grouped by traces. A trace is equivalent to any relevant entity with sequential behaviour, such as processes, threads, or even passive entities such as objects or communication channels. A client connecting to POET can receive the arriving events in a linearization of their partial order. Our monitor application connects to POET as one such client and tries to detect the specified pattern as the events appear.

To present this information to the user, POET contains a graphical viewer. The viewer presents the traces as horizontal lines, where time flows from left to right, and relationships between pairs of events that belong to different traces are drawn as vertical or diagonal lines that connect the two events. This is the same as the process-time diagram that we introduced in Figure 2.1 and have used subsequently.

In most cases, the set of events will extend far beyond the boundaries of the screen. Because these events are in a partial order, using a standard horizontal scroll bar to move events when a selected event is repositioned would be inappropriate. This type of scroll would mislead the user into thinking that there is an absolute ordering between all events as the events would always be in the same relative positions. Two concurrent events, for example, might always appear as though one happened before the other.

To avoid presenting a misleading view of the events, a partial-order scroll algorithm was devised [Taylor, 2005]. This algorithm allows a user to scroll through a single trace in a predictable order, and then adjusts the surrounding events in an appropriate manner. One of the goals in adjusting surrounding events is that they should be shifted as little as possible. This means, for example, that if there is no precedence relationship between two traces (i.e., all events on one trace are concurrent with all events on the other trace), when one trace is being scrolled, the other trace will not change. Figure 3.11 shows a screenshot of POET with data collected from a TCP-socket target environment [Nichols and Taylor, 2005]. There are additional events to the right of the display window that can be scrolled to.

POET allows the user to manipulate the view in other ways as well. The ordering of traces can be changed and multiple traces/events can be collapsed into a single trace/event.

3.4.2 Evaluation Methodology

We evaluate the effectiveness of OCEP using some representative bug patterns that we have chosen from the existing literature [Farchi et al., 2003; Lu et al., 2008]. These studies have found that the most-frequently occurring concurrency-related bugs in real-world applications are deadlock, race condition, atomicity violation, and violation of the programmer's intended order. We have simulated one test case for each of these bugs, using an application known to contain

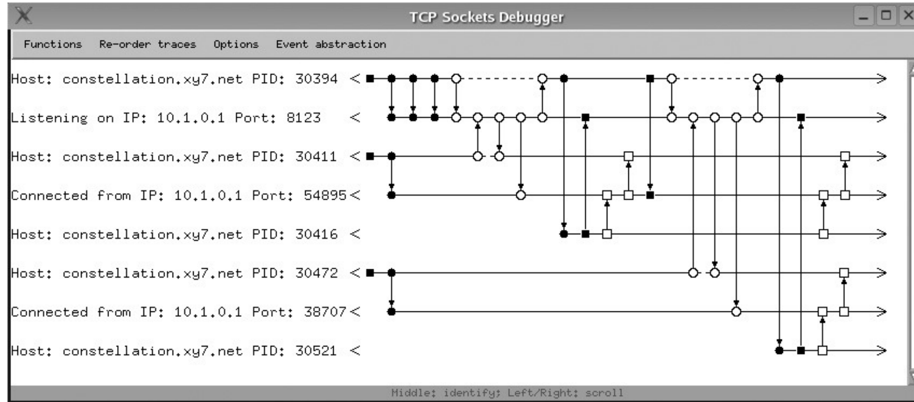


Figure 3.11: POET screenshot

an instance of that bug. The test cases were run on event data collected from μC++ and MPI environments. μC++ [Buhr et al., 1992] extends C++ with additional constructs to incorporate concurrent programming. MPI [Snir et al., 1998] is a widely used library for writing parallel applications.

Each test case is executed until the number of events generated exceeds one million. We also varied the number of processes to generate three different scenarios for each test case. We used the *dump* feature in POET to save the collected trace-event data in a file. The *reload* feature in POET allows us to reuse this file with the saved events passed to POET via the same interface used to collect events from a running application. OCEP connects to POET as a client in order to receive the events in a linearization of the partial order.

Our main performance metric is the *execution time* for a pattern search on arrival of a new *terminating event*. The pattern search finds a representative subset of matches to the given pattern or determines that no such match exists. We use the *reload* feature in POET to execute OCEP five times for the trace-event data of each test case and record the wall-clock time taken by the monitor on arrival of each terminating event. *Execution time* for each terminating event is measured as the average of these runs, which we used for our evaluation. All measurements are performed on a workstation with an Intel® Core™ i5-3320M 2.6 GHz CPU and 8GB memory running Linux kernel 3.5.0.

3.4.3 Case Studies

We use boxplots to show the measured execution time taken for each of our case studies in Figures 3.12-3.15. The centre rectangle spans the *inter quartile range* (IQR), which is the likely

range of variation, with the inner segment representing the median. The whisker marks are placed $1.5 \times IQR$ above the third quartile and below the first quartile, while the crosses mark the outliers. Thus the box itself contains 50% of the test results and the box and the whiskers together represent 99.3% of the execution times [Frigge et al., 1989]. We also chopped the extreme outliers so as to keep the box and whiskers prominent.

Deadlock

In MPI, when a process executes a blocking point-to-point communication, it does not return until it is complete and the buffer can be reused. So an application can deadlock if there exists a send-receive cycle due to incorrect usage of the communication routine. A commonly used method for detecting such a deadlock is to build a dependency graph and check for cycles [Agarwal et al., 2005]. Event patterns are not able to detect a generic cycle as opposed to a dependency graph, but they can be used to identify a deadlock of specific length. A cycle of length three can be detected with the following pattern:

```
Send1 := [$p1, Send, $p2]
Send2 := [$p2, Send, $p3]
Send3 := [$p3, Send, $p1]
Send1 S1
Send2 S2
Send3 S3
pattern := (($S1 || $S2) || $S3)
```

The basic idea here is to identify a set of *Send* events which are concurrent to each other and have a circular dependency. The concurrency is identified by the causal operator in the pattern. We also expose the receiving trace of a *Send* event using the *text* field of an event. Attribute variables are then used to compare the *text* field of a match to the *trace* field of another to identify the existence of a cycle.

We simulate deadlock using a parallel algorithm for *random walk* which has many applications in statistical and scientific computation. It divides a domain among the parallel processes and each process has a number of walkers traversing a contiguous sub-domain. The processes communicate among themselves to exchange the walkers that move across process boundaries. We deliberately leave a deadlock in the code for this point-to-point communication. Interestingly enough, this deadlock is rarely visible as *MPI_Send*, which should block until the message is received, only gets blocked when the network cannot buffer the message completely [Snir et al., 1998].

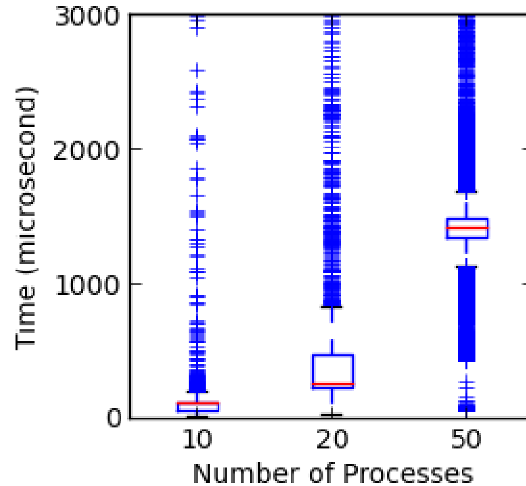


Figure 3.12: Execution time for detecting deadlock

Figure 3.12 shows the execution time for three different scenarios of *random walks* involving 10, 20, and 50 processes. In each case, given the structure of the code, a deadlock involves all the processes. OCEP uses a backtracking algorithm for its pattern search and thus the expected runtime for OCEP is exponential in terms of the length of the pattern. In Figure 3.12, however, we see a linear increase in the execution time with the pattern length. The efficiency of our backtracking algorithm depends on its ability to prune the domain for an event on a trace based on its causality relation with the other events. As we explained in Figure 3.7, we used vector timestamps of the causally related events to constrain the searched domain. In contrast, building and maintaining a dependency graph is costly, which is apparent from the runtime of 35 seconds to detect a cycle of length 30 [Agarwal et al., 2005]. It should be noted that this is not a precise comparison since different hardware is involved. Re-implementation of their algorithm would have required substantial effort and was not attempted.

Message Race

When two or more concurrent messages are sent to the same process they may arrive in a random order, causing nondeterministic execution of a parallel program. This may lead to sporadically occurring errors that are difficult to reproduce. Even when a message race happens by design, it is critical to detect it for debugging in order to ensure that all possible executions of a program are examined [Kranzlmüller and Schulz, 2002].

A common method for detecting message races is to keep track of the *receive* events on a trace and compare their vector timestamps for causality [Netzer and Miller, 1992]. If any two incoming messages to a process are concurrent then the two messages race. A causal-event-pattern can express this pattern, which we use to express message races.

```
Sends := ['', Send, '']
Receives := [$p1, Receive, '']
Sends S1, S2
Receives R1, R2
pattern := (($S1 || $S2) ∧ ($S1 . $R1) ∧ ($S2 . $R2) ∧ ($R1 → $R2))
```

The above pattern tries to match two concurrent *Send* events and their corresponding *Receive* events. Since the event-class *Receives* is defined using $p1$ as its trace-attribute variable, the matches for $R1$ and $R2$ will happen on the same trace.

We use a benchmark program in which all processes but one concurrently send messages to the remaining process while the latter accepts them using a blocking receive with the wild-card *MPI_ANY_SOURCE*.

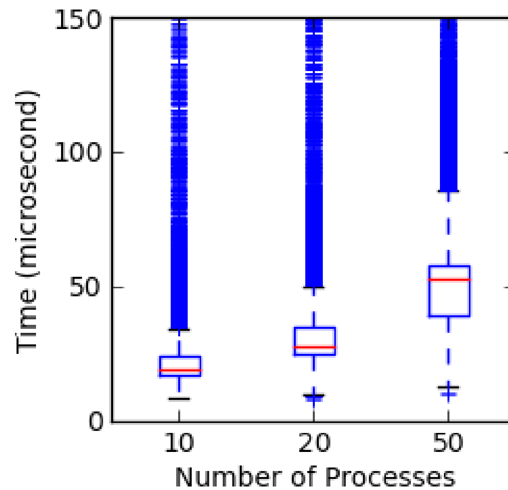


Figure 3.13: Execution time for detecting message races

Existing methods for detecting message races pass the vector timestamps among the processes by attaching them to messages [Park et al., 2007]. OCEP receives a vector timestamp

constructed in POET, not in the application, so there is negligible extra overhead on the application itself [Taylor et al., 1996]. In Figure 3.13 we show that OCEP is able to report the message races for our test cases in less than 100 μsecond in most cases.

Atomicity Violation

An atomic code segment is protected in the sense that when a process executes the first instruction in the segment, no other process can start executing the segment before the first process executes the last instruction. The approaches for detecting an atomicity violation rely on finding unserializable patterns of operations by searching the events that are related to shared-variable access and synchronization primitives [Wang and Stoller, 2006]. Our approach is similar in the sense that we search for a causal pattern among these same events, but we also monitor the synchronization primitives as separate traces, which allows us to represent an atomicity violation as a causal pattern.

We chose the μC++ environment for this case study as a POET plugin for μC++ already adds semaphores as separate traces. It is possible to add a different semaphore implementation, such as a *pthreads* semaphore, by adding additional plugins.

Our μC++ program has a method protected by a semaphore so that there is never more than one thread executing it. There is an intentional bug for which, when a thread attempts to execute the method, there is a 1% probability that the semaphore will not be acquired properly. If the semaphore is acquired properly the event *thread enter*, which denotes a thread starting to execute the protected method, occurring on the two contending processes will have a happened-before relationship ensured by the semaphore. When a violation occurs there will be two concurrent *thread enter* events denoting the absence of synchronization.

```
M1 := [‘’, thread enter, ‘’]
pattern := (M ∥ M)
```

Notice that we do not need a bigger pattern to identify a situation when more than two threads are running the protected method. Because of our representative-subset definition, this same pattern will detect multiple matches in that situation.

OCEP is able to detect a match to the violation pattern for all three scenarios in less than 50 μsecond in most cases. Existing approaches often build a conflict graph with the monitored events and synchronization primitives, which has previously taken 0.4-40 seconds for detecting similar violations [Wang and Stoller, 2006].

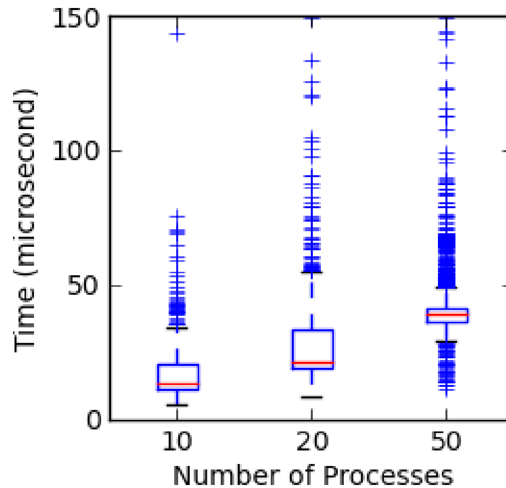


Figure 3.14: Execution time for detecting an atomicity violation

Undesired Order

Studies have found that most non-deadlock concurrency bugs in real-world applications belong to one of two categories: atomicity-violation and order-violation [Lu et al., 2008]. Existing concurrency-bug-detection tools do not address bugs that are manifested by the violation of order implied by the developer.

There are various ways to analyze the expected behaviour of a system. A developer can represent the violation of intended ordering with a pattern of causally related events. Large software systems often use software classification tools to identify repetitive patterns of events from program execution traces [Lo et al., 2009]. Over the last decade, *design patterns* have become one of the most powerful tools in designing large software systems [Gamma et al., 1995]. *Design patterns* are reusable software modules for recurring problems and both formal [Bayley and Zhu, 2010] and informal [Gamma et al., 1995] specifications exist defining the semantics of a pattern.

Ordering bug refers to the situation in which the desired ordering between groups of events is violated. This violation of order is known to be the most common concurrency bug that is not addressed by existing debuggers [Lu et al., 2008]. One example of this type of bug is bug#962 [ZooKeeper, 2010] of ZooKeeper [Hunt et al., 2010]. Zookeeper is a coordination service for distributed processes that achieves high availability through replication. It uses an

active-replication technique where a follower sends synchronization requests to the leader for a snapshot of the system. When a restarting follower sent a synch request to the leader, the leader was not blocked from making an update after it took a snapshot of the system. Thus a restarting follower could occasionally receive inconsistent service-data from the leader.

```
Synch := [$follower, Synch_Leader, $leader]
Snapshot := [$leader, Take_Snapshot, '']
Update := [$leader, Make_Update, '']
Forward := [$leader, Forward_Snapshot, $follower]
Snapshot $Diff
Update $Write
pattern := ((Synch → $Diff) ∧ ($Diff → $Write) ∧ ($Write → Forward))
```

The pattern above tries to detect a situation when a snapshot taken on a *synch* request is followed by an *update* before it gets forwarded to the follower. The variable-binding for the events ensures the proper causal order. It also shows how variables can be used inside the class definition for better precision. The text field can be used for various purposes and this particular pattern is using it to encode the corresponding trace for a Synch/Forward pair.

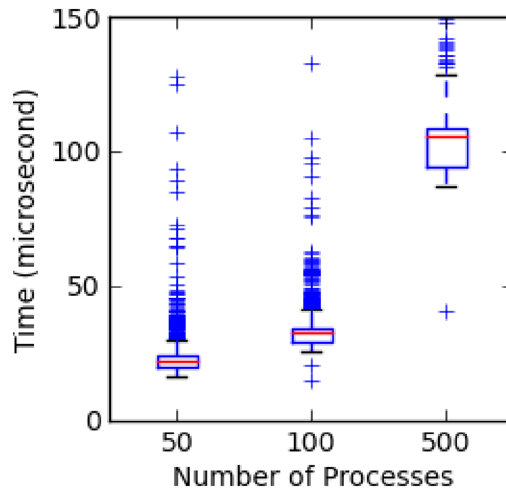


Figure 3.15: Execution time for detecting an ordering bug

We simulate a distributed application that handles a replicated service. We left a deliberate bug in the synchronization procedure so that there is a 1% chance that a leader may make an

update after it takes a snapshot of the system and then it will forward this stale snapshot to the synchronizing follower. Our test cases this time included 50 – 500 traces and OCEP is able to detect an ordering violation in less than 150 μsecond in most cases.

3.4.4 Results and Discussion

The strength of our event monitor lies in its ability to detect the occurrences of a pattern that will include each trace that has a constituent event matching to it. This makes it an excellent tool for identifying violations of safety conditions in a system. The outliers in the boxplots for test cases show that in order to identify such a subset we occasionally have to traverse a large section of the process-time diagram. We limited the number of outliers shown in Figures 3.12–3.15 so that the IQR and whisker marks are clearly shown. We summarize the quartiles, top whisker, and the maximum time taken by an event in Figure 3.16. For each test case, these are the execution times involving the largest number of processes.

Test Case	Q1	Med	Q3	Top Whisker	Max
Deadlock	1339	1419	1480	1692	8009
Races	39	53	58	86	9413
Atomicity	36	39	41	49	3432
Ordering	94	105	108	131	2696

Figure 3.16: Detailed execution time for test cases (μsecond)

The second performance metric that we used is completeness. Since we used known violation cases in simulated applications, we also knew all such occurrences. Our OCEP algorithm is complete as for each test case, it correctly reported a subset of matches that is representative of the set of all matches. OCEP also did not report any false positives for any of the test cases.

The major question that we wanted to answer in our simulation was whether our algorithm is efficient enough to provide fast detection of violation patterns with low overhead in realistic application settings.

We demonstrated this by choosing some of the prevalent concurrency-bug patterns in real-world applications. We find that OCEP is able to find a match to most of the tested violation patterns in less than one millisecond in most cases. The only test case where the whisker marks are above this limit is when we are searching for a long deadlock cycle. As shown in Section 3.4.3, OCEP runs orders of magnitude faster than existing graph-based approaches, when applied to similar problems. The implementations are not publicly available, so a direct compar-

ison is not possible, but the extreme timing differences clearly cannot be explained entirely by changes in computer technology.

Survey results often show that most concurrency bugs involve only a small number of processes [Lu et al., 2008]. A complete match to the message race involves two senders that are sending messages to the same receiver. Atomicity violation involves two clients that concurrently execute the critical section. An ordering bug only involves the leader and the follower. We use variable binding for both an event and its attributes inside the class definition to clearly specify the relationship among the constituent events in the pattern. OCEP can detect a match to the smaller patterns within 150 μseconds in most cases. This signifies that our algorithm was effectively able to isolate the relevant traces from the pattern specification. This is also evident from the boxplots in Figures 3.12–3.15 as they show an almost linear increase in runtime with the number of traces. Additionally, we support generic patterns which are able to detect various undesired behaviours as opposed to detecting a specific violation.

That leaves us the question of overhead in terms of the size of the history. OCEP uses a very simple approach for history management. It discards multiple occurrences of the same event on a trace which have no send or receive events between them. As we are dealing with potential causality, how an event is causally related to the other events is only affected by the messages. So two events which do not have any send or receive events between them have the same causal relation with events on other traces. This only requires keeping track of the arrival of a send or receive on a trace and is $O(1)$ in complexity. Unfortunately it does not guarantee a compact subset size and may end up storing all the events since start-up in the worst case. So OCEP is not scalable to an arbitrarily large number of long-running processes.

3.5 Conclusion

In this chapter, we introduced OCEP, an efficient online causal-event-pattern-matching framework that can be used to identify safety-condition violations in distributed applications. On arrival of each terminating event, OCEP returns a representative subset of matches that spans the entire execution time. It uses vector timestamps and the causality relation among the events in the pattern to efficiently prune the search space.

We evaluated our approach with some of the most frequently found concurrency-bug patterns. OCEP successfully finds all occurrences of the pattern in each application, within a millisecond in almost all cases. This is the first available online tool that detects safety violations using generic causally related sets of events that represent event patterns.

We found the simple mechanism it uses for maintaining the event history affects its scalability. The main challenge in maintaining a bounded history is that for a generic pattern, there are

cases in which it is always possible to extend a partial match with future events. Comparison of timestamps can help but each such operation is O(n) and hence prohibitively expensive. So a technique is required that exploits the causality present in the pattern instead of a general solution. We address this issue in the next chapter and discuss how different approaches affect the length of the history and the execution time.

Chapter 4

Towards a Subset-Based Algorithm

Try a hard problem. You may not solve it, but you will prove something else.

John Littlewood

Maintaining the event history is a critical component of event-pattern monitoring. OCEP only keeps track of a newly arriving send or receive event on each trace and uses that to determine which events are redundant. It is computationally inexpensive but is not very efficient in removing redundant events. In the worst case it ends up storing all the events since startup. Thus OCEP will not scale well when the number of processes increases or the processes run for an arbitrarily long time, as the size of the event history may become arbitrarily large in both cases.

In this chapter we look at how the causality relation among the events present in the pattern can imply which matched events can possibly generate a complete match (Section 4.3). This information can then be used to maintain a subset of matched events instead of all. We discuss three plausible approaches that attempt to maintain a compact subset of matches using their causal relation with future events (Sections 4.2, 4.4, and 4.5). All of these approaches fail to report a representative subset of matches. We list them here as on the one hand they enumerate the approaches that are intuitive but not suitable to solve the problem and on the other hand they trace the steps that lead us to the solution that we propose in the next chapter. Finally we theoretically analyze the number of matches that we must store in order to report a representative subset of matches (Section 4.6). We show that in order to report any match for a generic pattern, the size of the subset that we need to maintain is exponential in the length of the pattern.

4.1 System Model and Pattern Language

Our causality framework remains the same but we fall back to a simple pattern language for our analysis in this chapter, which we show in Figure 4.1. We only use the operators *precedence* and *concurrency* and skip additional features like binding of the event and attribute variables. We show that the event-history size is exponential even for this simplified language.

$$
\begin{array}{lcl}
\textit{pattern-def} & \Rightarrow & \textit{defs}\ \ \textit{predicate} \\
\textit{defs} & \Rightarrow & \textit{defs}\ \ \textit{class-def} \\
 & | & \textit{class-def} \\
\textit{class-def} & \Rightarrow & \textit{id} \mathrel{\mathbf{:=}} \textit{class} \\
\textit{predicate} & \Rightarrow & \textit{id} \mathrel{\mathbf{:=}} \textit{pattern} \\
\textit{pattern} & \Rightarrow & \textit{term operator term} \\
\textit{operator} & \Rightarrow & \rightarrow \\
 & | & \| \\
\textit{term} & \Rightarrow & \textit{id} \\
 & | & (\textit{pattern}) \\
\textit{class} & \Rightarrow & [\textbf{process},\ \textbf{type},\ \textbf{text}] \\
\textit{id} & \Rightarrow & \textbf{alpha}\ (\textbf{alnum})*
\end{array}
$$

Figure 4.1: A grammar for specifying a simple pattern language

We give two different names to the causality relations at the two sides of a *precedence* relation. In a pattern $A \rightarrow B$, we say the event *a precedes b* and the event *b follows a*. Thus we will discuss maintaining the event history for an event in terms of the three causality relations that it can have with the other events: *precedence*, *follow*, and *concurrence*.

Ideally an online monitoring tool will need to decide whether to add a newly arriving event to the event history based on its knowledge up to the current execution time. The current execution time in a distributed system is a *cut* of the process-time diagram, which we define below.

Definition 13: *Current Cut*
The *current cut* is an imaginary plane that cuts all the processes so that it precedes all future events on a process but follows all its past events.

As we discussed in Section 3.4.1, our online algorithm receives the events from POET in a linearization of their partial order and so the *current cut* that we use is always *consistent*.

4.2 Bottom-Up Search

Our first approach maintains an event history at the internal nodes as well as the leaves as shown in Figure 4.2. A newly arriving event is compared to the leaves for a match and if found, is added to the corresponding history. A terminating event will initiate the search which will traverse the path from the leaf node to the root generating *partial matches* at each step. Each internal node will store this *partial match* to the complete pattern, which is also a *complete match* to the subtree rooted at the particular internal node. Alternatively, we can say each leaf node stores matches to the primitive events defined by the event-classes in the pattern, while each internal node stores matches to the compound event defined by the subtree that it is the root of. The root of the tree does not need to store any matches as they are reported back to the user.

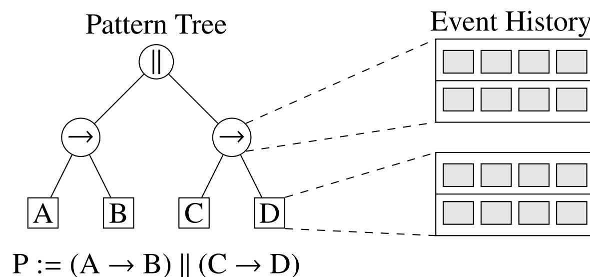


Figure 4.2: Pattern tree for a bottom-up search stores match history at all nodes

A sketch of the basic algorithm is given in Algorithm 4. The heart of the algorithm is in lines 8 – 12 where each partial match from a child node, stored in M_1, is compared to the ones stored at its sibling to see if they conform, i.e., satisfy their causality relation specified in the pattern. The conforming ones are used to build the set of partial matches for the parent node.

The three basic parameters that we will use repeatedly in theoretical analysis are given in Figure 4.3.

n	Number of traces in the monitored application
M	Average number of matched events on a trace
k	Number of events specified in the pattern

Figure 4.3: Parameters for the complexity analysis

Algorithm 4 will perform poorly as it tries to find all possible matches within the event history and hence must store all partial matches. In Section 3.2 we explained how such an approach will need to store a large number of events. In the worst case every matched primitive or compound

Algorithm 4 Bottom-Up Algorithm

Precondition: e is a newly arriving terminating event, T is the pattern tree

1: **for each** $leaf \in T$ **do**
2: **if** $e == leaf_{event}$ **then**
3: $node \leftarrow leaf$
4: $M_1 \leftarrow \{e\}$ ▷ Newly arriving event is part of the solution (if exists)
5: $M_2 \leftarrow \phi$
6: **while** $node_{parent}$ $! = NULL$ **and** $M_1 \; ! = \; \phi$ **do**
7: $sibling \leftarrow$ getSibling($node_{parent}, node$)
8: **for each** $entry \in M_1$ **do**
9: $node_{history} \Leftarrow entry$
10: **for each** $match \in sibling_{history}$ **do**
11: **if** conforms($match, entry$) **then**
12: $M_2 \Leftarrow$ createMatch($match, entry$)
13: $node \leftarrow node_{parent}$
14: $M_1 \leftarrow M_2$

event may be causally related to all the events stored at its sibling and in that case the number of stored matches will be $O(M^{logk-1}n^{logk-1})$.

Event-pattern search potentially executes on an arbitrarily large number of long-running traces and thus the amount of memory required to report all possible matches may also be arbitrarily large. While external-memory algorithms [Vitter, 2001] for handling large data sets have been studied, we are not considering them here in order to keep the latency of computation low. Arasu et al. have shown that without knowing the size of the input traces in advance, it is impossible to place a limit on the memory requirements for most common queries [Arasu et al., 2004b]. Their work focuses on Select-Project-Join queries and produces an approximate answer if the memory limit is exceeded.

In contrast, we search for the actual events that match a pattern, so it is paramount that we report a subset of matches that is representative of all matches and yet allows us to limit the size of the event history that needs to be maintained. In Section 3.3 we introduced the representative subset that OCEP used to report the matches to the pattern. In this chapter we will look at that definition and determine the size of the event history that we need to maintain in order to report it.

4.3 Minimizing the Event-History Size

Since we do not search for all possible complete matches, we do not attempt to store all partial matches either. Instead we maintain a coverage set of matches that covers the set of all matches in terms of our subset definition. Suppose we are trying to match $a \text{ op } b$ and $\mathscr{A}^a$ is the set of all possible matches for a prior to the current cut. The basic idea of a *coverage set* is to maintain a subset of partial matches that can report a *representative subset* of complete matches. In Section 3.3.1 we defined our representative subset as having *at least one occurrence per trace for each primitive event*. So the coverage set for an event should include at least one match on each trace that has a causal relation with an event on any other trace.

Definition 14: *Coverage Set*
Given the pattern a op b, the *coverage set* for the event a ($\mathscr{C}^a_{\text{op}}$) is a subset of $\mathscr{A}^a$ that contains all the traces from which there can be a match to the pattern from $\mathscr{A}^a$, i.e.,

$$\forall a_{ij} \in \mathscr{A}^a : \forall b : (a_{ij} \text{ op } b) \Rightarrow \exists a_{i'j'} \in \mathscr{C}^a_{\text{op}} : (a_{i'j'} \text{ op } b) \wedge (i = i')$$

If $\mathscr{A}^a$ has an event a_{ij}, which is the j-th event on the i-th trace and is causally related to an event b, the coverage set must also include an event $a_{i'j'}$ which is the j'-th event on the same trace and is also causally related to the event b.

4.3.1 Coverage Set for Primitive Events

The size of the event-history that we need to store at a node is thus essentially the cardinality of the coverage set. We first try to determine the cardinality of the coverage sets for primitive events, so we consider patterns that only have two event-classes related by a single causality operator. We call these *primitive patterns*.

Theorem 2. *A representative subset of complete matches can always be found for a primitive pattern by storing n matches for each primitive event in it.*

Proof. We first define the coverage sets of size n for the primitive patterns $a \to b$ and $a \parallel b$ and then prove by contradiction that these coverage sets will always be able to report a representative subset of complete matches.

Let us define the coverage sets for the the pattern $a \to b$ as

$$\mathscr{C}^a_{prec} = \{x \mid x \text{ is the } \textit{earliest} \text{ match for } a \text{ on some trace } t_i\}$$

$$\mathscr{C}^b_{folw} = \{x \mid x \text{ is the } \textit{latest} \text{ match for } b \text{ on some trace } t_i\}$$

Now let us assume that b_j is a newly arrived event on trace t_j and the set of reported matches does not include any match that involves the traces t_i and t_j. We assume there exists an $a_i \in \mathscr{A}^a$ on trace t_i for which $a_i \to b_j$. In that case our reported subset is not representative.

Since b_j is the newly arrived event on t_j, $b_j \in \mathscr{C}^b_{folw}$. If a_i is the first match for event a on t_i, $a_i \in \mathscr{C}^a_{prec}$ as well (according to definition) and $a_i \to b_j$ will definitely be reported.

Let us then assume a_0 is the first match for event a on t_i and $a_0 \in \mathscr{C}^a_{prec}$. But then $a_0 \to a_i$. So $a_0 \to b_j$ is also a match to the pattern and will be reported.

Next we define the coverage sets for the primitive pattern $a \parallel b$ as

$$\mathscr{C}^a_{conc} = \mathscr{C}^b_{conc} = \{\text{Set of } \textit{latest} \text{ matches for } a \text{ (or } b\text{) on each trace}\}$$

Again we assume that a newly arriving b_j on trace t_j fails to report a match involving the traces t_i and t_j. There, however, exists an $a_i \in \mathscr{A}^a$ on trace t_i for which $a_i \parallel b_j$, i.e., our reported subset is not representative.

Say the latest match to event a on t_i is $a_M \in \mathscr{C}^a_{conc}$. Since we did not report $a_M \parallel b_j$ as a match, either $a_M \to b_j$ or $b_j \to a_M$. But b_j is the latest event on t_j and so it cannot precede a_M. Then $a_M \to b_j$ and in that case $a_i \to b_j$ and hence cannot be a match either. □

The generic causal patterns that we search for are composed of multiple primitive patterns, so we call them *compound patterns*. Theorem 2 is not applicable to compound patterns, which we show with the simple process-time diagram in Figure 4.4. We are trying to match the pattern $A \to (B \to C)$. If we apply Theorem 2 to decide the set of matches for B, we should store the first occurrence of B on trace P_1, which is b_1. Although that will satisfy the $b_1 \to c$ part of the pattern, it will not generate a complete match as $b_1 \to a_1$ and a_1 is the only match for A. If we consider B should follow A, which it does not need to, then we can store the latest occurrence, b_4. But that also fails to generate a match as it no longer precedes c. This happens because of the contradictory requirements at node B, which needs to precede C, but must not precede A.

Storing the earliest match for covering a precedence relation thus fails when we have nested precedence requirement. The earliest match on a trace will causally precede every trace to which a message is sent after it occurs. Thus an event that should precede a future event and also follow another cannot be matched with it. The same problem will occur if we have an event that should precede and also be concurrent with some future events. Thus we need some way to compare the causality coverage of two events and make our decision based on that.

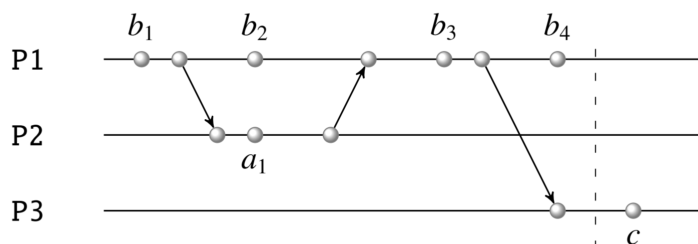


Figure 4.4: Compound patterns cannot be matched by storing n matches per event

4.3.2 Finding Redundant Events using Timestamps

In Section 3.3.4 we showed how the *greatest predecessor* (GP) and the *least successor* (LS) of an event can be used to determine on which portion of another process its causally related events can be found. An event's vector timestamp can be used to quickly calculate the event's GP on each trace (by subtracting a value of 1 from each non-zero integer in the timestamp) as it describes the number of events on each trace that precede it.

Determining the LS requires keeping track of the receive events on each trace. This can be encoded as an additional vector timestamp, which we call a *reverse timestamp*, as it keeps track of the least successors.

The relationship coverage for an event is shown in Figure 4.5 with the help of a process-time diagram. In order to keep the picture simple we have shown timestamps of a single event in each case. The timestamp shown above an event is its vector timestamp while the one shown below is its *reverse timestamp*. All three figures show the same portion of the process-time diagram.

Figure 4.5(i) shows, given an event and its two timestamps, in which parts of the process-time diagram another event can occur so that the first event will precede it. So after the current cut any event appearing on the three traces is preceded by the event e_1. As we have not seen any such event, we use the notation $e_1 \twoheadrightarrow t_1$ to denote that the event e_1 will precede any future event on the trace t_1. So in this case e_1 precedes the traces t_1, t_2, and t_3.

Figures 4.5(ii) and 4.5(iii) show the coverage for the follow and concurrence relations respectively. We can see that after the current cut, the event e_2 can only be concurrent with events appearing on t_3 as $e_2 \not\twoheadrightarrow t_3$ while $e_2 \twoheadrightarrow t_1$ and $e_2 \twoheadrightarrow t_2$.

Unfortunately keeping the reverse timestamps requires costly updates on the arrival of each receive event. We need to update it for all the events that occurred in the sending trace since the previous send event to the receiving trace. We also have to find other traces that are linked to the receiving trace by earlier send messages to the sending trace and continue the update to them

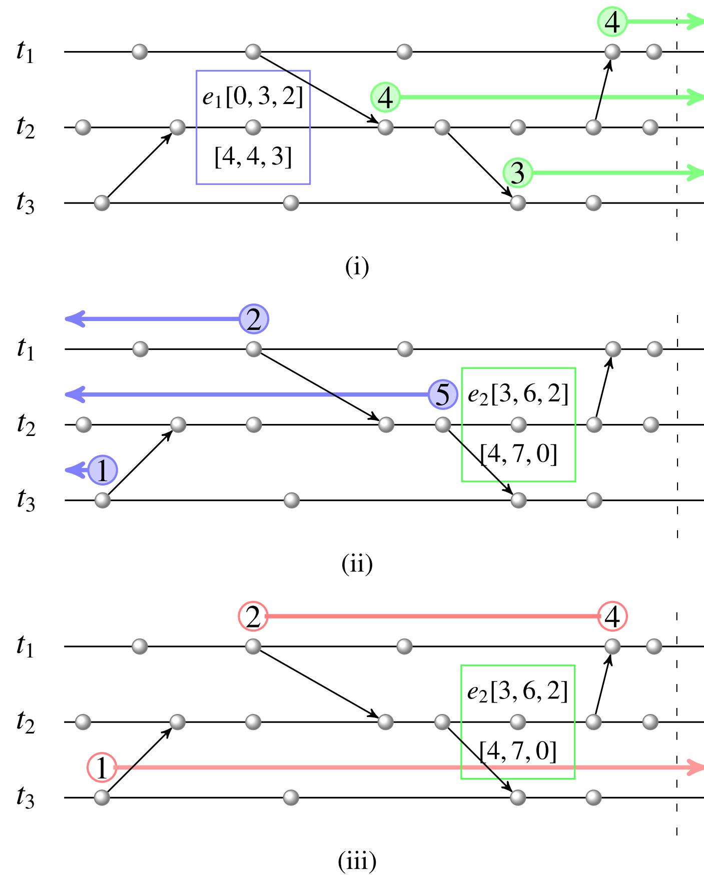


Figure 4.5: Determination of coverage using timestamps: i) Precedence, ii) Follow, iii) Concurrence

as well. Thus it is not feasible to maintain them for all events; they can only be maintained in a reasonable amount of time if the number of matches that we store is quite small.

4.4 Towards a Compact Subset

A coverage-based algorithm must look at each new partial match and compare it with the existing matches so as to maximize the future causal relations covered by the subset. Earlier we saw that a coverage subset with only the earliest/latest match per trace cannot be used for compound patterns. The main issue was that such a match was extended too far to maintain precedence while the other relations were overlooked. At any current cut, n GPs will be sufficient to maintain

precedence coverage for every other trace and similarly n LSs will be sufficient to maintain follow coverage from every other trace. The subset-based algorithms that we discuss next uses this rationale to store n matches on each trace.

In Algorithm 4, line 9 is where we add each new match to the event history. We can extend that algorithm by calling a function that checks whether the new match expands the existing causality coverage. On arrival of a new match, Algorithm 5 goes through the traces on which it has a constituent event and compares the new match with each existing match for these traces. The comparison is done by calling the function *coverage*, which basically compares each type of causality coverage as detailed in Figure 4.5. An old match is replaced if the new match expands its existing coverage.

Algorithm 5 Comparing causality coverage of a new match

Precondition: *history* is the subset of matches, arranged by traces, maintained at a node. *newmatch* is the new match that is compared to this *history*.

```
1: function checkCoverage(history, newMatch)
2:     retVal ← false
3:     for each trace on which newMatch has an event do
4:         for each oldMatch ∈ history_trace do
5:             if coverage(oldMatch) ⊆ coverage(newMatch) then
6:                 oldMatch ⇔ newMatch                    ▷ Replace with newMatch
7:                 retVal ← true
8:                 break
9:     return retVal
```

The execution time of the algorithm depends on the number of matches stored at each node. Since we store n matches on each trace at each node, there are $2^k - 1$ nodes in the tree each of which stores $O(n^2)$ matches. The execution time for *checkCoverage* is $O(kn^2)$ as we need to compare the vector timestamp, which has length $O(n)$, of the new match with each existing match. The internal nodes will have $2 \ldots k$ primitive events and the k element comes from comparing each of those.

The overall complexity can be tracked by determining the number of matches that are transferred from a child to its parent. The number of matches added to the history of the child node in the worst case is $O(n^2)$. In that case the number of matches that will be passed to its parent is $O(n^4)$. Unfortunately each of these will require a call to *checkCoverage* and so the overall time complexity is $O(n^6 logk)$ in the worst case.

4.4.1 Completeness Issue

Clearly the execution time is poor but the tree-based bottom-up traversal has some other issues as well. It is complicated to extend it to the variable-binding that we saw earlier, but the most detrimental is that coverage set cannot always be calculated based on only the partial matches in the node itself. As a result, a bottom-up search using Algorithm 5 will result in incompleteness, i.e., will fail to report some matches.

To illustrate this further we provide a simple example in Figure 4.6. Assume we are searching for the pattern $A \rightarrow (B \rightarrow C)$. a_3 is the newly arrived match for the event-class A and when we compare it with the two existing matches on t_1 we find that a_1 and a_2 precede the same two traces t_1 and t_2 and in terms of future coverage, their causality coverage is identical. So we can remove the event a_1 to make room for a_3. a_1 and a_2, however, have a different causality coverage for the existing event matches. We can see that only a_1 happens-before the event b on trace t_2. By the time a matching c event appears and a partial match $b \rightarrow c$ tries to find a preceding a event, the set of matches at its sibling node only includes a_2 and a_3.

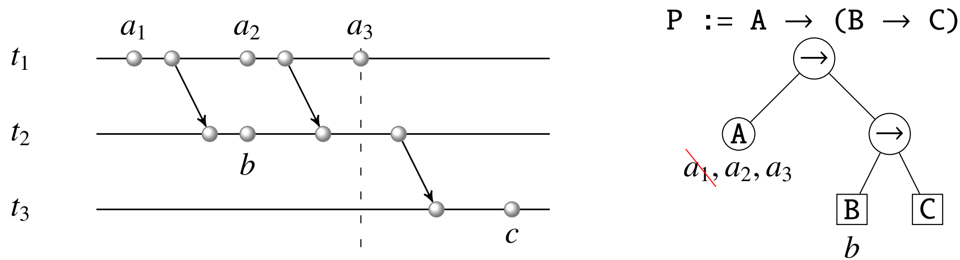


Figure 4.6: Completeness problem in the tree-based algorithm

We should add here that for this particular example, we could have chosen to remove a_2 thereby solving this problem. That solution is not generic in the sense that another pattern can be formed which would require a_2 to be stored instead of a_1. The main assumption in the subset algorithm is that it maximizes coverage for the events that arrive in the future. For previously observed events, it assumes the existing matches in a node have already formed a partial match with the matches found in the sibling node. This assumption is invalid as we are dealing with weak precedence. In this example, although a match for $b \rightarrow c$ is not found yet, a matching b exists which may generate such matches in future. So while calculating subset of matches for the node A the existence of such a match in node B needs to be taken into consideration. Thus a node cannot maintain completeness by comparing only the complete matches at the subtree rooted at itself to create a coverage subset. It will need to look at the child nodes of its sibling as well to see if there is a match which already meets the causality relation. Such a match can

already extend a partial match to the match at its parent node with some missing events in it. So we call it an *incomplete match*. The tree-based method fails because the *incomplete matches* are not taken into account while maintaining the coverage subset.

4.5 Finite Automaton

An alternative then is to store each of those *incomplete matches* in the form of a tuple instead of only the *partial matches* stored at each node of the tree. We can do that by using a finite automaton (FA) that builds a state for each of these tuples. Figure 4.7 shows the finite automaton for the pattern $A \rightarrow (B \rightarrow C)$. Each state stores an incomplete tuple and transitions are marked by the event that will cause the transition along with its causal relation with the events in the previous tuple. Because of weak precedence, an event a that precedes either of b and c will match the pattern. Thus in a complete match to this pattern, the event b may or may not follow the event a. However, it must not precede the event a.

Going back to our previous example in Figure 4.7, when the event b arrives we will find it follows event a_1 and a tuple a_1b will be added to the state AB. The event a_3 will eventually replace a_1, but since the incomplete match is already at the next state, subsequent arrival of the event c this time will report the desired match.

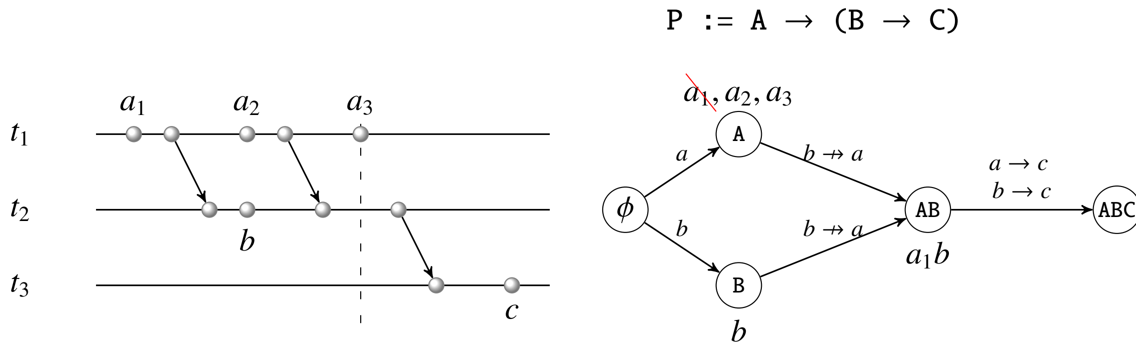


Figure 4.7: Tracking incomplete matches with a finite automaton

We show the algorithm for pattern search in Algorithm 6. On arrival of an event it examines all the states that have a transition on the event and searches through the existing matches to see if the new event forms an incomplete match with any of them. If a match is created, it is added to the history of the state at the other side of the transition after checking for coverage.

We can think about the FA as having multiple *steps* where each step comprises all the states that have the tuples of the same length. Obviously there are $k+1$ such steps for a pattern of length

Algorithm 6 FA-based Pattern Search Algorithm

```
1: for each state S ∈ FA do
2:     if e ∈ S_transition then
3:         for each match ∈ S_history do
4:             if CONFORMS(match, e) then
5:                 newMatch ← CREATEMATCH(match, e)
6:                 target ← S_target[e]
7:                 CHECKCOVERAGE(target_history, newMatch)
```

k. In the best case each of these steps will have exactly one state that has a transition on an event e. The event e can potentially match with all the events stored in a state resulting in $O(n^2)$ new matches. Considering the coverage algorithm at the target node takes $O(kn^2)$ time, the best-case will then touch one state at each level and the complexity is then $O(k^2n^4)$.

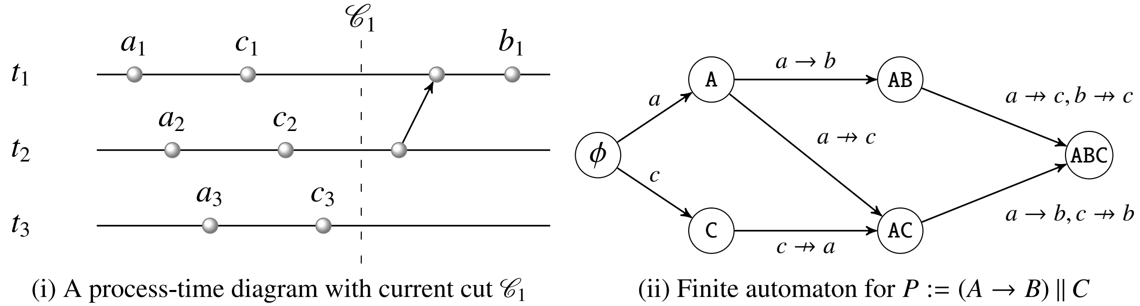
In the worst case, each of the *steps* may have all possible tuples, i.e., the number of states having r-tuples is $\binom{n}{r}$. The total number of states can then be given by

$$\sum_{0 \le r \le k} \binom{k}{r} = 2^k$$

If all of them have a transition on event e, the worst-case complexity becomes $O(2^k kn^4)$. Thus, the FA-based approach suffers from the state-explosion problem for certain types of patterns. One example can be a pattern in which all events are concurrent to each other. It should be noted that the number of states is exponential in the pattern length, which normally is a much smaller number than the number of traces.

Another notable aspect is that this approach is more suitable for a parallel algorithm. The complete matches to a pattern can only be found from the states that have a transition to the final state. So, the set of matches can be computed and returned from these states alone while the remaining states may compute their matches using a lazy update. That update process can also be parallelized as each state only needs the matches stored at itself to form the new matches.

Unfortunately it still fails to maintain completeness which we explain using a simple pattern $(A \rightarrow B) \parallel C$ in Figure 4.8. The resulting finite automaton in Figure 4.8(ii) shows that a match for event b can extend a partial match ac if $a \rightarrow b$ and $c \nrightarrow b$. Notice that a newly arriving b cannot precede an existing c and so $c \nrightarrow b$ is equivalent to $c \parallel b$ for the state AC. The partial matches at the state AC at the current cut $\mathscr{C}_1$ that relate to the trace t_1 are shown in Figure 4.8(iii). The matches are assumed to be ordered by a linearization of the partial order with time flowing left-to-right in the process-time diagram.



(i) A process-time diagram with current cut $\mathscr{C}_1$

(ii) Finite automaton for $P := (A \to B) \parallel C$

Matches for t_1	Precedence coverage	Concurrency coverage	History
a_2c_1	t_2	t_2, t_3	$\{\mathbf{a_2c_1}\}$
a_3c_1	t_3	t_2, t_3	$\{\mathbf{a_2c_1}, \mathbf{a_3c_1}\}$
a_1c_2	t_1	t_1, t_3	$\{a_2c_1, a_3c_1, \mathbf{a_1c_2}\}$
a_1c_3	t_1	t_1, t_2	$\{a_2c_1, a_3c_1, a_1c_2\}$

(iii) Maintaining the history in the state AC at the cut $\mathscr{C}_1$

Figure 4.8: Completeness problem with finite automaton

Figure 4.8(iii) uses a single coverage heuristic, maximize the coverage of both causal relationships for the maintained history. In that case the history will contain $\{a_2c_1, a_3c_1, a_1c_2\}$. In this history, the partial match that covers the trace t_1 is a_1c_2. There is a communication event from t_1 to t_2 after the current cut and as a result the event c_2 is no longer concurrent with the future events on t_1. Thus the event b_1 will fail to create a total match with the partial matches in the history although it could do so with the discarded match a_1c_3.

We only analyzed the matches for the trace t_1 in this example. The total match $a_1b_1c_3$ is also relevant to the trace t_3 and the partial matches that we store for that trace will also contribute to the subset of matches that are reported. In fact, as our example is relatively small, the matches stored at t_3 will include a_1c_3 and the reported matches will be representative. Notice that when there are n traces and we are dealing with a partial match of length m, there are ${}^nP_m$ possible matches. We are maintaining n partial matches per trace and for a partial match of length two, $n^2 > {}^nP_2$. The completeness becomes a problem when we are dealing with partial matches of length three or more and the number of traces exceeds five. In that case the number of matches that we store per trace fails to maintain a maximal coverage and we end up discarding a partial match which could later create a total match. The small example in Figure 4.8 also demonstrates this fundamental issue faced by the coverage algorithm, although for a single trace.

There are two notions that we must include in our coverage heuristic. Firstly, we should max-

imize the number of traces on which a transition event may appear and extend the partial matches currently in the history. Secondly, we need to consider how the existing coverage changes on arrival of new communication events. We consider both of these notions in the next section to determine the cardinality of the coverage subset.

4.6 Coverage Set for Compound Events

The subset-based algorithms that we discussed in the last two sections both stored n matches per trace at each node/state to maintain a coverage set. It turns out both of them fail the completeness test. The tree-based algorithm has a fundamental issue that it is not possible to choose a subset of partial matches by only looking at the partial matches available at the node. The finite automaton-based approach solved this problem by creating states for incomplete matches as well, but it finds the maximal coverage cannot be maintained with the number of matches that we are storing.

In this section we provide a theoretical analysis of the coverage set that we need to maintain for a compound event in order to report a subset of matches. To obtain the strongest possible lower-bound result, we define our reported subset as having *at least one match (if it exists)*. This is a much weaker condition than what we considered previously. No guarantee is made about the reported match (es) or its (their) location, only that if there are any matches, at least one will definitely be reported.

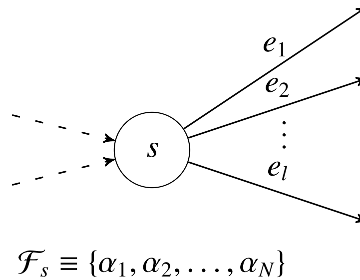


Figure 4.9: A single state in a finite automaton with all its transitions

The framework is taken as finite automaton-based. Say we are trying to match a pattern of length k and the state s in Figure 4.9 is part of the finite automaton built to store the partial matches. We assume that state s has already matched m events in the pattern and so all the partial matches in s will have m primitive events in them. We also assume the state s has already seen N partial matches, $\alpha_1, \alpha_2, \ldots, \alpha_N$. We define $\mathcal{F}_s$ as the set that contains the set of all partial matches at state s.

There can be $l \leq (k - m)$ transitions from state s and those are marked by the events $e_1, e_2, \ldots, e_l$. An incomplete match α_i in state s can be extended using the transition on arrival of such an event e_j if and only if all the events in α_i satisfy their causality relation with e_j. In that case a new incomplete match is created with α_i and e_j, which is then a candidate for the subset of matches at the target state.

Our grammar defines two causality operators, *precedence* and *concurrence*, which are defined for primitive as well as compound events. In an FA, we always extend a partial match by one event at a time and the resultant match may also be an *incomplete match*. So we need a causality relation between each pair of primitive events resulting in two additional causality relations.

The first is $e_j \nrightarrow a_i$. Consider the pattern $(A \rightarrow B) \rightarrow (C \rightarrow D)$ and a partial match with a single event a, i.e., $\alpha \equiv< a >$. In Figure 4.10 we use rounded rectangles to show how the incomplete matches can lead to a complete match. The directional arrows connecting two rectangles show the transitions and are marked by the transition events. The directional arrows inside the rectangles show the precedence relationship among the matched events. On arrival of an event c, α may form two different incomplete matches depending on its causal relation with a. In a complete match to the pattern, the event c may or may not follow event a. In both of these cases a complete match can be reached provided we find subsequent b and d events which satisfy the pattern. So the causal relationship between a and c can be written as $c \nrightarrow a$, since violating it will make $(a \rightarrow b)$ and $(c \rightarrow d)$ entangled with each other. This also means that any newly arriving c will satisfy this relation since we already received event a and the events arrive in a linearization of the partial order.

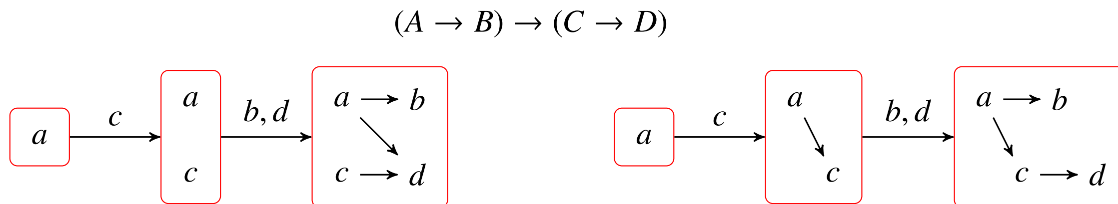


Figure 4.10: Two different ways in which event c can be causally related to event a in the pattern $(A \rightarrow B) \rightarrow (C \rightarrow D)$

The second causal relation is $a_i \nrightarrow e_j$. Consider $\alpha \equiv< a, c >$ in the previous pattern (for either of the two cases) that has a transition on event b. This time c must not precede b, as otherwise, subsequent $(a \rightarrow b)$ and $(c \rightarrow d)$ will be entangled.

The causality relationship of each transition event with the constituent events in the partial matches can be expressed with these four relations. Notice that a newly arriving event e_j may not precede any of the existing events. As a result, e_j will always satisfy the relation $e_j \nrightarrow a_i$ for

an existing a_i. And for the same reason, $a_i \parallel e_j$ is equivalent to $a_i \nrightarrow e_j$. So our coverage set can focus only on two relations: *precede* ($a_i \rightarrow e_j$) and *not precede* ($a_i \nrightarrow e_j$).

Suppose we have n traces $T \equiv \{t_1, t_2, \ldots t_n\}$. We first define a function *cov*(a_i, e_j) which gives us a set of traces on which a newly arriving e_j will satisfy its causal relation ($\triangleright$) with a_i.

$$cov(a_i, e_j) = \{t_p : e_j \text{ is a new event on } t_p \wedge a_i \triangleright e_j\}$$

Given an incomplete match α_q and a transition event e_j we can then define a function *Cov*(α_q, e_j) that gives us the traces on which a newly arriving e_j will extend α_q.

$$Cov(\alpha_q, e_j) = \bigcap_{a_i \in \alpha_q} cov(a_i, e_j)$$

Essentially the function *Cov*(α_q, e_j) quantifies the relationship coverage of a partial or incomplete match for a transition event e_j.

Theorem 3. *Finding a minimal coverage-subset is NP-hard.*

Proof. Our coverage subset in this context is the subset of matches that will be able to report at least one match if any exists. We consider the state s in Figure 4.9. Say we have l transitions from state s that are $E_s = \{e_1, e_2, \ldots e_l\}$. The complete set of matches in state s is

$$\mathcal{F}_s \equiv \{\alpha_1, \alpha_2, \ldots, \alpha_N\}$$

An incomplete match $\alpha_q \in \mathcal{F}_s$ which does not cover any transition on an event $e_j \in E_s$ will have an empty $Cov(\alpha_q, e_j)$. This particular match cannot generate a complete match even if we extend it using other transition events. So we can restrict $\mathcal{F}_s$ further as

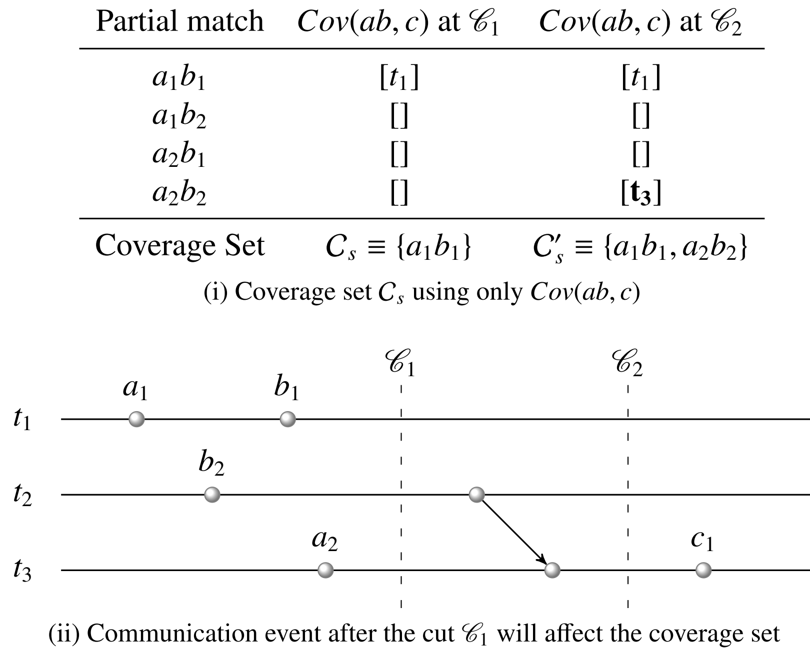
$$\mathcal{F}'_s \equiv \{\alpha_q \in \mathcal{F}_s : \nexists_{e_j} Cov(\alpha_q, e_j) \text{ is empty}\}$$

Then we can define our coverage subset as the set $C_s \subseteq \mathcal{F}'_s$ for which,

$$\forall_{e_j \in E_s} \bigcup_{\alpha_q \in \mathcal{F}'_s} Cov(\alpha_q, e_j) = \bigcup_{\alpha_r \in C_s} Cov(\alpha_r, e_j) \tag{4.1}$$

Finding a minimal subset C_s is then a set-cover problem, which is known to be NP-Hard. □

It is, however, possible to build an approximate minimal coverage-subset using a *greedy algorithm*. Recall that the function $Cov(\alpha_q, e_j)$ gives us a set whose members are taken from the set of all traces T. In that case a *greedy algorithm* that chooses a subset with every trace t_p at least once, can create an approximate minimal coverage subset. Thus the number of matches in the coverage subset by maintaining only Equation 4.1 is $O(n)$. The main problem with this algorithm is that future communication events will change the values of $Cov(\alpha_q, e_j)$ by adding or removing member traces. As a result the coverage subset that we determine at the current cut may become incomplete in future as we show in Figures 4.11 and 4.13.



Partial match	$Cov(ab, c)$ at $\mathscr{C}_1$	$Cov(ab, c)$ at $\mathscr{C}_2$
a_1b_1	$[t_1]$	$[t_1]$
a_1b_2	[]	[]
a_2b_1	[]	[]
a_2b_2	[]	$[\mathbf{t_3}]$
Coverage Set	$C_s \equiv \{a_1b_1\}$	$C'_s \equiv \{a_1b_1, a_2b_2\}$

(i) Coverage set C_s using only $Cov(ab, c)$

(ii) Communication event after the cut $\mathscr{C}_1$ will affect the coverage set

Partial match	*Prec-Trail* at $\mathscr{C}_1$	*Prec-Trail* at $\mathscr{C}_2$
a_1b_1	$[t_1]$	$[t_1]$
a_1b_2	$[t_1, t_2]$	$[t_1, t_2, \mathbf{t_3}]$
a_2b_1	$[t_1, t_3]$	$[t_1, t_3]$
a_2b_2	$[t_2, t_3]$	$[t_2, t_3]$

(iii) New event e on any trace in $Prec\text{-}Trail(\alpha, e)$ *can possibly* extend the match α

Figure 4.11: We are searching for the pattern $A \rightarrow (B \rightarrow C)$ and the communication event after the current cut is adding a new partial match to the coverage set

In Figure 4.11(ii) we are searching for the pattern $A \rightarrow (B \rightarrow C)$ and at the current cut we

have four partial matches for the incomplete pattern $A \twoheadrightarrow B$, namely a_1b_1, a_1b_2, a_2b_1, and a_2b_2. In a complete match both a and b must precede the event c and so at the current cut only the partial match a_1b_1 can be extended by a newly arriving event c on trace t_1. Thus a coverage set that only uses $Cov(\alpha, e)$ will be $C_s \equiv \{a_1b_1\}$, as shown in Figure 4.11(i). Such a coverage set ignores the other matches in which one of the events is covering some trace that a_1b_1 does not cover. If there is a new communication event from trace t_2 to t_3 then $Cov(a_2b_2, c)$ changes to $\{t_3\}$ as b_2 now precedes t_3. As a result the event c_1 on t_3 will fail to report a match using C_s.

A trace t_p can be *added* to the $Cov(\alpha, e)$ when two conditions are satisfied at the same time: i) our partial match includes one or more events that precede the transition event and ii) these events precede the trace from which a communication event is sent to the trace t_p. We define a function called $\textit{Prec-Trail}(\alpha, e)$ which captures these two conditions:

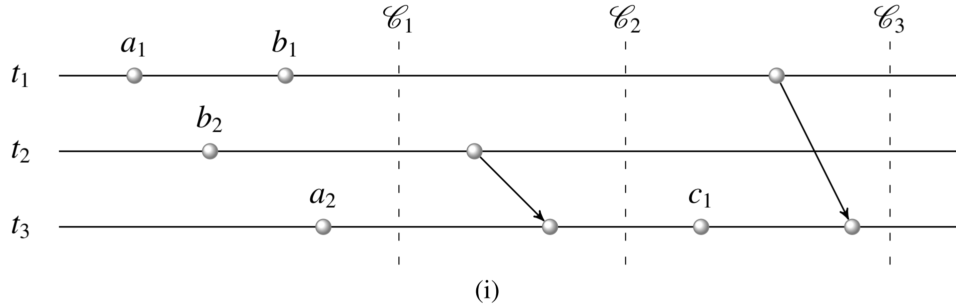
$$\textit{Prec-Trail}(\alpha, e) = \{t_p : \exists_{a_i \in \alpha}(a_i \rightarrow e) \wedge (a_i \multimap t_p)\}$$

In Figure 4.11, $\textit{Prec-Trail}(a_1b_1, c) \equiv [t_1]$ as both primitive events occur on the trace t_1. As a_2 and b_2 occur on the traces t_2 and t_3, $\textit{Prec-Trail}(a_2b_2, c) \equiv [t_2, t_3]$. Given a partial match α, the function *Prec-Trail* gives us the set of traces which are preceded by some primitive events in α that precede the transition event e. Another way to describe it is when we view the process-time diagram as a directed acyclic graph. Then *Prec-Trail* returns the set of traces to which a directed path exists from some primitive events in α that precede e. Thus *Prec-Trail* gives us a trail of the partial match's precedence relation within which a transition event should occur.

We can then use *Prec-Trail* to identify those partial matches that may be added to the coverage set on arrival of future communication events. Now can we obtain a complete coverage if we maximize the number of traces covered by the precedence trails of the partial matches in C_s? Before answering this question we point out the subtlety in the causal relations with a transition event. In the pattern $A \rightarrow (B \rightarrow C)$, the primitive event A precedes the compound event $B \rightarrow C$. Since we are using weak precedence, A can precede either B or C. When we build a finite automaton for this pattern, each transition is marked by the causal relation between the transition event and each individual event in the previous state. As we show in Figure 4.7, a transition event c from state AB must follow both a and b in an incomplete match. Maximizing $\textit{Prec-Trail}(\alpha, e)$ will fail to take this into account when we have multiple events in α that should precede e. In Figure 4.11(ii) coverage set C_s will be $\{a_1b_1, a_1b_2, a_2b_1\}$ at the cut $\mathscr{C}_1$ and $\{a_1b_1, a_1b_2\}$ at the cut $\mathscr{C}_2$. None of them will be able to report the match $a_2b_2c_1$.

Thus we need to consider the precedence trail for each individual event that precedes the transition event. Can we then obtain complete coverage by maximizing the precedence trail of each of these primitive events? We redraw the process-time diagram in Figure 4.12 with *Prec-Trail* for each primitive event. This time the coverage set C_s will be $\{a_1b_1, a_1b_2, a_2b_1\}$ at

both $\mathscr{C}_1$ and $\mathscr{C}_2$. As we maximized each event's precedence trail, we have at least one incomplete match with a single event preceding the transition event, i.e., a_2 in a_2b_1 and a_1 in a_1b_2. Thus the coverage set will still fail to report a complete match.



(i)

Partial match	Primitive Event	*Prec-Trail* at $\mathscr{C}_1$	*Prec-Trail* at $\mathscr{C}_2$	*Prec-Trail* at $\mathscr{C}_3$
a_1b_1	a_1	$[t_1]$	$[t_1]$	$[t_1, \mathbf{t_3}]$
	b_1	$[t_1]$	$[t_1]$	$[t_1, \mathbf{t_3}]$
a_1b_2	a_1	$[t_1]$	$[t_1]$	$[t_1, \mathbf{t_3}]$
	b_2	$[t_2]$	$[t_2, \mathbf{t_3}]$	$[t_2, t_3]$
a_2b_1	a_2	$[t_3]$	$[t_3]$	$[t_3]$
	b_1	$[t_1]$	$[t_1]$	$[t_1, \mathbf{t_3}]$
a_2b_2	a_2	$[t_3]$	$[t_3]$	$[t_3]$
	b_2	$[t_2]$	$[t_2, \mathbf{t_3}]$	$[t_2, t_3]$

(ii)

Figure 4.12: Precedence trails of individual events are required to correctly identify which incomplete match covers a new trace on arrival of a communication event

Two incomplete matches in which the primitive events have non-intersecting *Prec-Trail*, will be affected differently by future communication events. In Figure 4.12, the communication after the cut $\mathscr{C}_1$, will add t_3 to *Prec-Trail*(b_2, c). Since *Prec-Trail*(a_2, c) already contains t_3, the incomplete match a_2b_2 will now cover a transition event c on the trace t_3. We can thus only discard an incomplete match if for each of its events, *Prec-Trail* is a subset of *Prec-Trail* for the corresponding event in another match. We define a boolean function $Covers_P(\alpha_q, \alpha_r, e)$ that returns true when α_r covers the match α_q for preceding the event e. If the i-th primitive events in α_q and

α_r are a_i and a'_i respectively,

$$Covers_P(\alpha_q, \alpha_r, e) \equiv \forall_i : (a_i \rightarrow e) \wedge (a'_i \rightarrow e) \Rightarrow (\textit{Prec-Trail}(a_i, e) \subseteq \textit{Prec-Trail}(a'_i, e))$$

Using $Covers_P$ we can summarize the rule that a coverage set must satisfy in order to keep those matches that can cover new traces in the future.

$$\forall_{e_j \in E_s} \exists_{a_i \in \alpha_q} (a_i \rightarrow e_j) : \forall_{\alpha_q \in \mathcal{F}'_s}\ \alpha_q \notin C_s \Rightarrow \exists_{\alpha_r \in C_s} Covers_P(\alpha_q, \alpha_r, e_j) \tag{4.2}$$

In the above equation, we are using set notation for α_q although it is a vector of matched primitive events. Also notice we have not explicitly mentioned that $\exists_{a'_i \in \alpha_r}(a'_i \rightarrow e_j)$. Since both α_q and α_r are matches from the same state in the finite automaton, they are actually a match to the same sub-pattern. Thus their corresponding events will have same causal relation with a transition event.

The number of matches that we need to store for Equation 4.2 depends on the number of distinct values of *Prec-Trail*(a, e). For an event a on trace t, *Prec-Trail* changes monotonically on arrival of subsequent communication events. Thus there can be $O(n)$ distinct values of *Prec-Trail* for events occuring on a single trace. If the partial match at the state s has m primitive events that precede a transition event e, then we may need to store $O(n^m)$ partial matches to cover an arbitrary transition event.

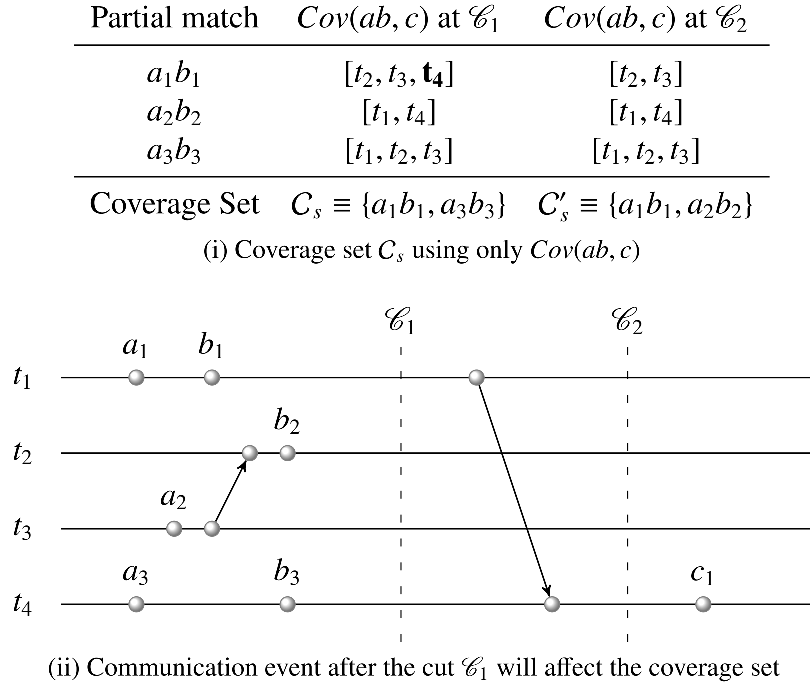
In Figure 4.13 we are searching for the pattern $(A \rightarrow B) \parallel C$ and at the current cut we have three partial matches for the sub-pattern $A \rightarrow B$, namely a_1b_1, a_2b_2, and a_3b_3. This time, both a and b must be concurrent with c. If we look at the traces the partial matches cover for a newly arriving event c, we can determine that a possible coverage set $C_s \equiv \{a_1b_1, a_3b_3\}$. As long as no new communication events arrive, this subset can return a match to the complete pattern whenever an event c happens on any of the traces.

If there is a new communication between trace t_1 and t_4, then $Cov(a_1b_1, c)$ changes to $\{t_2, t_3\}$ as b_1 now precedes t_4. As a result the event c_1 on t_4 will fail to report a match using C_s, although our previously discarded match a_2b_2 could create a complete match to the pattern. A minimal or an approximate coverage-subset using only $Cov(\alpha, e)$ will thus fail to maintain completeness unless we look at the issue of traces becoming covered or uncovered as a result of future communication events.

A trace t_p can be *removed* from the $Cov(\alpha, e)$ when the following two conditions are satisfied at the same time: i) our partial match includes one or more events that *do not* precede the transition event and ii) these events precede the trace from which a communication event is sent to the trace t_p. These conditions are similar to the ones we used to define *Prec-Trail* except this time we are removing a trace and the transition event is concurrent. We define a function called

Partial match	$Cov(ab, c)$ at $\mathscr{C}_1$	$Cov(ab, c)$ at $\mathscr{C}_2$
a_1b_1	$[t_2, t_3, \mathbf{t_4}]$	$[t_2, t_3]$
a_2b_2	$[t_1, t_4]$	$[t_1, t_4]$
a_3b_3	$[t_1, t_2, t_3]$	$[t_1, t_2, t_3]$
Coverage Set	$C_s \equiv \{a_1b_1, a_3b_3\}$	$C'_s \equiv \{a_1b_1, a_2b_2\}$

(i) Coverage set C_s using only $Cov(ab, c)$



(ii) Communication event after the cut $\mathscr{C}_1$ will affect the coverage set

Partial match	*Conc-Trail* at $\mathscr{C}_1$	*Conc-Trail* at $\mathscr{C}_2$
a_1b_1	$[t_1]$	$[t_1, \mathbf{t_4}]$
a_2b_2	$[t_2, t_3]$	$[t_2, t_3]$
a_3b_3	$[t_4]$	$[t_4]$

(iii) New event e on any trace in *Conc-Trail*(α, e) *will not* extend the match α

Figure 4.13: We are creating the coverage set C_s using only $Cov(\alpha, e)$ while searching for the pattern $(A \rightarrow B) \parallel C$. The communication event after the current cut is removing an existing partial match to $Cov(\alpha, e)$

Conc-Trail(α, e) which tracks the precedence trail of events that *do not* precede the transition event.

$$\textit{Conc-Trail}(\alpha, e) = \{t_p : \exists_{a_i \in \alpha}(a_i \nrightarrow e) \wedge (a_i \rightharpoonup t_p)\}$$

Looking back at Figure 4.13, *Conc-Trail*$(a_1b_1, c) \equiv [t_1]$ as both primitive events occur on the trace t_1. As a_2 and b_2 occur on the traces t_2 and t_3 *Conc-Trail*$(a_2b_2, c) \equiv [t_2, t_3]$. Only the match a_1b_1 is linked to the trace t_1 from which the communication event is sent to the trace t_p. So the communication event affects the match a_1b_1 but not a_2b_2 as none of the latter's primitive events

are linked to t_1.

Thus if there is a communication event from a trace $t_s \in \textit{Conc-Trail}(\alpha, e)$ to $t_r \in \textit{Cov}(\alpha, e)$ then the incomplete match α can no longer extend an event e that occurs on t_r. So we can remove t_r from $\textit{Cov}(\alpha, e)$ and also put it in $\textit{Conc-Trail}(\alpha, e)$. The function $\textit{Cov}(\alpha, e)$ looks at only the existing coverage at the current cut while two partial matches with non-intersecting *Prec-Trail* or *Conc-Trail* means future communication events can affect one but not the other. In Figure 4.13, the communication event from t_1 to t_4 will thus remove t_4 from $\textit{Cov}(a_1, b_1, c)$ and put it in $\textit{Conc-Trail}(a_1 b_1, c)$. Also a subsequent communication event from t_4 to t_2 will remove t_2 from the *Conc-Trail*s of both $a_1 b_1$ and $a_3 b_3$.

We can then only discard an incomplete match if its *Conc-Trail* is a superset of an existing match's *Conc-Trail*. In that case all the traces that the existing match is linked to are also linked to the discarded match and so there cannot possibly be a situation when the discarded match extends a transition event while an existing match does not. We can summarize this into another rule that a coverage set must satisfy when the i-th primitive event in a match is concurrent to a transition event e_j.

$$\forall_{e_j \in E_s} \exists_{a_i \in \alpha_q} (a_i \nrightarrow e_j) : \forall_{\alpha_q \in \mathcal{F}'_s}\ \alpha_q \notin C_s \Rightarrow \exists_{\alpha_r \in C_s} \textit{Conc-Trail}(\alpha_r, e_j) \subseteq \textit{Conc-Trail}(\alpha_q, e_j) \quad (4.3)$$

Our coverage set then needs to satisfy all the Equations 4.1– 4.3. Equation 4.1 maintains an approximate coverage set with respect to the current cut. Equation 4.2 ensures we do not discard any unique match that satisfies a *precede* relation with the transition event on arrival of future communication events. Equation 4.3 ensures we do not discard a partial match that satisfies a *not precede* relation with the transition event and is not affected by some future communication events.

Notice that Equation 4.3 does not need to consider *Conc-Trail* of individual events. In effect, *Conc-Trail* also keeps the precedence trail of the incomplete match but the transition event must avoid it. So it tells us which traces do not conform with the incomplete match and, more importantly, future communication events will only expand *Conc-Trail*. The last property is true for *Prec-Trail* as well, i.e., it can only expand on arrival of future communication events. But *Prec-Trail* gives us the traces that conform with the incomplete match and so we only discard matches when all of its individual precedence trails are covered by another match.

We assumed that the state s contains m primitive events in each of its incomplete matches. If all of them are concurrent to a transition event, then there are n^m ways in which they can precede n traces. Since Equation 4.3 only removes supersets, there are $\binom{n}{m}$ matches in the worst case which will have non-intersecting *Conc-Trail*s.

Theorem 4. *At least one match (if it exists) can be reported if and only if the coverage set satisfies Equations 4.1– 4.3.*

Proof. Assume the coverage set stored in a state s is C_s. We first prove the necessary condition that if C_s satisfies all the equations, it will always be able to extend a match (if any exists) on arrival of a transition event.

Let us assume to the contrary that a newly arriving event e on trace t cannot extend any of the matches in C_s, but there is a match $\alpha_q \in \mathcal{F}_s$ that can be extended by the event e. This can happen only if some *precede* or *not precede* relation was not covered by the coverage set C_s.

Since $\alpha_q \notin C_s$, there are two possible cases: i) α_q was in C_s and then a new match α_r substituted it or ii) α_q was a new match at some point and it was discarded because of an existing α_r (possibly multiple) that already covered the traces it covered.

We first assume α_q covers a *precede* relation that no other match in C_s does. Without loss of generality, we can assume $a_i \in \alpha_q$ precedes the transition event e. Since α_q is not in C_s because of α_r, $Covers_P(\alpha_q, \alpha_r, e)$ must be true. In that case, *Prec-Trail*$(a_i, e) \subseteq$ *Prec-Trail*(a'_i, e), where a'_i is the corresponding event in α_r. But then a'_i is also linked to all the traces that a_i is linked to and so must precede e. As $Covers_P$ compares each individual event's *Prec-Trail*, this will still be valid when α_q has multiple events that precede e.

In the second case *Conc-Trail*$(\alpha_r, e) \subseteq$ *Conc-Trail*(α_q, e) at the *cut* ($\mathscr{C}$) when we compared α_q with α_r. Since α_q has a transition on the event e, $Cov(\alpha_q, e)$ must include t. Recall that a trace is removed from *Cov* only if we have a communication from any trace in *Conc-Trail* of the partial match. Thus there was no communication from any $t_s \in$ *Conc-Trail*(α_r, e) to trace t since $\mathscr{C}_s$. But $ConcTrail(\alpha_r, e) \subseteq$ *Conc-Trail*(α_q, e). So $t \in Cov(\alpha_r, e)$ as well and α_r should also have a transition on the event e.

For the sufficient condition, let us first assume that we do not satisfy Equation 4.1:

$$\exists_{t \in T}(\exists_{\alpha_q \in \mathcal{F}_s} t \in Cov(\alpha_q, e) \land \nexists_{\alpha_r \in C_s} t \in Cov(\alpha_r, e))$$

But in that case if a new event appears that has a transition event e on trace t, none of the matches in C_s will be able to extend it.

Secondly, we assume Equation 4.2 is not fulfilled, i.e.,

$$\exists_{\alpha_q \in \mathcal{F}_s} : \nexists_{\alpha_r \in C_s} Covers_P(\alpha_q, \alpha_r, e)$$

We assume two matches $\alpha_q \in \mathcal{F}_s$ and $\alpha_r \in C_s$ that do not cover a newly arriving event e on trace t. We also assume that the only unfulfilled causal relation with respect to the trace t for both

of them is $a_i \in \alpha_q$ (and $a'_i \in \alpha_r$) precedes e, i.e.,

$$t \notin \textit{Prec-Trail}(a_i, e) \text{ and } t \notin \textit{Prec-Trail}(a'_i, e)$$

Each partial match then needs a communication event from any trace in its *Prec-Trail* to the trace t to cover it. Since Equation 4.2 is not satisfied, it is possible that $\textit{Prec-Trail}(\alpha_q, e) - \textit{Prec-Trail}(\alpha_r, e) \neq \phi$. But if there is a communication event from any of these traces to trace t then t will be added to $Cov(\alpha_q, e)$ only. Thus none of the matches in C_s will be able to extend an event e on trace t.

Let us then assume we do not satisfy Equation 4.3. We can then assume that at the current cut both $\mathcal{F}_s$ and C_s cover a certain trace t, i.e.,

$$\exists_{\alpha_q \in \mathcal{F}_s} t \in Cov(\alpha_q, e) \text{ and } \exists_{\alpha_r \in C_s} t \in Cov(\alpha_r, e)$$

So both α_q and α_r will have a transition on a future event e on trace t. Since Equation 4.3 is not satisfied we may have

$$\exists_{\alpha_q \in \mathcal{F}_s} : \nexists_{\alpha_r \in C_s} \textit{Conc-Trail}(\alpha_r, e) \subseteq \textit{Conc-Trail}(\alpha_q, e)$$

In that case $\textit{Conc-Trail}(\alpha_r, e) - \textit{Conc-Trail}(\alpha_q, e) \neq \phi$. But if there is a communication event from any of these traces to trace t then t will be removed from $Cov(\alpha_r, e)$ but not from $Cov(\alpha_q, e)$. So α_r will no longer match a new event e on t although α_q will. □

The requirements for the two equations 4.2 and 4.3 are independent as one covers the *precedes* relation and the other covers the *not precedes*. Thus the cardinality of the coverage subset that will be able to report *at least one match (if exists)* is $O(n^k)$. Recall that the definition of the reported subset in this section is weaker than our *representative subset*. Thus, even for a simple grammar, reporting any match to a generic pattern requires storing a history exponential in the pattern length.

4.7 Conclusion

In effect, maintaining a compact subset of matches is a costly operation. A bounded subset can only be used for a limited number of patterns. A generic pattern may require storing an exponential number of matches to maintain completeness. Using a coverage algorithm to maintain such a subset adds to the average complexity of the pattern-search algorithm as we need to execute it to make space for every partial match that is created. Additionally, we need to track each receive

event in order to compare relationship coverage between partial matches, which becomes costly as the subset size increases.

In the previous chapter OCEP only stored the primitive events that match the pattern. It did not store any partial matches at the internal nodes. On arrival of a new terminating event it tried to gather all its causally related events from the events that were already seen. We saw that the vector timestamps can be used to effectively prune the search space so as to find a representative subset of matches. We tracked the receive events only for comparing consecutive matches on a trace for redundancy, which is simpler than updating the reverse timestamps of a potentially large number of events.

The main problem with OCEP was that the event history was not bounded as we did not eliminate enough redundant events. In this chapter we saw that using causality to maintain a subset of matches is computationally expensive and does not guarantee a compact subset either. In the next chapter we use an alternative approach that uses application-specific knowledge to determine when an event can no longer generate a new match.

Chapter 5

Removal by Rules

We know what does not work much better than what works.

Nassim Nicholas Taleb in Antifragile

The size of the event history determines the execution time of the pattern-search algorithm that executes on top of it. In the last chapter we saw how the causality relation among events in the pattern can be used to discard redundant events from the history. The complexity of maintaining such a subset is polynomial with respect to the size of the subset. Two major problems with this approach are that we have to update this subset on arrival of each matched event and the size of the subset may grow exponentially.

At this point we take a step back and look at the causality relationship that we are using. The happens-before relationship on messages indicates *potential causality* instead of *actual causality*. If one event *happens-before* another event, the first event does not necessarily cause the second [Cheriton and Skeen, 1993]. Thus a bounded event history created using this potential causality may at best be too costly and at worst rule out *actual causality* in favor of *potential causality*. Information about a program's behaviour is not available at the communication level. In Chapter 3, we used event attributes that expose some information about the event itself. These attributes are used along with causality to define a pattern of interest. In this chapter we take a similar approach by providing a mechanism in the pattern to express when a matching event becomes redundant and can be removed.

5.1 Pattern Language with Removal Rules

We have so far used a 3-tuple for specifying a class of events in our pattern language. In this chapter, we introduce another component, a removal rule, to the definition of an event class. The grammar is given in Figure 5.1. It builds on the same grammar that we used for OCEP and adds the rules for removing the matches for an event class.

pattern-def	⇒	*defs decls predicate*
defs	⇒	*defs class-def*
	|	*class-def*
class-def	⇒	*id*[*removal-rule*] **:=** *class*
decls	⇒	*decls var-decl*
	|	ϵ
var-decl	⇒	*id variable* (**,** *variable*)$*$
predicate	⇒	*id* **:=** *pattern*
pattern	⇒	*term operator term*
	|	*term connector term*
operator	⇒	$\rightarrow$
	|	$\parallel$
connector	⇒	$\wedge$
	|	$\vee$
term	⇒	*id*
	|	*variable*
	|	(*pattern*)
class	⇒	[**process**, **type**, **text**]
	|	[**process**, **type**, *variable*]
	|	[*variable*, **type**, **text**]
	|	[*variable*, **type**, *variable*]
variable	⇒	$*id*
id	⇒	**alpha** (**alnum**)$*$
removal-rule	⇒	[*operator*, *qualifier*, **type**, **text**]
qualifier	⇒	**same** | **different** | **any**

Figure 5.1: Grammar for specifying a pattern with removal rules

The removal rule is an optional component when specifying an event class. We have used two different types of removal rules:

- *Default rules* look only at the causal relations among the events in the pattern and remove

events from the history that can no longer generate any new matches. If no removal rule is specified for an event class, the default rule is used for it.

- *Event-based rules* specify a *trigger event* and its causal relation to a specific event in the pattern. The occurrence of the trigger event is then used to clear the specified events from the history.

The rules that are present in a pattern are event-based rules. A given rule can specify the trigger event (using *type* and *text*) and the causal relation with the removed events (using *operator*). The scope of the removal can further be controlled by a *qualifier* to remove events that are on the same/different trace as the trigger event.

```
X [ →, any, EVENT_Y, ‘’] := [‘’, EVENT_X, ‘’];
```

For example, the class definition above specifies that a match to *EVENT_X* should be removed when it is found to precede an *EVENT_Y* on any trace.

5.2 Pattern Search with Ananke

We use the same backtracking algorithm that OCEP uses for finding a representative subset of matches to a pattern. Each event in our pattern tree in Figure 5.2 now includes a new attribute for the removal rule. We also have a separate data structure that we call *Trigger events* and it holds the events that match the ones specified in the removal rules. These events trigger the removals and are removed once the rule is applied to all relevant event histories.

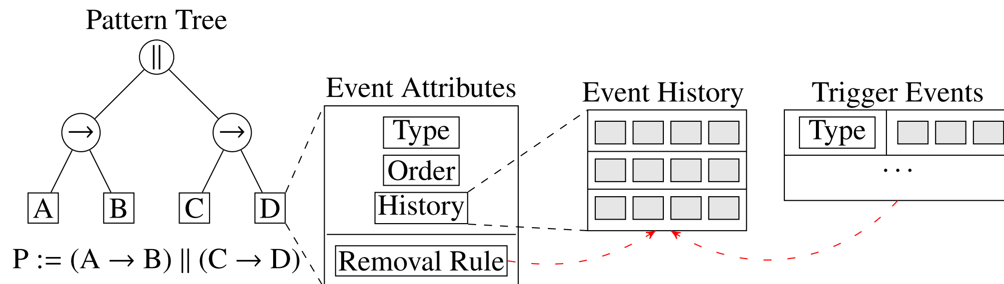


Figure 5.2: Pattern tree and the use of trigger events along with the rules for removing events from the history

The pattern search itself is handled in one thread and history management with *Ananke*[1] in a separate one.

5.3 Managing History with Ananke

The events that are targeted for removal by the rules can be categorized into two different types.

- *Unmatched* events are the ones which cannot create any complete match.
- *Redundant* events are those which may be part of a complete match, but we also have another event in history that covers these complete matches.

The motivation for using the removal rules can be explained using Figure 5.3. The vertical line represents the *current-cut* and so we assume any complete matches that are found before it are already reported. We concern ourselves with the problem of storing the events that can potentially provide matches with the events that appear after the current-cut.

5.3.1 Default Removal Rules

Concurrency

In Section 4.3.2 we introduced the notion of an event preceding a trace, $a_1 \twoheadrightarrow T_1$, when there exists a communication event connecting the event to the trace. In Figure 5.3, a_1 will precede any future event on trace T_1. Now consider the removal rule for an event-class A that is concurrent to the event-class B in a pattern. In that case, events a_1 and a_2 can no longer generate new matches to the pattern as they precede all the existing traces. a_1 and a_2 will happen-before any new event that appears on any trace after the current cut. We assume a newly starting trace is created by one of the existing traces and so the two events will precede events happening on any possible future trace as well. If we are searching for a pattern $(A \parallel B) \parallel C$, we can then remove the events a_1 and a_2 from the history.

Now consider another pattern $(A \parallel B) \rightarrow C$, in which the event-class A is concurrent to the event-class B and also precedes the event-class C. This time a_1 and a_2 already have a match for B and so they can no longer be removed based only on concurrency. Thus, a removal decision is essentially made by considering all the causal relations for an event class.

[1] *Ananke* was the primordial Greek deity of force, constraint, and necessity who brought about the creation of the *ordered universe* along with the god of *time*, Chronos.

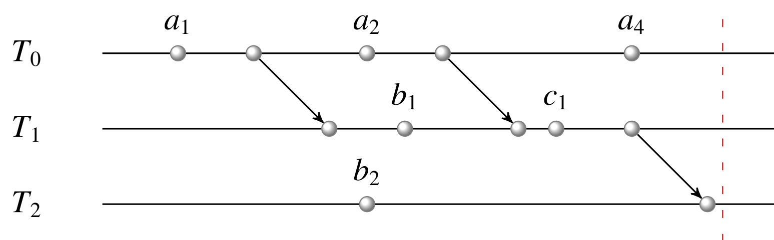


Figure 5.3: Using rules for removing events

Existing Matches	Future Matches	Decision
X	X	Strong Remove (SR)
X	√	Weak Keep (WK)
√	X	Weak Remove (WR)
√	√	Weak Keep
SR	Remove regardless of other relations	
WR	Remove if all relations call for removal	
WK	Keep unless an SR appears	

Figure 5.4: Removal decisions for concurrency

We summarize the removal decision for a concurrency relation in Figure 5.4. If the event is concurrent to some traces then future events on them can satisfy the concurrency and so we keep the event unless some other causal relation calls for a removal.

If an event precedes all the traces then there are no future concurrency matches for it and so we check for its existing concurrency matches. If none exist, it can never be part of a complete match to the pattern and so it can be removed. If there are some existing matches, then a complete match can still be found on arrival of a future event with a different causal relation. We wait for the removal decision in this case until all the relations are checked and the event is removed only if every relation considers it to be a removal.

A precedence pattern, $A \to B$, will have two different rules for the preceding side and the follower side.

Algorithm 7 Using Default Rules for Event Removal

Precondition: *remNode* and *remTrace* define the (*node*, *trace*) tuple from which events are being removed. *remEvent* is the event that is being considered for removal.

```
1: function useDefaultRule(remNode, remTrace, remEvent)
2:     removal ← WR
3:     ▷ First, check default rule for concurrency
4:     for each concNode ∈ remNode.concList do
5:         remConc ← checkConcurrency(remEvent, concNode)
6:         if remConc = SR then
7:             return true
8:         removal ← remConc
9:     ▷ Now, check default rule for follow
10:    for each folwNode ∈ remNode.folwList do
11:        remFolw ← checkFollow(remEvent, folwNode)
12:        if remFolw = SR then
13:            return true
14:        else if remFolw = WK then
15:            removal ← WK
16:    ▷ Check default rule for precedence
17:    for each precNode ∈ remNode.precList do
18:        nextEvent ← getNext(remNode, remTrace, remEvent)
19:        remPrec ← checkPrecedence(nextEvent, remTrace, precNode)
20:        if remPrec = WK then
21:            removal ← WK
22:    if removal = WR then
23:        return true
24:    else
25:        return false
```

Precedence

Consider the two matches for event class A on trace T_0. In Figure 5.3, a_1 and a_2 are both on T_0 and precede the same trace T_1. So any new event that appears on trace T_1 will be preceded by both and we can remove a_1 as it is redundant.

Essentially an event that has a precedence relation can always be considered to have a future match, on its own trace or some other trace to which it is linked by communication events. The

default rule for precedence is conservative in the sense that it only tries to remove a matched event that has no unique match compared to its successor. For such a redundant event, a Weak Remove (WR) decision is taken for precedence and the event is removed if all other relations also consider it to be a removal.

Follow

At the follower side of $A \rightarrow B$, event class B has two matches b_1 and b_2 in Figure 5.3. Follow events can only have past matches and so an event that has no existing match (b_2 on T_2 has no matching A preceding it) can be removed as it will not generate any complete match (SR). If there are existing matches, then it can potentially create future matches provided other relations are satisfied. Thus a WR decision is taken in this case, which will be canceled if some other relation has a future match.

The algorithm for using the default removal rules for an event-class is given in Algorithm 7. It is called for each event stored in the current node (*remNode*) that is being traversed for removals. It goes through all the causally related nodes of *remNode* and gathers their individual removal decisions. The functions that are called for checking each of these decisions will be discussed in the next section after we introduce the indices that are used for them. If any of the decisions is an SR, it returns the removal decision as *true*. The WR decisions will need all of the relations to call for a removal for the event to be removed.

5.3.2 Indices for Default Rules

We first introduce a few indices that we build from the timestamps of the stored events and the communication events. These indices are constantly updated as new events appear and/or existing matches are removed and are used for the default-rule-based removals.

gpTrace

We defined *greatest predecessor* of an event e on a trace as the most-recent event on that trace that happens before e (Definition 11). On the other hand, *least successor* of an event e on a trace is the least-recent event on that trace that happens after e (Definition 12).

The least successor of an event on trace t is useful to track in which portion of the trace a following event may appear. In Section 4.3.2 we used *reverse timestamps* to keep track of the least successors for each event. Unfortunately, maintaining the reverse timestamp for an event requires updating it on arrival of the first communication event to each trace since the event appeared and is computationally expensive.

Instead we store this information as a per-trace index in *gpTrace*. In essence, the i-th entry in $gpTrace_t$ stores the $TS[t]$ for the latest communication event on trace t that connects it directly or indirectly to trace i, i.e. the GP on trace t for trace i.

gpTrace is only updated on arrival of a new communication event. It might seem that in each case we need to update the entries for only the two traces that are communicating. It can happen that the sending trace is already linked to some other traces all of which will now have a GP to the receiving trace. Thus the operation is $O(n)$ in the worst case.

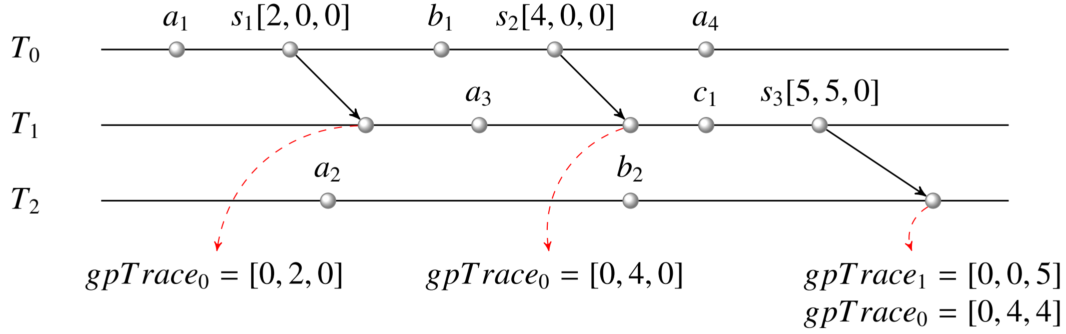


Figure 5.5: A new communication event is used to update *gpTrace*

We show how *gpTrace* gets updated on arrival of a few communication events in Figure 5.5. We have shown the timestamps for only the send events for brevity. Each of the events s_1 and s_2, on arrival of its corresponding receive event, will become the GP on trace t_0 for the trace t_1. So both of them update $gpTrace_0[1]$ to their respective $TS[0]$. The event s_3 will be the GP for trace t_2 on t_1 and so it updates $gpTrace_1[2]$. But as the trace t_0 already has a GP for t_1, the event s_2 will now become a GP to t_2 as well and so $gpTrace_0[2]$ is also updated.

gpToAll

The t-th entry in $gpToAll$ stores the $TS[t]$ for the event on the t-th trace that is a GP to all the traces. This can be calculated from *gpTrace* using the following formula:

$$gpToAll[t] := min_{\forall i \in [1...n] \wedge i \neq t}(gpTrace_t[i])$$

If a concurrent event precedes all the traces then it will precede all newly arriving events and thus has no future concurrency matches. Then the condition that needs to be *true* for an event a on *evTrace* to have a future concurrency match can be summarized as

$$a.TS[evTrace] > gpToAll[evTrace]$$

Algorithm 8 Default Removal Rule for Concurrency

Precondition: *concNode* is the causally related node and *remEvent* is being considered for removal

1: **function** CHECKCONCURRENCY(*remEvent*, *remTrace*, *concNode*)
2: **if** remEvent.TS[remTrace] > gpToAll[remTrace] **then**
3: **return** WK
4: **else**
5: **for each** $trace \neq remTrace$ **do**
6: **if** *concNode*.MATCHEXISTS(*remEvent*.TS[*trace*]) **then**
7: **return** WR
8: **return** SR

The algorithm for the concurrency decision is given in Algorithm 8. If there are no future matches, we then check the concurrent node to see if it has any existing matches for the event. We showed in the last section how the returned decision is compared with all the decisions from the other causally related nodes to reach the removal decision.

The following two indices are stored for each node in the pattern tree and they are relevant only to the matches that are currently stored at the node. They are initialized when a match is found for the particular node and later updated when we remove an existing event from it.

firstIndex

The i-th entry in $firstIndex$ stores the $TS[i]$ for the first matched event on trace i that is stored at the node. In Figure 5.6, we show the previous process-time diagram with the timestamps for the a events. Considering all the matches to event A are currently in the history, the events a_1, a_2, and a_3 are the first matches on each trace. In that case $firstIndex_a = [1, 2, 1]$. If we now remove the event a_1 from history, the subsequent event a_4 on trace t_0 will update $firstIndex_a$ to $[5, 2, 1]$.

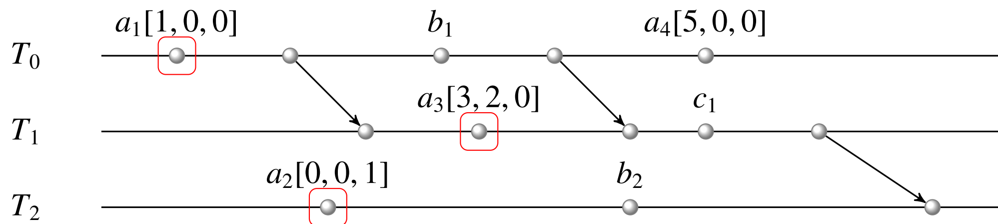


Figure 5.6: The first stored event on each trace is used to calculate *firstIndex*

A following event will need to find the traces that it is following and then check for the existence of a preceding event match on those traces. An event b on $evTrace$ will have a preceding event a in the stored history if the following condition is true.

$$\exists_{i \in [1...n]} b.TS[i] > a.firstIndex[i]$$

The pseudo-code for the follow removal decision is given in Algorithm 9. Recall that the i-th timestamp $TS[i]$ of an event, if positive, gives us its GP on the i-th trace. So we compare *remEvent*'s GP on a trace with the corresponding $firstIndex$ at the node that it follows. If the above condition is true, $remTrace$ has an existing match and so a WR decision is returned. If the condition is false for all the GPs, *remEvent* is an *unmatched* event and so can be removed.

Algorithm 9 Default Removal Rule for Follow

Precondition: *remEvent* is being considered for removal and it follows $folwNode$

```
1: function checkFollow(remEvent, folwNode, recur)
2:     for each trace such that remEvent.TS[trace] > 0 do
3:         if remEvent.TS[trace] > folwNode.firstIndex[trace] then
4:             return WK
5:     return SR
```

earliestGP

The index *earliestGP* keeps track of the earliest among all the GPs to the stored matches at a node. For example in Figure 5.7, we have two matches for the event class B on the trace t_1, namely b_1 and b_2. Their GPs on trace t_0 are s_1 and s_3 respectively. The i-th entry in $earliestGP_t$ stores the $TS[i]$ of the earliest GP to trace t on trace i. So $earliestGP_1[0]$ should be assigned the $TS[0]$ for the event s_1. Since we are calculating it for the node storing event class B, it can be obtained from the $TS[0]$ of b_1, which is the first matched event on trace t_1 with a non-zero value for its $TS[0]$.

The $earliestGP_1$ for node B can then be calculated as $[2, 1, 2]$, in which each entry represents the GP on the corresponding trace: s_1, r_1, and s_2 respectively. If the match b_1 is now removed, s_1 and r_1 are no longer GPs to the existing matches. The event b_2's timestamp is then used to update $earliestGP_1$ to $[3, 4, 2]$.

A preceding event will need to compare two consecutive matches on a trace to see if only the earlier has a corresponding follower event match. It can be found by comparing the *earliestGP* for all the traces in the follower node. Consider two such events a_1 and a_2 on *evTrace*. If a_1

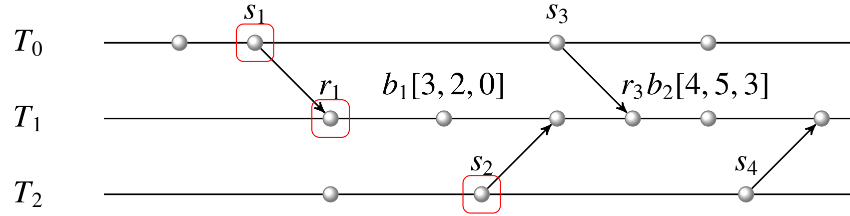


Figure 5.7: The earliest GP to the existing matches for B on trace t_1 are s_1, r_1, and s_2, which are used to calculate $earliestGP_1 = [2, 1, 2]$

is redundant then the *earliestGP* for all traces in the follower node will happen after a_2. The removal condition for a_1 can then be summarized as:

$$a_2.TS[evTrace] \leq \forall_{t \in [1 \ldots n]} b.earliestGP_t[evTrace]$$

Algorithm 10 Default Removal Rule for Precedence

Precondition: *nextEvent* is the immediate next event after the removal event on *evTrace* and *precNode* is the node it precedes

```
1: function CHECKPRECEDENCE(nextEvent, evTrace, precNode, recur)
2:     for each trace do
3:         if nextEvent.TS[evTrace] > precNode.earliestGP[trace][evTrace] then
4:             recur ← false
5:             return WK
6:     return WR
```

The pseudo-code for the precedence removal decision is given in Algorithm 10. If the *earliestGP* for all the traces in a follower node happens after the immediate next event, *remEvent* is a redundant match and so a WR decision is returned. If there exists a trace which has an *earliestGP* that does not follow the *nextEvent*, then the first match on this trace potentially follows *remEvent* and so a WK decision is returned. Notice that it is possible that such an event does not follow *remEvent* either. In that situation, the said match in the follower node will have no precedence match and so will be removed by its own removal rule. The corresponding update in its *earliestGP* will ensure the removal of *remEvent* in the next round.

5.3.3 Event-Based Removal Rules

The concurrency and follow relations only remove events that are considered *unmatched*, while precedence removes events that are *redundant*. Thus the default rules only remove events that can no longer generate new matches or are evidently redundant. These rules will not be sufficient to clear the stored history if there exists a large number of non-redundant precedence matches. In the last chapter we have shown how *potential causality* can be used to decide which events are redundant. It can be used to generate a subset of events, which may still require an exponentially large number of stored events to ensure completeness. *Actual causality*, if known, can help in this situation. This information is only available to the end user and so we expose this decision to the user through a set of removal rules that can be defined while specifying the pattern.

Consider the *atomicity violation* pattern that we used as one of our test cases. We look for a match to the pattern representing two concurrent *thread enter*s. Our goal here is to identify two threads that are executing the critical section at the same time. In that case we do not need to store all the *thread enter* events that have happened in the past. We only need to store those *thread enter* events that can report any future violation.

Going back to Figure 5.3, suppose we define a rule for removing any match to the event-class A when it precedes an event C. In that case, c_1 will cause both a_1 and a_2 to be removed. We call such an event c_1 a *trigger event* and the removal rule for event class A can be defined as

```
A [ →, any, EVENT_C, ‘’] := [‘’, EVENT_A, ‘’];
```

The removal rule specifies that if we have a matched C on *any* trace then all the matches to the A event which are related to it by $A \rightarrow C$ can be removed from the history.

Our grammar allows the use of trigger events with any causal relation to the event being removed. In this work, however, we have restricted the removal to the precedence relation only, i.e., the events that precede the trigger event. One reason for this decision is that the default rules mainly focus on *unmatched* events and an event can never be considered to be *unmatched* for a precedence relation. Thus default rules are inadequate to maintain a compact event history when precedence is present in the pattern. Another reason is that, if the trigger event itself follows the events that are being considered for removal, it only affects past events. Trigger events with a concurrency or follow relation would add additional complexity for removing themselves as we would need to retain them for future events.

There are no indices to maintain for the event-based rules as the removal decision is taken based on the matched trigger event and the causality specified. The pseudo-code for the event-based removal rule is given in Algorithm 11. It checks whether the *remEvent* happens-before the

Algorithm 11 Using Event-Based Rules for Event Removal

Precondition: *trigEvent* points to the event that triggered the removal of *remEvent*.

```
1: function useEventRule(remEvent, trigEvent)
2:     if remEvent precedes trigEvent then
3:         if checkVariables(remEvent, trigEvent) then
4:             return true
5:     return false
```

trigger event and their instantiated attribute variables conform (using the function *checkVariables*). An event is removed when both of these conditions are true.

Now that we have discussed all the components of the event-removal rules, the overall algorithm for history management using Ananke is given in Algorithm 12, omitting the backtracking algorithm for pattern search, since it is similar to the OCEP algorithm described in Section 3.3.4. The pattern search itself is handled in one thread and is called only on arrival of a *terminating event*. *Ananke* executes in a separate thread and it executes in a continual loop in which each round has two distinct parts: i) decision loop (lines 4-15) and ii) removal loop (lines 16-23).

In the decision loop, the two parameters *remNode* and *remTrace* together identify the node and trace from which events are considered for removal. If the node has an event-based rule, *triggerEvent* will point to the trigger event. The removal loop goes through each stored event and calls the functions for the default and event-based removal rules (if one exists) to check the removal decisions. Once a removal decision is reached, it only updates the removal tag in the saved event. We also update the indices that we maintain by calling the function updateIndices (line 12).

Another point to note here is that the history for each node is implemented as a queue data structure. The backtracking algorithm for pattern search uses binary search and backjumping using the timestamp of the stored events. This requires that we store the matched primitive events on a single trace in their total order. We ensure this by removing events only from the front of the queue while newly arriving events are appended to its back. So if *remEvent* does not get removed we break out of the decision loop.

The removal loop is where the events that are tagged in line 11 actually get removed. The two parameters controlling this loop are *clearNode* and *clearTrace*. We simply go through the list of matched events and remove events that are tagged for removal. The actual removals are separated from the decisions to keep the critical section smaller. The pattern-search algorithm and the decision loop can execute concurrently as both only need to read the event data. We synchronize the pattern-search algorithm with the removal loop so that they do not operate on

Algorithm 12 Ananke Algorithm

```
1:  function MANAGEHISTORY
2:      while true do
3:          didRemoval ← false
4:          ACQUIRE(readMutex)                         ▹ Acquire mutex for removal decisions
5:          ▹ Prepare remNode, remTrace, triggerEvent for this round
6:          for each remEvent ∈ remNode.matchList[remTrace] do
7:              didRemoval ← USEDEFAULTRULE(remNode, remTrace, remEvent)
8:              if remNode has event-based rule then
9:                  didRemoval ← USEEVENTRULE(remEvent, triggerEvent)
10:             if didRemoval then
11:                 remEvent.remTag ← true
12:                 UPDATEINDICES()
13:             else
14:                 break
15:         RELEASE(readMutex)
16:         ACQUIRE(updateMutex)                       ▹ Acquire mutex for actual removals
17:         ▹ Prepare clearNode and clearTrace for this round
18:         for each clearEvent ∈ clearNode.matchList[clearTrace] do
19:             if clearEvent.remTag then
20:                 clearNode.matchList[clearTrace].ERASE(clearEvent)
21:             else
22:                 break
23:         RELEASE(updateMutex)
```

the same trace in the same node.

5.4 Building the Patterns for the Test Cases

In Section 3.4.3 we discussed the four test cases that we used for evaluating the performance of the OCEP algorithm. We will be using the same concurrency bug-patterns for evaluating Ananke as well. In this section we discuss how application-specific knowledge for each of the bug patterns can be used to create removal rules for them.

Deadlock

The pattern that OCEP uses to detect a deadlock of length three in an MPI program is:

```
Send1 := [$p1, Send, $p2]
Send2 := [$p2, Send, $p3]
Send3 := [$p3, Send, $p1]
Send1 S1
Send2 S2
Send3 S3
pattern := (($S1 || $S2) || $S3)
```

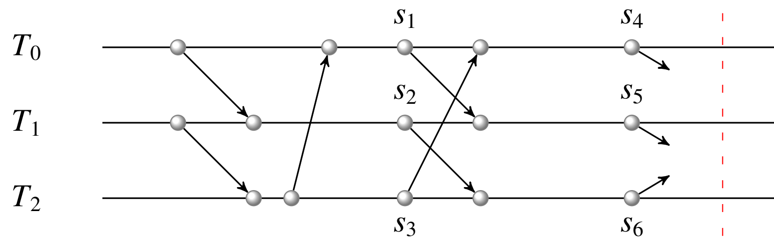


Figure 5.8: Some deadlocks may remain hidden because of network buffering

The pattern tries to identify three *Send* events that are concurrent to each other and have circular dependency. In Figure 5.8 we have two such deadlocks involving the three traces. The first one is a *potential deadlock* as the events s_1, s_2, and s_3 involve messages that can be buffered completely by the network and so they do not get blocked. The events s_4, s_5, and s_6 cause an *actual deadlock* as they operate as blocking *Send* operations.

If we are trying to identify an *actual deadlock* then any *Send* event will remain blocked until it is received at the other trace. In that case no new event will appear on the trace on which a deadlocked *Send* has appeared. Thus we can remove all previously occurring *Send* events on a trace once we find another new event on it. This pattern may look like:

```
Send1[→, same, ‘’, ‘’] := [$p1, Send, $p2]
Send2[→, same, ‘’, ‘’] := [$p2, Send, $p3]
Send3[→, same, ‘’, ‘’] := [$p3, Send, $p1]
Send1 S1
Send2 S2
Send3 S3
pattern := (($S1 || $S2) || $S3)
```

OCEP uses the deadlock pattern to identify potential deadlocks as well. If the network could buffer the message that is being sent, the event *MPI_Send* does not get blocked. In that case we encounter a potential deadlock which will remain undetected for a long time because it is dependent on the size of the message being sent.

The immediately preceding pattern cannot detect this type of deadlock as the *Send* events will not block. Since all the events are concurrent to each other, an event can no longer generate a match once it precedes all the other traces. Thus we can stick to the default removal rule and use the first pattern for identifying this type of deadlock.

Message Race

Message races need to detect two concurrent *Send* events that are communicating with the same process. Thus a *Send* event can only be removed when it is *unmatched* and so we can use the default removal rule for this pattern as well.

```
Sends := [‘’, Send, ‘’]
Receives := [$P1, Receive, ‘’]
Sends S1, S2
Receives R1, R2
pattern := (($S1 ‖ $S2) ∧ ($S1 . $R1) ∧ ($S2 . $R2) ∧ ($R1 → $R2))
```

Atomicity Violation

The events that an atomicity-violation pattern tries to identify are the requests from two different threads to enter a critical section that are granted simultaneously. This can happen when the two requests are not serialized through a semaphore and the two events will then be concurrent. One approach would be to remove the *thread enter* event once the corresponding thread leaves the critical section.

```
M [→, same, thread leave, ‘’ ] := [‘’, thread enter, ‘’]
pattern := (M ‖ M)
```

This pattern, however, will fail to report a violation if the monitoring process receives a *thread leave* event before it receives a violating *thread enter* event. Since the two events are causally independent, in partially ordered event data they may arrive in any order. Instead we can use a rule that removes all previous *thread enter* events on arrival of a new one. Any pending violation that is causally independent to the removed events must also be causally independent to the new *thread enter* event.

```
M [→, same, thread enter, ‘’ ] := [‘’, thread enter, ‘’]
pattern := (M ∥ M)
```

Ordering Bug

Our ordering tries to detect a simulated ZooKeeper bug that occurs when a leader forwards an out-of-sync snapshot to its follower. In this case any matched *Synch* request has a limited scope to match the pattern determined by its corresponding *Forward* event at the leader. Thus a *Forward* event, once found, can trigger event removal from the stored history.

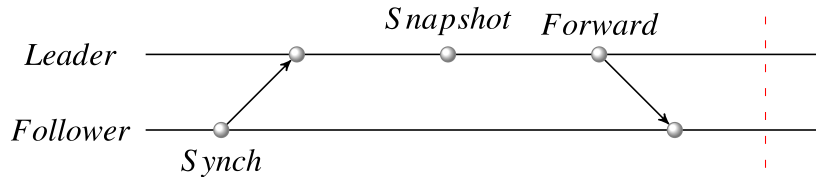


Figure 5.9: The events that follow the desired program behaviour can often be removed

```
Synch [→, different, Forward_Snapshot, $follower] :=
                                [$follower, Synch_Leader, $leader]
Snapshot := [$leader, Take_Snapshot, ‘’]
Update := [$leader, Make_Update, ‘’]
Forward := [$leader, Forward_Snapshot, $follower]
Snapshot $Diff
Update $Write
pattern := ((Synch → $Diff) ∧ ($Diff → $Write) ∧ ($Write → Forward))
```

We add an event-based rule for the *Synch* event that will remove it once a corresponding *Forward* event is found. Notice that we have used the attribute variables *leader* and *follower* to identify the *Forward* event that corrsponds to the *Synch* event from a follower. We have not used the event-based rule for all the event classes in the pattern as once the Synch event is removed, the default rule for the remaining events will take effect.

5.5 Performance Evaluation

5.5.1 Evaluation Methodology

We compare the performance of OCEP with Ananke using the test cases for the same bug patterns that we used in Chapter 3. All measurements are performed on a workstation with an Intel® Core™ i5-3320M 2.6 GHz CPU and 8GB memory running Linux kernel 3.5.0. We use the *reload* feature in POET to execute OCEP and Ananke five times for the trace-event data of each test case and record the wall-clock time taken by the pattern search on arrival of each terminating event. Our first performance metric is the *execution time* for each terminating event, which is measured as the average of these runs. The second performance metric is the *history size*, which is measured as the total number of matches stored at the nodes in the pattern tree.

5.5.2 Results and Discussion

The boxplots for the execution time for our four test cases are given in Figure 5.10. We find that, except for the deadlocks, in 99% of cases we can detect a violation in less than 150 microseconds. A deadlock cycle of length 50 is detected in less than 2 milliseconds in 99% of cases. These boxplots are mostly similar to the ones we obtained for OCEP in Section 3.4. On average, Ananke takes almost same time to detect the matches, except for the deadlock patterns. The exponential nature of the search algorithm, similar to OCEP, is also not visible in these plots. We would like to point out here that the execution time of Ananke is exponential in the length of the pattern (k), not in the number of processes (n). In Figure 5.10(i), we are searching for a deadlock involving all the processes. So both n and k change along the x-axis. But for the other patterns in Figure 5.10, only n changes as the pattern-length remains fixed for all scenarios.

We compare the execution time of Ananke and OCEP in Figures 5.11- 5.14 using the *empirical cumulative distribution function*.

We found that Ananke is able to determine the presence or absence of a deadlock more quickly than OCEP for more than 99% of terminating events. Recall that a deadlock pattern tries to match concurrent *Send* events on all the processes. The backtracking search algorithm will keep traversing the matched events on a trace until it finds that they cannot create a complete match. OCEP is reasonably fast as it uses vector timestamps to prune a large number of matches in the process. Ananke goes one step further by removing those matches that are evidently *unmatched*, which substantially reduces the search space for the vast majority of terminating events.

In Ananke, both the backtracker and the history cleaner are contending for a trace in the same node. Thus waiting time in the search routine may increase when the backtracker has al-

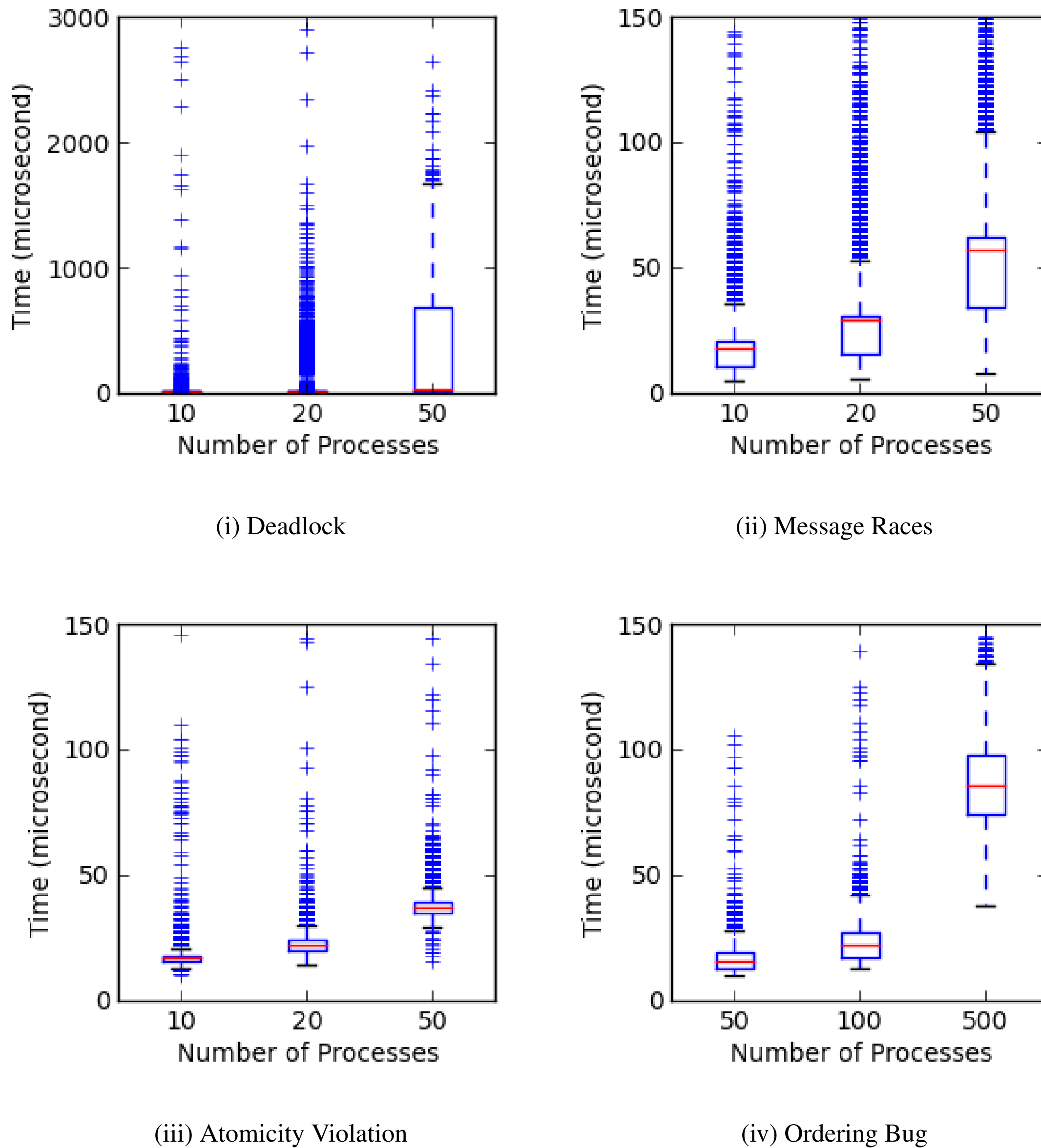
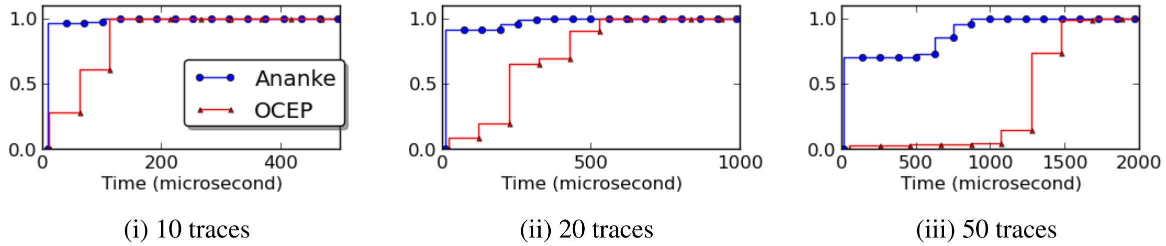


Figure 5.10: Boxplots for execution time with Ananke

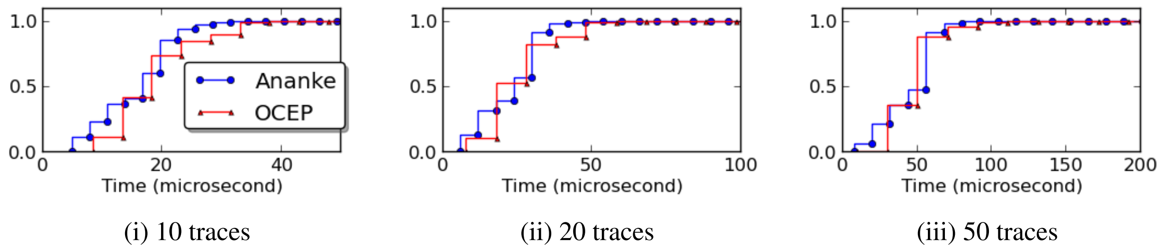
ready built a large partial match. We would like to point out here that the backtracking search routine will wait before beginning its search on a *(node, trace)* pair only if the history thread is



(i) 10 traces (ii) 20 traces (iii) 50 traces

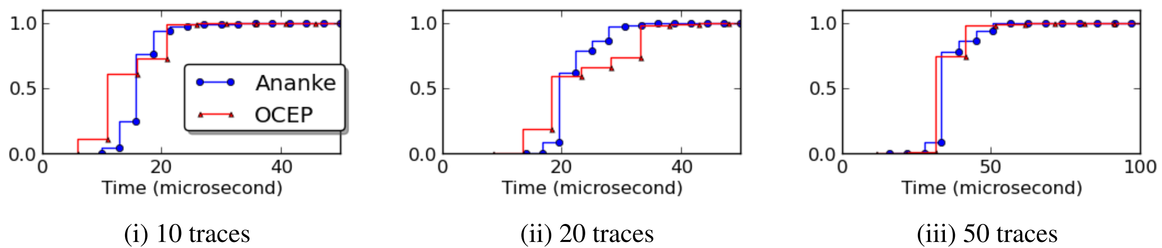
Figure 5.11: Cumulative distribution of execution time for deadlock pattern

already working on the the same *(node, trace)* pair in the removal loop. There is no contention if the history thread is working on that *(node, trace)* pair in its decision loop. Thus some complete matches were reported in average time while the execution times for some other complete matches were outliers.



(i) 10 traces (ii) 20 traces (iii) 50 traces

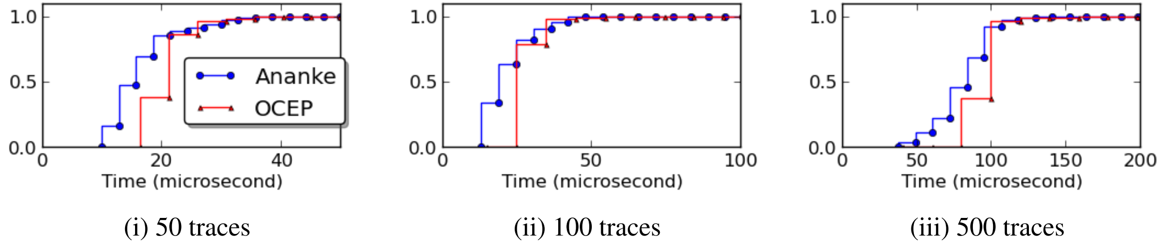
Figure 5.12: Cumulative distribution of execution time for message-races pattern



(i) 10 traces (ii) 20 traces (iii) 50 traces

Figure 5.13: Cumulative distribution of execution time for atomicity-violation pattern

The cumulative distribution of execution times for searching the message-races, atomicity-violation, and ordering-bug patterns are shown in Figures 5.12– 5.14. The execution time for



(i) 50 traces (ii) 100 traces (iii) 500 traces

Figure 5.14: Cumulative distribution of execution time for ordering-bug pattern

these patterns closely matches with OCEP. A match to these patterns involves a small number of traces. Message races involve two senders and one receiver, atomicity violations involve two concurrent clients and the server, and the ordering bugs involve a leader and a follower. OCEP can quickly pick these traces using attribute variables and timestamps and thus is very efficient. Thus the synchronization cost for the event-removal algorithm in Ananke is offset by the small size of the event history, even for relatively small patterns.

We summarize detailed statistics of the execution time of Ananke for all test cases in Figure 5.15. The Q_1 and Q_3 are the first and third quartile respectively. W_1 and W_2 correspond to the whisker marks that we have used in the boxplots and so they are placed 1.5 times the *inter quartile range* (IQR) above and below the Q_1 and Q_3 respectively. Thus we can see that except for the deadlock pattern of length 50, for more than 99% of terminating events, Ananke can report matches to the pattern within 150 microseconds. Ananke can report a deadlock cycle of length 50 within 2 milliseconds in more than 99% of cases. We demonstrated in Figures 5.11– 5.14 that pattern search with Ananke is quicker or similar to OCEP for some frequently occurring concurrency bug patterns. Ananke also reports a representative subset of matches according to the definition in Section 3.3.1.

That leads us to our second performance metric, which is the total number of events that are stored in the history. In Figures 5.16 and 5.17 we show the execution time for each individual terminating event along with the current history size. On the x-axis we plot the number of terminating events that are seen. The y-axis on the left is for the execution time taken and the y-axis on the right is for the number of matches that are stored. In each figure, except for the ordering bug, the number of traces is 10, 20, and 50 respectively, from top to bottom. The number of traces for the ordering bug is 50, 100, and 500, respectively, from top to bottom. Only one execution of the search algorithm for each pattern is plotted in these figures, however, every execution shows similar characteristics when plotted.

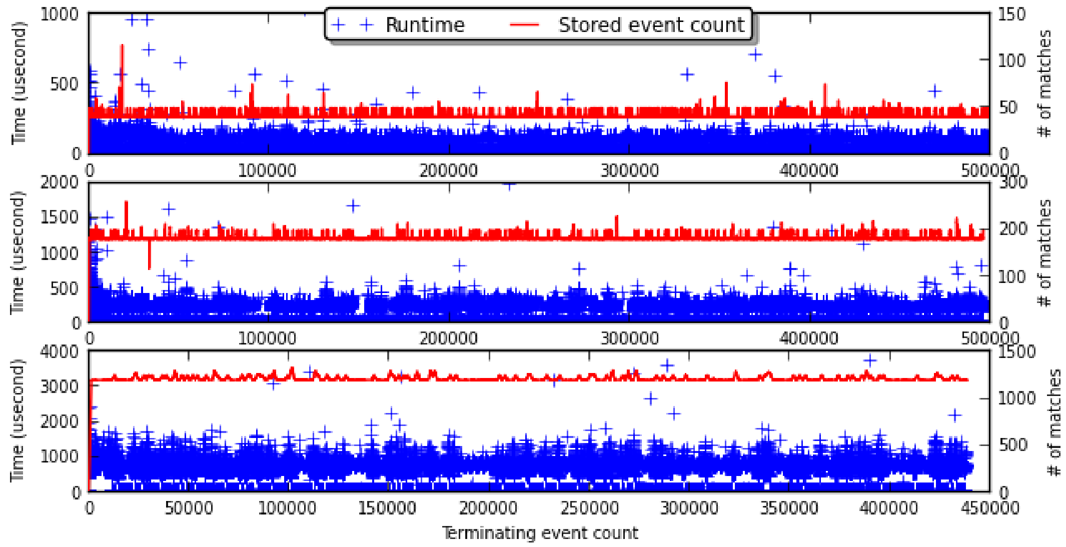
We find that for each pattern the removal rules successfully maintain a bounded history size.

Test Case	Number of Traces	W_1 ($Q_1 - 1.5 \times IQR$)	Q_1	*Median*	Q_3	W_2 ($Q_2 + 1.5 \times IQR$)	*Maximum*
Deadlock	10	10	10	10	12	16	5722
	20	14	14	15	17	22	7899
	50	21	21	21	668	1639	9947
Message Races	10	11	11	18	21	37	8292
	20	16	16	29	31	54	9986
	50	34	34	57	62	105	12465
Atomicity Violation	10	12	16	17	18	22	1685
	20	11	20	22	25	33	3808
	50	26	36	37	42	52	4860
Order Violation	50	13	13	14	19	29	1831
	100	17	17	23	27	43	2020
	500	38	75	87	99	136	3507

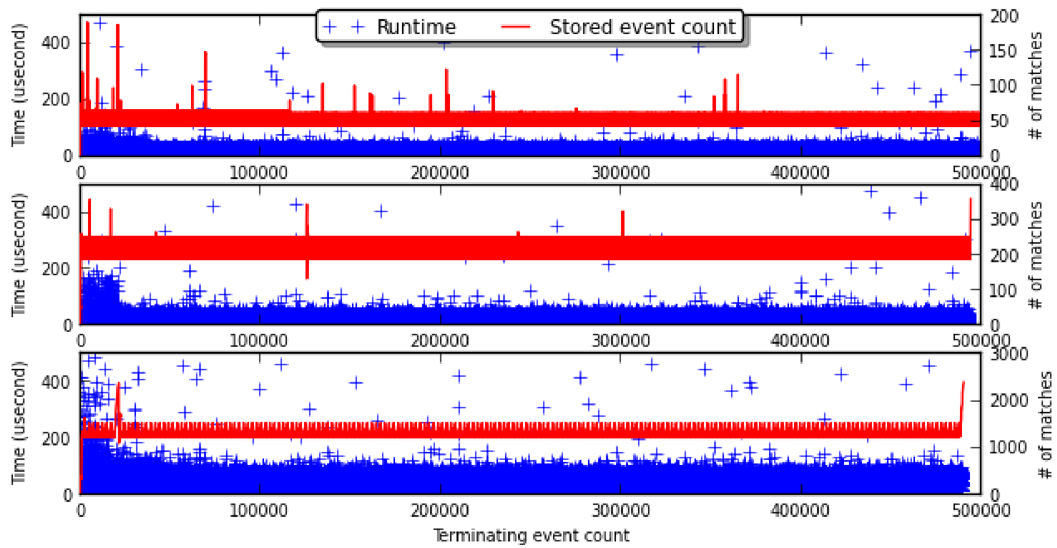
Figure 5.15: Summary statistics for the execution time (μsecond) for all the test cases

We have not shown the history size maintained by OCEP in these figures as these were the worst cases for its simple event-removal algorithm. Recall that we only remove events in OCEP when there are multiple occurrence of the same event with no communication event between them. The first two patterns, deadlock and message race, have communication events as part of the pattern. The *thread enter* events in the atomicity violation will always be followed by communication between the trace and the semaphore. Each event in the ordering bug pattern has attribute variables and so two consecutive events need to have the same attribute values as well as no communication event between them in order to be removed. Thus, for all of these patterns OCEP ends up saving all events since the start of execution. An effective failure detection tool such as OCEP or Ananke should maintain a catalogue of violation patterns, which are connected together to routinely monitor a bigger pattern. OCEP's event removal technique is thus not scalable to monitor arbitrarily long-running applications as the history size may become arbitrarily large.

In Section 5.4 we have shown how we used application-specific knowledge to determine removal rules for all our patterns. We found that it is easier to determine when a concurrent or following event can no longer generate new matches. An event that precedes another, however, can never be considered *unmatched* by an online monitor. Our event-based rules are used to specify the condition for removing this type of event. The behaviour of the application helped us

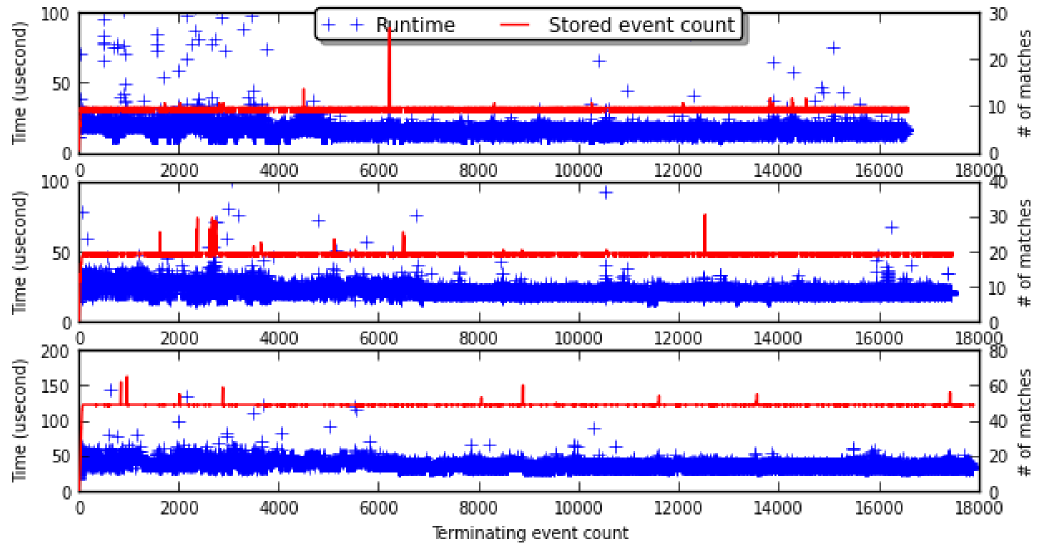


(i) Deadlock

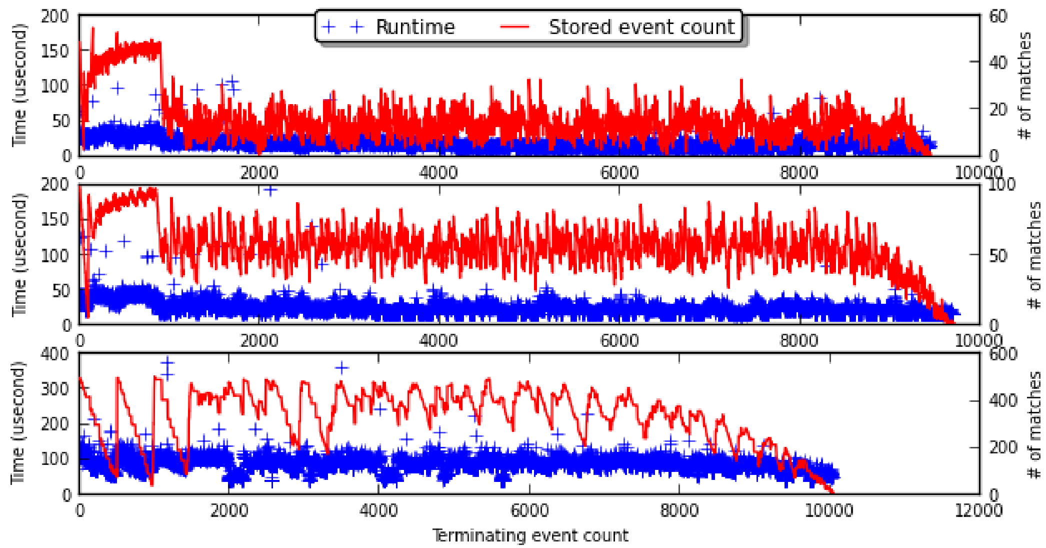


(ii) Message Races

Figure 5.16: History size and individual execution time for deadlock and message races



(i) Atomicity Violation



(ii) Ordering Bug

Figure 5.17: History size and individual execution time for atomicity violation and ordering bug

define a removal rule for the atomicity-violation pattern. The ordering pattern can be thought of as a transaction between the two servers where the last event in a sequence of events marks the end of the transaction. Thus a bug in the pattern is only relevant to a single such sequence and the end event can be used to remove the other events.

The effect of the removal rules is evident in the history-line spikes. These spikes are more prominent for message races and the ordering bug. We synchronized the two threads running pattern search and event removal to give higher priority to the search algorithm. The message-race simulation only has send/receive events and so the history thread has to wait more often.

The ordering-bug pattern has four events of which only the $FORWARD$ event is a terminating event. This $FORWARD$ event is also used in the removal rule and causes all its corresponding matches with the other events to be removed. So the change in the size of the history happens in a step-wise fashion, which becomes more prominent as the number of traces is increased. In the beginning, all the traces that are starting also send a synchronization request to the server. When the number of traces is small, this causes a large spike in the beginning as a number of requests gather before the server starts sending the $FORWARD$ events. There is no such spike for the largest scenario as the server can keep sending snapshots to some traces while synchronization requests from other traces keep coming.

The history size also closely matches the number of traces for the atomicity-violation and the ordering-bug patterns. At the same time there can be only one request from each process and so our removal rule is effectively maintaining a compact history size. The ordering-bug pattern also shows the history size reduce to zero at the end as every matched event sequence has a corresponding $FORWARD$ event.

5.6 Conclusion

OCEP is not scalable to a large number of traces as it ends up storing all the partial matches in a large number of cases. In Chapter 4, we found that a finite-sized history for any generic pattern cannot be obtained if we want to report a match if it exists. As we showed in Section 5.4, if we know the repetitive nature of the searched pattern, we can use it to determine when a partial match can be discarded.

In this chapter we have introduced rules that can be used in the pattern, which allow a user to expose some information about the application itself. We used some of the most frequently occurring types of concurrency bugs in real-world applications to show that the desired causal order of events can be utilized to specify such removal rules. More importantly, these rules were able to maintain a finite history and still report a representative subset of matches within a millisecond in most cases.

Chapter 6

Conclusion

Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.

Edsger W. Dijkstra

The major question that we wanted to answer in this work was whether an online causal-event-pattern-matching algorithm is efficient enough to provide fast detection of violation patterns with low overhead in realistic application settings. Essentially the answer to this question for finding all the matches to any generic pattern is *negative*, which is well documented [Cooper and Marzullo, 1991; Arasu et al., 2004b]. So we reformulated the question as whether an efficient online algorithm exists that is able to report a subset of matches that is *representative* of all matches to the pattern.

Our first objective was to define such a *representative* subset that is applicable in a large number of situations. We ruled out the window-based solution as it fails to report matches that span multiple windows. A minimal approach can be to report at least one match (if it exists). Although one match can detect the presence of a fault in the monitored application, it fails to identify the breadth of the problem. A pattern represents a violation in the distributed application as a collection of causally related events. Our *representative subset* includes at least *one occurrence of each event in the pattern on each process* (if it exists). Thus we can detect if a violation has occurred anywhere in the system.

Ideally we would want a search algorithm that maintains a finite subset of partial matches and reports the desired matches to the pattern as new events appear. We found that an event's vector timestamp is useful in analyzing its causal relation with the other events. Maintaining a finite subset by comparing the vector timestamps, however, was computationally expensive and

we also had to make costly updates on arrival of each *receive* event. Our first few attempts that used this technique were *incomplete* as they failed to report a representative subset.

Our next attempt, OCEP, relaxed the condition of a bounded subset and only removed events that it found are redundant. OCEP uses vector timestamps and the causality relation among the events in the pattern to efficiently prune the search space of the stored partial-matches. An evaluation of the framework using representative patterns from key distributed primitives shows that OCEP is capable of handling the event rates when used to monitor violation of safety conditions in distributed applications.

OCEP, however, is not scalable to a large number of long-running processes as it cannot effectively remove the stored events. We found that maintaining a compact subset of stored events is computationally expensive and does not guarantee a bounded subset. A generic pattern may require storing an exponential number of partial matches for reporting any match if one exists. This finding, although not entirely unexpected, is frustrating for two reasons. Firstly, we are not trying to find all the matches to a pattern and secondly, for a large number of cases it is possible to tell when a partial match can no longer generate any new match to the pattern. Unfortunately, this information is application-specific and hence cannot be used in a generic solution.

Our second algorithm, Ananke, introduces *removal rules* in the pattern itself. The *default rules* use the causal relationship among the events in the pattern to determine which events are *unmatched* or *redundant*. Users can also specify some *event-based rules* in the pattern to declare when a matched event is redundant and can be removed. We showed that these rules can be used to maintain a finite history and still report a representative subset of matches for our test patterns.

If we go back to our initial question, we have found an efficient online algorithm that is able to report a *representative subset* of matches. It can express various types of undesired behaviour in the system using a causal pattern of events. It can also use application-specific knowledge to maintain a finite subset of partial matches. We validated our claim using some of the most frequently found concurrency bug patterns in real-world applications. A generic pattern may still exist for which effective removal rules cannot be specified, resulting in an arbitrarily large number of partial matches. We proved such a pattern is intractable as storing a finite number of partial matches may fail to report some matches. Removal rules can still become helpful for these patterns in constraining the memory overhead.

Finally, the detection of a violation with causal-event patterns requires knowledge of the underlying system's behaviour. Events of possible interest need to be instrumented in advance. We also need an *anticipation* of the causality structure among these monitored events that represents an undesired behaviour. An unexpected behaviour with a different causality structure may remain undetected although it has the same effect on the global property.

6.1 Future Work

Our work can be further extended in various directions.

6.1.1 Extend the Grammar

Offline approaches use some elaborate operators that extend the expressiveness of the grammar. For example, our grammar does not include a *not* operator. It is, however, possible to build a pattern that searches for two events that are not related by a specific causality. For example, $(B \rightarrow A) \vee (B \parallel A)$ can be used to represent $A \nrightarrow B$ and $(B \rightarrow A) \vee (A \rightarrow B)$ can be used to represent $A \nparallel B$.

Instead, we might be searching for a single event for which no causally related match exists. For example, when we say $A! \rightarrow B$, we might be looking for an event a that does not precede any b. This type of pattern can be used to detect *omission faults*. An online matcher that searches for this pattern will have a high volume of false positives as until the b event appears all of its preceding a events will be identified as faults.

One solution can be to extend our limited operator to include an arbitrary event. For example, we can use the pattern $A \xrightarrow{B} C$ to represent omission of b events between two causally related a and c. It is a trade-off and further study of faults in distributed systems is required to assess which additional operators are best suited for an online monitoring tool.

6.1.2 Scalability

We have relied on POET for the vector timestamps of the events that are being monitored. POET is also able to provide these events in a linearization of the partial order, which we have utilized in identifying redundant events. POET only allows efficient monitoring of about 1000 traces [Sheikh, 2012], which limits our ability to monitor large systems. Recently we have seen a lot of interest in NoSQL databases and systems such as Cassandra also provide vector timestamps. We would like to investigate how our event-pattern matcher can be used on top of Cassandra and what role its *eventual consistency* plays in the event removal.

6.1.3 Parallel Algorithm

Ananke and OCEP use a backtracking search-algorithm in which, at each level, we sequentially traverse the traces on which events are stored. Each of these traces represents a subtree in the total search space. This parallelism can be exploited to improve the performance of the algorithm.

Both of our algorithms use a tree-based data structure in which only the primitive events are stored at the leaves. The partial matches of greater length are only created during search time and never stored. We explained in Section 4.4 that a subset of longer partial matches cannot be maintained by only comparing the matches at the node itself. In Section 4.5, we used a finite automaton to represent a pattern, which solves this problem by creating tuples for the *incomplete match*es as well. A finite automaton is more suitable for a parallel algorithm as the complete matches to a pattern can be found from only the states that have a transition to the final state. So, the set of matches can be returned using only these states while the remaining states can use *idle updates* to modify their collection of incomplete matches. Both of these steps can be parallelized as each state only needs the matches stored at itself to form the new matches.

The Finite-automaton-based approach faces the same problem as OCEP, i.e., a bounded-subset of *incomplete match*es will fail to report some matches. Thus a parallel algorithm using a finite automaton will also need to use the removal rules to solve the completeness problem and maintain a finite subset of matches.

6.1.4 Autonomic Computing

An important component of *autonomic computing* is *self-healing*, which detects and repairs existing faults in a system. Self-healing software must have knowledge about the desired behaviour of the system. A search algorithm that can detect a violation of the desired behaviour and send this information to the participating processes is a good starting point for such a system.

6.1.5 Monitoring Application State

In our pattern grammar, we have used attribute variables for an event, which are in effect exposing some state of the application itself. Extending Ananke to incorporate both event causality and application state would significantly extend the class of patterns that can be handled using Ananke. This, however, is a more challenging endeavour compared to the others as the basic building block of a pattern will also include application state.

References

Agarwal, R., Wang, L., and Stoller, S. D. (2005). Detecting potential deadlocks with static analysis and run-time monitoring. In *Proceedings of the First Haifa International Conference on Hardware and Software Verification and Testing*, HVC'05, pages 191–207, Haifa, Israel. Springer-Verlag. 22, 43, 44

Agrawal, J., Diao, Y., Gyllstrom, D., and Immerman, N. (2008). Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 147–160, Vancouver, BC. ACM. 19

Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. (2003). Performance debugging for distributed systems of black boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 74–89, Bolton Landing, NY. 20

Arasu, A., Babcock, B., Babu, S., Cieslewicz, J., Datar, M., Ito, K., Motwani, R., Srivastava, U., and Widom, J. (2004a). STREAM: The Stanford data stream management system. Technical Report 2004-20, Stanford InfoLab. 19

Arasu, A., Babcock, B., Babu, S., McAlister, J., and Widom, J. (2004b). Characterizing memory requirements for queries over continuous data streams. *ACM Transactions on Database Systems*, 29(1):162–194. 19, 56, 105

Arasu, A., Babu, S., and Widom, J. (2003). The CQL continuous query language: Semantic foundations and query execution. Technical Report 2003-67, Stanford InfoLab. 19

Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. (2004). Using Magpie for request extraction and workload modelling. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, OSDI'04, pages 259–272, Seattle, WA. USENIX Association. 20

Basten, T., Kunz, T., Black, J. P., Coffin, M. H., and Taylor, D. (1997). Vector time and causality among abstract events in distributed computations. *Distributed Computing*, 11:21–39. 17

Bates, P. C. and Wileden, J. C. (1983). High-level debugging of distributed systems: The behavioral abstraction approach. *Journal of Systems and Software*, 3(4):255 – 264. 16

Bayley, I. and Zhu, H. (2010). Formal specification of the variants and behavioural features of design patterns. *Journal of Systems and Software*, 83(2):209–221. 47

Birman, K. P. (2005). Overcoming failures in a distributed system. In *Reliable Distributed Systems*, Texts in Computer Science, pages 247–275. Springer New York. 22

Bitner, J. R. and Reingold, E. M. (1975). Backtrack programming techniques. *Communications of the ACM*, 18(11):651–656. 33

Buhr, P. A., Ditchfield, G., Stroobosscher, R. A., Younger, B. M., and Zarnke, C. R. (1992). μC++: Concurrency in the object-oriented language C++. *Software: Practice and Experience*, 22(2):137–172. 42

Chandy, K. M. and Lamport, L. (1985). Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75. 13

Chen, M. Y., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A., and Brewer, E. (2004). Path-based failure and evolution management. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, NSDI'04, pages 309–322, San Francisco, CA. USENIX Association. 20

Cheriton, D. R. and Skeen, D. (1993). Understanding the limitations of causally and totally ordered communication. *SIGOPS Operating Systems Review*, 27(5):44–57. 79

Cheung, H. (1989). *Process and Event Abstraction for Debugging Distributed Programs*. PhD thesis, University of Waterloo, Waterloo, Ontario. 11

Cooper, R. and Marzullo, K. (1991). Consistent detection of global predicates. In *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, PADD '91, pages 167–174, Santa Cruz, CA. ACM. 1, 13, 15, 105

Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2011). *Distributed Systems: Concepts and Design*, pages 1, 621. Addison Wesley, fifth edition. 5

Demers, A., Gehrke, J., Hong, M., Panda, B., Riedewald, M., Sharma, V., and White, W. (2007). Cayuga: A general purpose event monitoring system. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research*, CIDR '07, pages 412–422, Asilomar, CA. www.cidrdb.org. 19

Farchi, E., Nir, Y., and Ur, S. (2003). Concurrent bug patterns and how to test them. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, Nice, France. IEEE Computer Society. 41

Fidge, C. (1991). Logical time in distributed computing systems. *Computer*, 24(8):28–33. 11, 24

Fidge, C. (1996). Fundamentals of distributed system observation. *IEEE Software*, 13(6):77–83. 6, 9

Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. (2007). X-Trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, NSDI'07, pages 271–284, Cambridge, MA. USENIX Association. 21

Fox, M. (1998). Event-Predicate Detection in the Monitoring of Distributed Applications. Master's thesis, University of Waterloo, Waterloo, Ontario. 22

Frigge, M., Hoaglin, D. C., and Iglewicz, B. (1989). Some implementations of the box plot. *The American Statistician*, 43(1):50–54. 43

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, first edition. 47

Garg, V. K. and Waldecker, B. (1994). Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307. 15

Geels, D., Altekar, G., Maniatis, P., Roscoe, T., and Stoica, I. (2007). Friday: Global comprehension for distributed replay. In *Proceedings of the 4th USENIX Conference on Networked Systems Design and Implementation*, NSDI'07, pages 285–298, Cambridge, MA. USENIX Association. 21

Haban, D. and Weigel, W. (1988). Global events and global breakpoints in distributed systems. In *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences*, pages 166–175, Kailua-Kona, HI. 16

Herlihy, M. P. and Wing, J. M. (1990). Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12:463–492. 1

Horowitz, E. and Sahni, S. (1978). *Fundamentals of Computer Algorithms*, page 334. Computer Science Press, first edition. 28

Hunt, P., Konar, M., Junqueira, F., and Reed, B. (2010). Zookeeper: Wait-free coordination for Internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, ATC '10, Boston, MA. USENIX Association. 47

Isaacs, R. and Barham, P. (2002). Performance analysis in loosely-coupled distributed systems. In *7th CaberNet Radicals Workshop*, Bertinoro, Italy. 20

Jaekl, C. (1996). Event-Predicate Detection in the Debugging of Distributed Applications. Master's thesis, University of Waterloo, Waterloo, Ontario. 26

Joyce, J., Lomow, G., Slind, K., and Unger, B. (1987). Monitoring distributed systems. *ACM Transactions on Computer Systems*, 5(2):121–150. 5

Jula, H., Tralamazza, D., Zamfir, C., and Candea, G. (2008). Deadlock immunity: Enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 295–308, San Diego, California. USENIX Association. 22

Krammer, B., Bidmon, K., Müller, M. S., and Resch, M. M. (2003). MARMOT: An MPI analysis and checking tool. In *Proceedings of the 2003 International Conference on Parallel Computing*, PARCO '03, pages 493–500, Dresden, Germany. Elsevier. 22

Kranzlmüller, D. and Schulz, M. (2002). Notes on nondeterminism in message passing programs. In *9th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 357–367, Linz, Austria. Springer-Verlag. 44

Kunz, T. (1994). *Abstract Behaviour of Distributed Executions with Applications to Visualization*. PhD thesis, Technische Hochschule Darmstadt, Darmstadt, Germany. 2

Kunz, T., Black, J. P., Taylor, D., and Basten, T. (1997). POET: Target-system independent visualizations of complex distributed application executions. *Computer Journal*, 40(8):499–512. 40

Lamport, L. (1978). Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565. 6, 9, 10

Lamport, L. (1986). On interprocess communication, part I: Basic formalism, part II: Algorithms. *Distributed Computing*, 1:77–101. 16

Lee, K. H., Sumner, N., Zhang, X., and Eugster, P. (2011). Unified debugging of distributed systems with Recon. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference*

on Dependable Systems and Networks, DSN '11, pages 85–96, Hong Kong, China. IEEE Computer Society. 21

Liu, X., Guo, Z., Wang, X., Chen, F., Lian, X., Tang, J., Wu, M., Kaashoek, M. F., and Zhang, Z. (2008). D^3S: Debugging deployed distributed systems. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI '08, pages 423–437, San Francisco, CA. USENIX Association. 22

Lo, D., Cheng, H., Han, J., Khoo, S.-C., and Sun, C. (2009). Classification of software behaviors for failure detection: A discriminative pattern mining approach. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, pages 557–566, Paris, France. ACM. 47

Lu, S., Park, S., Seo, E., and Zhou, Y. (2008). Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, Seattle, WA. ACM. 41, 47, 50

Luckham, D. (2002). *The Power of Events*. Addison-Wesley. 19

Magee, J. and Kramer, J. (2006). *Concurrency: State Models and Java Programming*, page 2. John Wiley & Sons, second edition. 6

Mansouri-Samani, M. and Sloman, M. (1997). GEM: A generalized event monitoring language for distributed systems. *Distributed Systems Engineering*, 4:96–108. 19

Mattern, F. (1988). Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. North-Holland / Elsevier. 11, 12, 24

Miller, B. P. and Choi, J.-D. (1988). Breakpoints and halting in distributed programs. In *8th International Conference on Distributed Computing Systems*, pages 316–323, San Jose, California. IEEE Computer Society. 16

Mittal, N. and Garg, V. K. (2001). On detecting global predicates in distributed computations. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, Phoenix (Mesa), AZ. 1

Netzer, R. H. B. and Miller, B. P. (1992). Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, Supercomputing '92, pages 502–511, Minneapolis, Minnesota. IEEE Computer Society Press. 45

Nichols, M. (2008). *Efficient Pattern Search in Large, Partial-Order Data Sets*. PhD thesis, University of Waterloo, Waterloo, Ontario. 17, 20, 26

Nichols, M. and Taylor, D. (2005). A mechanism for visualizing TCP-socket interactions. In *Proceedings of the 2005 Centre for Advanced Studies Conference (CASCON)*, pages 212–224, Markham, Ontario. IBM Press. 41

Park, M.-Y., Shim, S. J., Jun, Y.-K., and Park, H.-R. (2007). MPIRace-Check: Detection of message races in MPI programs. In *Proceedings of the 2nd International Conference on Advances in Grid and Pervasive Computing*, GPC'07, pages 322–333, Paris, France. Springer-Verlag. 22, 45

Peled, D. A. (2001). *Software Reliability Methods*, page 98. Springer-Verlag, first edition. 9

Pramanik, S., Taylor, D., and Wong, B. (2013). Towards an efficient online causal-event-pattern-matching framework. In *Proceedings of the 33rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 481–490, Philadelphia, PA. 23

Prosser, P. (1993). Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299. 33

Reynolds, P., Killian, C., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat, A. (2006). Pip: Detecting the unexpected in distributed systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation*, NSDI'06, pages 115–128, San Jose, CA. USENIX Association. 20

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., and Anderson, T. (1997). Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411. 22

Schwarz, R. and Mattern, F. (1994). Detecting causal relationships in distributed computations: in search of the holy grail. *Distributed Computing*, 7(3):149–174. 2, 6, 8, 9, 16

Sheikh, M. B. (2012). Online Trace Reordering for Efficient Representation of Event Partial Orders. Master's thesis, University of Waterloo, Waterloo, Ontario. 107

Singh, A., Roscoe, P. M. T., and Druschel, P. (2006). Using queries for distributed monitoring and forensics. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems*, EuroSys '06, pages 389–402, Leuven, Belgium. ACM. 22

Slauenwhite, P. E. (2007). Pattern Matching in Generic Event Data. Master's Essay. University of Waterloo, Waterloo, Ontario. 24

Snir, M., Dongarra, J., Kowalik, J. S., Huss-Lederman, S., Otto, S. W., and Walker, D. W. (1998). *MPI: The Complete Reference*. The MIT Press, second edition. 42, 43

Taylor, D. (1999). The POET prototype: Structure and operation. 40

Taylor, D. (2005). Scrolling partially ordered event displays. *Journal of Parallel and Distributed Computing*, 65(5):643–653. 41

Taylor, D., Kunz, T., and Black, J. P. (1996). A tool for debugging OSF DCE applications. In *20th Computer Software and Applications Conference*, pages 440–446, Seoul, Korea. IEEE Computer Society. 40, 46

Taylor, D. and Xie, P. (2004). Specifying and locating hierarchical patterns in event data. In *Proceedings of the 2004 Centre for Advanced Studies Conference (CASCON)*, pages 81–95, Markham, Ontario. IBM Press. 17

Terry, D., Goldberg, D., Nichols, D., and Oki, B. (1992). Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, pages 321–330, San Diego, CA. ACM. 19

Vetter, J. S. and de Supinski, B. R. (2000). Dynamic software testing of MPI applications with Umpire. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, Supercomputing '00, article no. 51, Dallas, TX. IEEE Computer Society. 22

Vitter, J. S. (2001). External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271. 56

Wang, L. and Stoller, S. D. (2006). Accurate and efficient runtime detection of atomicity errors in concurrent programs. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 137–146, New York, NY. ACM. 22, 46

Xie, P. (2003). Convex-Event Based Offline Event-Predicate Detection. Master's thesis, University of Waterloo, Waterloo, Ontario. 2, 17, 20

ZooKeeper (2010). Bug# 962. `https://issues.apache.org/jira/browse/ZOOKEEPER-962`. Accessed: 10/10/2012. 47