Software Fault Tolerance in Distributed Systems Using Controlled Re-execution

by

Ashis Tarafdar, B.Tech., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

August 2000

Software Fault Tolerance in Distributed Systems Using Controlled Re-execution

Approved by
Dissertation Committee:

To my parents

Acknowledgments

I consider myself to have been an extremely fortunate Ph.D. student in having Vijay Garg as my Ph.D. supervisor. It is impossible to fully express the many ways in which he has helped to make my Ph.D. experience a fulfilling and enjoyable phase of my life. He has been a constant source of inspiration, encouragement, and guidance in my research work and, thanks to him, I have been able to freely explore new ideas and work on what is fun. Moreover, he has been a friend, encouraging me in my non-Ph.D. related endeavours as well.

This dissertation has been shaped by discussions that I have had at various times with fellow students. I am indebted to Rajmohan Rajaraman and Kedar Namjoshi for discussions early in my Ph.D., which inspired me to follow up on my early ideas. Sriram Rao and Om Damani have always been there for me whenever I needed someone to bounce my ideas off of. Neeraj Mittal has helped me at various times by reviewing my papers and providing me with valuable criticism that has greatly improved my work. I am also grateful to Sandip Ray who has helped me by providing a critical ear during the final stages of my dissertation through which I learned the lesson of "not claiming too much".

My committee members, Mohamed Gouda, Harrick Vin, Lorenzo Alvisi, Craig Chase, and Keith Marzullo, have also helped to shape my Ph.D. into its current form through their valuable comments and criticisms. Their varied expertise has provided me with different perspectives from which to re-evaluate my work.

I am grateful to my friends in Austin for making my Ph.D. a very enjoyable experience. My ever-cheerful office-mates, Seldron Geziben and Sandip Ray, have helped to make my working environment not seem like one. Fun evenings, weekend trips, late-night discussions, intense gym workouts are but some of the memorable experiences that I have variously shared with Nandan Nayampally, Om Damani, Seema Damani, Landy Haile, Sriram Rao, Vivek Nagaraj, Elizabeth Tolley, Shalu Srinivasan, Meghan Lessor, Paul Lessor, Tyrell Williams, Noelle Hairston, Madhukar Korupolu, Tushar Parikh, Rajmohan Rajaraman, Pawan Goyal, Prashant Shenoy, Mukesh Khare, Kedar Namjoshi, Anuj Gosalia, Aamir Nawaz, Tarun Anand, and C. Bala Kumar.

The biggest credit goes to my family for their love and support. My late father, Sankar Tarafdar, an astrophysicist, inspired my interest in research from an early age. My mother, Anima Tarafdar, has kept me going by her never-ending confidence in me. My brother, Shantanu Tarafdar, has helped me with practical advice at various stages. My other brother, Soumen Tarafdar, has helped to keep me smiling through the hard times. Without my family, this dissertation would never have been written.

ASHIS TARAFDAR

The University of Texas at Austin
August 2000

Software Fault Tolerance in Distributed Systems Using Controlled Re-execution

Publication No. _______________

Ashis Tarafdar, Ph.D.
The University of Texas at Austin, 2000

Supervisor: Vijay K. Garg

Distributed applications are particularly vulnerable to synchronization faults. An important approach to tolerating synchronization faults is *rollback recovery*, which involves restoring a previous state and re-executing. Existing rollback recovery methods depend on chance and cannot guarantee that synchronization faults do not recur during re-execution. We propose a new rollback recovery method, *controlled re-execution*, based on selectively adding synchronizations during re-execution to ensure that synchronization faults do not recur. The controlled re-execution method gives rise to three interesting questions: How do we determine the synchronizations that ensure a safe re-execution? How do we monitor an application to detect faulty global conditions? How well does controlled re-execution perform in practice?

The first part of the dissertation addresses the *predicate control problem* which takes a computation and a global property and adds synchronizations to the computation to maintain the property. We design efficient algorithms to solve the problem for many useful predicates, including disjunctive predicates and various types of mutual exclusion predicates. These predicates correspond to commonly encountered synchronization faults such as races.

The second part of the dissertation investigates the *predicate detection* problem which involves determining whether a global property occurs in a computation. We address the problem for the useful class of conjunctive predicates and in the context of an *extended* model of computation that allows improved predicate detection over the conventional model. We show that, in general, the problem is NP-Complete. However, an efficient solution is demonstrated for the useful cases of *receive-ordered* and *send-ordered* computations. Further, this solution can be used to achieve an improved, though exponential, solution for the general problem.

The third part of the dissertation involves an experimental study of the controlled re-execution method. We evaluate the controlled re-execution method in tolerating race faults and find that the extra tracing costs imposed are within tolerable limits and that it greatly enhanced the likelihood of recovery. We conclude that controlled re-execution is an effective and desirable method for tolerating races in long-running non-interactive distributed applications.

Contents

Chapter 1

Introduction

Distributed applications are difficult to write correctly. An important reason is that, in addition to the usual software faults, distributed applications suffer from synchronization faults such as race conditions. Such synchronization faults are particularly hard to detect during testing and, therefore, are responsible for a significant fraction of application failures [IL95]. As a result of these difficulties many critical software packages are not available in distributed form, in spite of potential performance benefits [Pan95].

Software fault tolerance [1] offers a complementary approach to dealing with software faults. Whereas fault *prevention* techniques are applied during the design, implementation, and testing phases of the software cycle to prevent faults from persisting in the operation phase, fault *tolerance* techniques are applied during the operation phase to minimize the damage caused by those faults that do persist.

A software fault tolerance system relates to an application in two ways: as an *observer* during normal operation, and as a *controller* during failure recovery. The observation aspect of the fault tolerance system involves monitoring the application for the early detection of a failure. Failure detection depends on the type of failure involved. For example, heartbeat protocols aim at detecting crash failures, and

[1] We use *software fault tolerance* to denote software-fault tolerance (the tolerance of software faults) and not software fault-tolerance (tolerance techniques implemented in software).

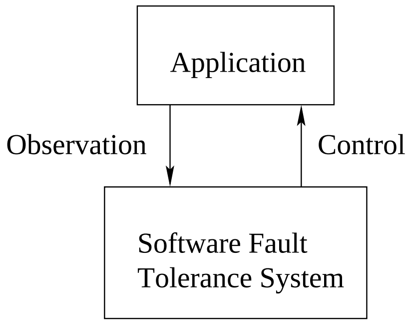

Figure 1.1: Observation and Control

data integrity checks aim at detecting data corruption failures. On the other hand, the control functionality of a fault tolerance system has usually involved general-purpose strategies, such as failing over to a backup process or restarting from a previously stored checkpoint. One of our aims is to show that, using information about the failure and its causes, specialized control schemes can take more effective corrective action than general-purpose schemes. In particular, our focus will be on failures arising from synchronization faults, an important category in the context of distributed applications.

A particular execution of an application is determined by three factors: the application program, the data provided as input, and the environment in which the program is executed. This corresponds to three approaches to tolerating software faults in a given execution. The *design diversity* approach [AC77, Ran75] uses redundant modules that are independently developed so that a failed execution could be recovered by resorting to an alternative module. The main shortcomings of this approach are the high cost in programming effort and the critical assumption that the same fault will not appear in independently developed implementations for the same function. The *data diversity* approach [AK88] uses application-specific techniques to re-express the input data in an attempt to provide an alternative failure-free execution. However, this technique is only applicable in cases for which the re-expression of input data is possible and such a re-expression can help bypass a fault. Our focus is on the third approach, *environment diversity* [WHF+97], which rolls back the application to a previous state and re-executes so that changes in environmental conditions may induce a distinct and failure-free re-execution. This approach assumes that a fault may not recur each time an application is re-executed with the same inputs. It has been observed in several studies [Ada84, Cri91, GR93, IL95] that many software faults exhibit such behavior. In particular, synchronization faults belong to this class since they are sensitive to an environmental determinant – the relative timing between processes.

The environment diversity approach to software fault tolerance is an application of the *rollback recovery* method [EAWJ99] to software faults. Rollback recovery of a failed application occurs in three distinct phases:

- **Detection:** The detection of failures is more difficult in distributed applications because of the challenge of detecting global conditions on distributed state. While it is sometimes sufficient to check for local conditions on a single process, early detection of failures often requires the detection of global conditions. For example, a load imbalance across the distributed application could be an early indicator of potential failures. In particular, detecting synchronization failures, such as races, involves global rather than local detection. The problem of detecting a global condition in a distributed execution has been formalized as the *predicate detection problem* [BM93, Gar96].

  Predicate detection is usually specified in a model of distributed computation that consists of a partially ordered set of events with each process viewed as a sequence of events [Lam78]. However, this model causes "false causality" [ACG93, CS93, SBN+97], causing some global states that satisfy the predicate to go undetected. An extended model has been proposed [ACG93] to correct this problem by viewing a process as a partially ordered set of events. In

Chapter 4 of this dissertation, we investigate the predicate detection problem in the framework of this extended model of a distributed computation.

- **Restoration:** The problem of restoring a failed distributed application to a previous state has been the central focus in studies of rollback recovery [EAWJ99]. While this dissertation does not deal with this problem, we briefly note the implications of software faults in the context of general-purpose restoration techniques. It has been observed [CC98b] that failures arising from software faults are not usually *fail-stop*, that is, faulty data may have been written to stable storage before a failure is detected. Therefore, instead of focusing on restoring the *latest* restorable state, the focus is on the important problem of determining *which* state to restore. The interested reader is referred to [WHF+97] and [CC98a] for interesting and extensive studies of this issue.

- **Re-execution:** The re-execution phase involves re-executing the application from a restored state to avoid a recurrence of a failure. In most rollback recovery techniques, this consists of simply restarting the application from the restored state. This method is effective under the assumption that failures are unrelated. While in some failures, such as power outages, this assumption may hold, failures due to software faults are usually not unrelated. Therefore, it is desirable to determine re-execution methods that explicitly attempt to *control* the application during re-execution so as to bypass software faults and avoid the recurrence of a failure.

  In this dissertation, our focus is on the class of synchronization faults. These faults are known to be an important source of failures [IL95]. In the context of synchronization faults, we propose a novel re-execution method, *controlled re-execution*, that uses trace information to appropriately synchronize during re-execution to prevent a recurrence of a synchronization failure. Unlike re-execution methods that depend on chance, controlled re-execution can guarantee in advance that the re-execution will be failure-free. In Chapter 5, we experimentally evaluate the controlled re-execution method in the case of race faults, an important subset of synchronization faults. The controlled re-execution method is based on a general problem, the *predicate control problem*, which involves controlling a given computation by adding synchronizations such that a given global property is maintained. In Chapter 3, we formulate and investigate the predicate control problem.

To summarize in a different order, our goals are:

- To formulate and investigate the predicate control problem.
- To investigate the predicate detection problem in a general model.
- To specify and experimentally evaluate controlled re-execution.

In the next three sections, we give an introduction to our work towards each of these goals. This is followed by a short summary and an overview of the dissertation.

1.1 The Predicate Control Problem

Informally, the predicate control problem is to determine how to add synchronizations to a distributed computation so that it maintains a global predicate (property). As an example, consider the computation shown in Figure 1.2(a) with three processes, P1, P2, and P3. The processes have synchronized by sending messages to one another. Suppose the stated global predicate is the mutual exclusion predicate, so that no two processes are to be in critical sections (labeled CS1, CS2, CS3, and CS4) at the same time. Clearly, the given computation does not maintain mutual exclusion at all times. Figure 1.2(b) shows the same computation with added synchronizations that ensure that mutual exclusion is maintained at all times. We call such a computation a "controlling computation". The main difficulty in determining such a controlling computation lies in adding the synchronizations in such a manner as to maintain the given property without causing deadlocks with the existing synchronizations.

The predicate control problem is given the input computation *a priori* whereas conventional synchronization problems are given the computation on-line. In this respect, the predicate control problem is the *off-line variant* of on-line synchronization problems. For example, the mutual exclusion problem has been widely studied in its on-line version [Ray86] but not in its off-line version – mutual exclusion predicate control [2] It is to be expected that, since the computation is not known before-hand, the on-line synchronization problems are harder to solve than their off-line counterparts. For example, in on-line mutual exclusion, one cannot, in general, avoid deadlocks without making some assumptions (e.g. critical sections do not block). Thus, on-line mutual exclusion is impossible to solve. To understand why this is true, consider the scenario in Figure 1.2(a). Any on-line algorithm, being unaware of the future computation, would have a symmetric choice of entering CS1 or CS2 first. If CS2 is entered first, it would result in a deadlock. An off-line algorithm, being aware of the future computation, could make the correct decision to enter CS1 first and add a synchronization from CS1 to CS2. Thus, while it is often impossible to synchronize on-line, it may be possible to synchronize off-line. Our aim in studying the predicate control problem, is to determine when it is possible to synchronize off-line and how efficiently this can be done.

While the main focus in this dissertation is the software fault tolerance application, the predicate control problem has potential applications in other domains. An example application is distributed debugging in which a computation is often known *a priori* for the purpose of replaying. Applying predicate control in this

[2] Note that we do not use the term "off-line" to imply that predicate control must be applied *before* run-time. It merely means that whenever it is applied, it is given the computation *a priori*.

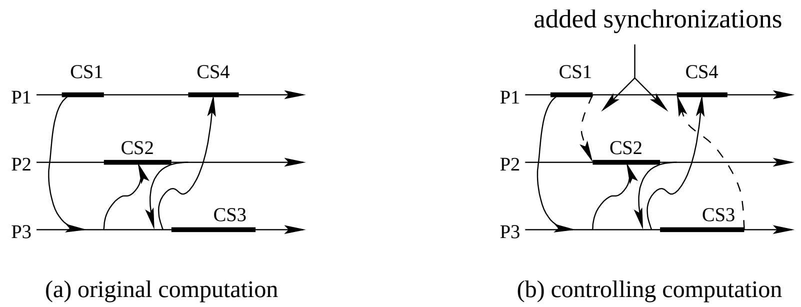

Figure 1.2: The Predicate Control Problem

context would allow the programmer to observe the replayed computation under various constraints, facilitating the localization of software faults. Another potential application is off-line scheduling, in which a set of parallel tasks are to be executed with inter-task dependencies known before-hand. Predicate control allows automatic scheduling of the tasks to satisfy a given global constraint.

Contributions

Since, in its full generality the predicate control problem deals with *any* predicates, it is not surprising that we find the problem to be NP-Complete. It is, therefore, important to study the predicate control problem in the context of specific and useful predicates.

The first type of predicates that we consider is the class of "disjunctive predicates". Intuitively, these predicates state that at least one property must be maintained at all times or, in other words, that a bad combination of events does not occur. Some examples of these predicates are: "at least one server is available" and "at least one process has a token". We determine the necessary and sufficient conditions under which predicate control can be solved for disjunctive predicates. We then describe an $O(np)$ algorithm to solve disjunctive predicate control, when possible, where n is the number of processes and p is bounded by the number of events in the computation.

The next important class of predicates are "mutual exclusion predicates". These are particularly important in software fault tolerance since they exactly correspond to bypassing races. (Note: we use the term "race" to denote concurrent accesses of a shared object. This is sometimes called a "data race" and is distinct from other types of races, especially message races and producer-consumer races). We determine the necessary and sufficient conditions for solving predicate control for the mutual exclusion predicate and describe an efficient algorithm. It is interesting that, though the algorithm is very different from the one for disjunctive predicate control, the complexity is similar – $O(np)$, where p is the number of critical sections in the computation.

The mutual exclusion predicate has two important extensions: "readers writers predicates" and "independent mutual exclusion predicates". Readers writers predicates correspond to mutual exclusion with the semantics of read and write critical sections. Independent mutual exclusion has the semantics of critical sections that have different types, where only critical sections of the same type are mutually exclusive. We determine the necessary and sufficient conditions for solving for readers writers predicates and construct an $O(np)$ algorithm that generalizes the algorithm for mutual exclusion. Unfortunately, we find that, for independent mutual exclusion predicates, predicate control is NP-Complete. However, we are able to solve it under certain conditions, using an algorithm similar to the one for readers writers predicates.

1.2 The Predicate Detection Problem

The predicate detection problem [BM93, Gar96] is usually specified in a *happened before* model of distributed computation that consists of a partially ordered set of events with each process viewed as a sequence of events [Lam78]. Consider a happened-before model representation of a certain distributed computation shown

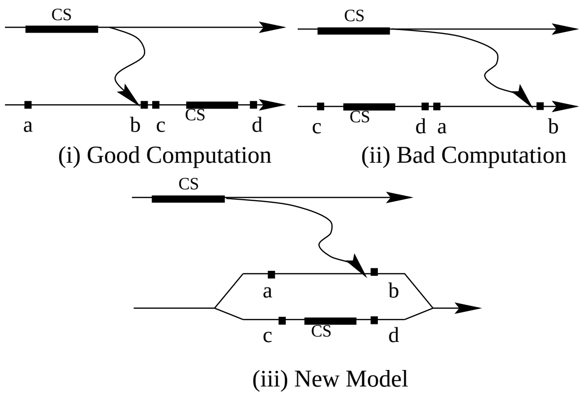

Figure 1.3: Example: Addressing False Causality

in Figure 1.3(i). If mutual-exclusion violation is the predicate that we are trying to detect, it would not be detected because the message ensures that the two critical sections cannot occur at the same time. However, in reality, the message may have been fortuitous and may have induced *false causality* between the critical sections which leads to missing the violation of mutual exclusion. The sections of execution marked by intervals (a, b) and (c, d) may be "independent" (for example, independent threads). The scenario in Figure 1.3(i) is just one possible scheduling of events. Figure 1.3(ii) shows another scheduling in which mutual-exclusion would be violated.

The happened before model was extended [ACG93] to a model that partially orders the events in a local process, allow events within a process to be independent. Figure 1.3(iii) shows such a model for the example. This representation models both of the previous schedulings. In general, there would be an exponential number of happened-before representations corresponding to a single representation in the new model. We refer to the new model as the *extended model*. Thus, the extended model reduces the false causality problem and allows better detection of predicates.

While it is desirable to solve the predicate detection problem in the extended model, most of the algorithms for predicate detection have been framed in the happened before model. What, then, are the implications of solving predicate detection in the extended model?

Contributions

We focus on the important class of "conjunctive predicates" which can be solved efficiently in the happened-before model [GW94]. Conjunctive predicates are specified as conjunctions of local predicates. Thus, they express the combined occurrence of local events. Note that conjunctive predicates are the negations of disjunctive predicates. Some examples of these predicates are: "all servers are unavailable" and "no process has a token".

We demonstrate that for general computations (in the extended model), the problem is NP-Complete. This is an indication of how using the extended model makes predicate detection a harder problem. However, for certain restricted, but useful, classes of general computations, the problem may be solved efficiently. These restricted classes correspond to computations that have either the receive events or the send events totally ordered, while allowing all other events to be partially ordered. Some natural programming styles produce computations that fall in these categories. Further, we can decompose a general computation into a set of general computations in these classes, to achieve an improved (though exponential) algorithm.

It is definitely harder to solve predicate detection in the extended model than in the happened-before model. However, even being able to solve predicate detection efficiently for a restricted class is a great improvement over the alternative of solving predicate detection on an exponential number of corresponding happened-before representations. In the worst case where the general computation does not have either totally ordered sends or totally ordered receives, we may decompose the problem into extended computations each of which belongs to one of these classes and so efficiently solve for each such diagram. Even though this solution is exponential, it still achieves, in general, an exponential reduction as compared to considering all possible happened-before representations.

1.3 Controlled Re-execution: An Experimental Study

In this study, we focus on long-running, non-interactive applications that communicate using messages and use a common network file system. Such applications are suitable for rollback recovery approaches. Typical examples of such applications are scientific programs and simulations. Further, we focus on a particular type of synchronization fault — "races". A race occurs when two processes access a file concurrently. Consider an execution that has races, such as the execution shown in Figure 1.2(a) (viewing the file accesses as critical sections). Suppose that a rollback recovery approach is used so that a failure has been detected (the failure could be corrupted data on the file) and the application has been restored to a previous saved state. How should the processes be re-executed so as to avoid a recurrence of the failure?

One method, which we call "simple re-execution", is to simply restart all the processes and allow them to execute freely [HK93, WHF+97]. This method relies on chance to change the relative timing between the processes in such a way that races do not recur in the re-execution. Another method, "locked re-execution", involves locking all the file accesses during re-execution. This method eliminates races but introduces the possibility of deadlocks. Again, it depends on chance to re-execute safely. The "controlled re-execution" method uses traced information from the previous execution as input to the predicate control algorithm (for mutual exclusion predicates). During re-execution, extra synchronization messages are used to enforce the synchronizations added by the predicate control algorithm. This method does not rely on chance, and can give a guarantee that there are neither races nor deadlocks during re-execution. However, this benefit comes at the cost of tracing the computation during normal execution.

In order to make an evaluation of controlled re-execution, we require a quantitative evaluation of: (1) how much of a disadvantage is posed by the tracing cost, and (2) how much of an advantage is involved in the improved recovery. This is the objective of our experimental study.

Contributions

In our experiments, we used synthetic applications so that we could study controlled re-execution under different parameters. In particular, there were two parameters that are important: the frequency of communication and the frequency of file accesses. Our experiments demonstrate that the tracing overhead is less than 1% even for applications with relatively high communication and file access frequencies (with respect to parameters of real scientific applications). In order to check the relative likelihood of recovery, we executed applications that periodically have races and measured how long it takes before each of the three re-execution methods fails to tolerate races. We varied the applications for various values of communication and file access density. We found that controlled re-execution is significantly better than either locked re-execution or simple re-execution for tolerating races. Thus, since the tracing costs are within tolerable limits and the advantage in recovery is significant, we conclude that the controlled re-execution method is effective and desirable for tolerating races in long-running, non-interactive scientific applications. Further, this case study serves as an indicator for the potential advantages of using controlled re-execution for other types of synchronization faults and in other application domains.

1.4 Summary

- **Predicate Control:** By designing efficient algorithms for solving the predicate control problem for some important classes of predicates, we make a first step in determining the feasibility of off-line synchronization.

- **Predicate Detection:** Towards understanding the implications of the extended model on the predicate detection problem, we study and design algorithms for the detection of conjunctive predicates.

- **Controlled Re-execution:** As a case study in evaluating controlled re-execution, we demonstrate that it is a desirable and effective method for tolerating race faults in long-running, non-interactive distributed applications.

1.5 Overview of the Dissertation

The remainder of this dissertation is organized as follows. In Chapter 2, we define our model. Next, we have the three main chapters of the dissertation: Chapter 3 discusses our study of the predicate control problem, Chapter 4 investigates the predicate detection problem, and Chapter 5 describes our experimental evaluation of the controlled re-execution method. In Chapter 6, we give a summary of related work. Finally, we draw conclusions and describe future directions in Chapter 7.

Chapter 2

Model

The predicate control problem and the predicate detection problem are based on a model of distributed computations and concepts related to distributed computations. Our model is similar to the *happened before* model [Lam78], described in more detail in [BM93, Gar96] and extended in [ACG93]. However, some of the concepts presented are unique to our requirements, and we will make special note of this where appropriate.

2.1 Computations

Since we are going to be concerned mainly with distributed computations, we drop the qualifier *distributed* and call them simply *computations*.

- $\langle$**computation**$\rangle$ **:** A *computation* is a tuple $\langle E_1, E_2, \cdots E_n, \rightarrow \rangle$ where the E_i's are disjoint finite sets of "events" and $\rightarrow$ (precedes) is an irreflexive partial order on $E = \bigcup_i E_i$. We abbreviate $\langle E_1, E_2, \cdots E_n, \rightarrow \rangle$ to $\langle E, \rightarrow \rangle$ with the understanding that n is to be always used to represent the size of the partition of E into $E_1, E_2, \cdots E_n$.

Informally, each E_i represents a *process*, and the partial order $\rightarrow$ represents a partial ordering of events such that $e \rightarrow f$ means that the event e *may* have directly or indirectly caused the event f. We say e "causally precedes" f. Note that, in general, the $\rightarrow$ relation is merely an approximation of causality. An example of approximating causality to obtain a $\rightarrow$ relation on events is given by the *happened before* model [Lam78], in which $\rightarrow$ is defined as the smallest relation satisfying the following: (1) If e and f are events in the same process and e comes before f according to the local process clock, then $e \rightarrow f$, (2) If e is the send event of a message and f is the receive event for the same message by another process, then $e \rightarrow f$, and (3) If $e \rightarrow f$ and $f \rightarrow g$, then $e \rightarrow g$. Note that this approximation causes the events in a process to be totally ordered. We define such computations as follows:

- $\langle \rightarrow_i \rangle$ **:** Let $\rightarrow_i$ denote the relation $\rightarrow$ restricted to the set of events E_i.
- $\langle$**locally ordered computations**$\rangle$ **:** A computation $\langle E, \rightarrow \rangle$ is a *locally ordered* computation if, for each i, $\rightarrow_i$ is a total order.

For now, we assume that all computations are locally ordered. In Chapter 4, we will return to general computations and motivate the distinctions between the models. Another special class of computations are runs, defined as:

- $\langle$**run**$\rangle$ **:** A computation $\langle E, \rightarrow \rangle$ is a *run* if $\rightarrow$ is a total order.

A run is an even weaker approximation of causality than locally ordered computations. A run corresponds to an interleaving of all the events in the distributed computation. Finally, we define some notation concerning computations:

- $\langle \underline{\rightarrow} \rangle$ **:** $(e \underline{\rightarrow} f) \;\equiv\; (e = f) \;\vee\; (e \rightarrow f)$

  In a similar way, we represent a corresponding reflexive relation for any given relation, e.g.: $(e \preceq f) \;\equiv\; (e = f) \;\vee\; (e \prec f)$
- $\langle e.proc \rangle$ **:** $(e.proc = i) \;\equiv\; (e \in E_i)$

Informally, $e.proc$ is the identifier for the process containing e.

2.2 Cuts

- $\langle$**cut**$\rangle$ **:** A *cut* is a subset of E containing at most one event from each E_i.

A cut corresponds to the intuitive notion of a global state. Sometimes a cut has been defined to correspond to the notion of a history (all events preceding a global state). In such definitions, the *frontier of a cut* corresponds to our *cut*. Since, we will be dealing more frequently with the notion of a global state than a history, we formalize the notion of history in terms of global state rather than the other way around.

If a cut intersects a process at an event e, then the local state on that process corresponding to the cut is the one reached by executing event e. If the cut does not intersect a process, then the local state on that process corresponding to the cut is the initial state on that process. To represent an initial state explicitly, we augment our model with initialization events:

- $\langle \bot_i \rangle$ **:** Corresponding to each E_i we define a special event $\bot_i$ ($\bot_i \notin E$).

- $\langle \bot \rangle$ **:** Let $\bot = \bigcup_i \{\bot_i\}$.

Each $\bot_i$ corresponds to a special "dummy" event that initializes the state of process E_i. Note that $\bot_i$ is not a real event. Introducing the dummy events, $\bot$, allows a one-to-one correspondence between the informal notion of local states corresponding to a process E_i and the formal notion of $E_i \cup \{\bot_i\}$, in which an event corresponds to the local state that follows it. We next define a local ordering between events and dummy events:

- $\langle \prec_i \rangle$ **:** For each i, let $\prec_i$ be the smallest relation on $E_i \cup \{\bot_i\}$ such that:

  $\forall e \in E_i : \ \bot_i \prec_i e$, and

  $\forall e, f \in E_i : \ e \rightarrow_i f \Rightarrow e \prec_i f$.

- $\langle \prec \rangle$ **:** Let $\prec = \bigcup_i \ \prec_i$.

Thus, the $\prec$ relation augments the $\rightarrow_i$ relation by ordering the initialization event $\perp_i$ before all events in E_i. Using the initialization events $\perp$, we can now define, for any cut, a unique event/initialization event corresponding to each process as:

- $\langle C[i] \rangle$ **:** For a cut C, define $C[i] \in E \cup \perp$ such that:

  $C[i] = e$, if $C \cap E_i = \{e\}$, and

  $C[i] = \perp_i$, if $C \cap E_i = \emptyset$.

Finally, we define a relation $\leq$ on cuts as follows:

- $\langle \leq \rangle$ **:** For two cuts C_1 and C_2 define a relation $\leq$ as:

  $C_1 \leq C_2 \quad \equiv \quad \forall i : C_1[i] \preceq C_2[i]$

2.3 Predicates

Predicates represent the informal notion of properties on local or global states:

- $\langle$**local predicate**$\rangle$ **:** A *local predicate* of an E_i in a computation is a (polynomial-time computable) boolean function on $E_i \cup \{\perp_i\}$.

- $\langle$**global predicate**$\rangle$ **:** A *global predicate* of a computation $\langle E, \rightarrow \rangle$ is a (polynomial-time computable) boolean function on the set of cuts of the computation.

Local predicates are used to define properties local to a process, such as: whether the process is in a critical section, whether the process has a token, or whether, for a process variable x, $x < 4$. Global predicates define properties across multiple processes, such as: whether two processes are in critical sections, whether at least one process has a token, or whether, for variables x and y on different processes, $(x < 4) \wedge (y < 3)$. We use *predicate* instead of global predicate, when the meaning is clear from the context.

2.4 Consistent Cuts

All cuts in a computation do not correspond to global states that may have happened in the computation. To see why this is true, consider the set of events which causally precede any event in a given cut, which we call the *causal past* of the cut:

- $\langle G.past \rangle$ **:** The *causal past* of a set of events $G \subseteq E$ in a computation $\langle E, \rightarrow \rangle$ is $G.past = \{e \in E : \exists f \in G : e \underline{\rightarrow} f\}$

Suppose there is an event e in the causal past of a cut C and suppose e has not yet occurred in the global state corresponding to C. Since a cause must always precede the effect, the cut C does not represent a global state that may have happened. Cuts which represent global states that may have happened are called *consistent* global states. The reason we say that a consistent global state *may* have happened is that its occurrence depends on the particular interleaving of the partially ordered set of events that occurred in real time.

- $\langle$**consistent cut**$\rangle$ **:** A cut C is *consistent* if $\forall e \in C.past : C[e.proc] \not\prec e$.

Note that, in the case of a locally ordered computation, $C[e.proc] \not\prec e$ is the same as $e \preceq C[e.proc]$.

We now prove two useful results about consistent cuts. The first one allows us to extend a single event to a consistent cut that contains it. This corresponds to the intuition that the local state following the event must have occurred at some time. Therefore, there must be some consistent cut (global state that may have happened) that contains it.

Lemma 1 *(Extensibility) Let $\mathcal{C}$ be the set of consistent cuts of a computation $\langle E, \rightarrow \rangle$, then:*

$$\forall e \in E : \exists C \in \mathcal{C} : e \in C$$

Proof: Let e be an event in E. Consider a maximal (w.r.t. $\leq$) cut C contained in $\{e\}.past$. Since C is maximal, it must contain e. Consider any event f in $C.past$.

Since C is contained in $\{e\}.past$, f is also in $\{e\}.past$. Suppose $C[f.proc] \prec f$. Then since f is in $\{e\}.past$, C is not maximal and we have a contradiction. Therefore, $C[f.proc] \not\prec f$. So, C is consistent. $\Box$

The next result tells us that a cut that is consistent in a computation is also consistent in any computation that is less strict (that is, the partial ordering of events in the latter is less strict than the first).

Lemma 2 *If $\langle E, \rightarrow' \rangle$ and $\langle E, \rightarrow \rangle$ are computations such that $\rightarrow \subseteq \rightarrow'$, then any cut that is consistent in $\langle E, \rightarrow' \rangle$ is also consistent in $\langle E, \rightarrow \rangle$.*

Proof: Let C be a consistent cut in $\langle E, \rightarrow' \rangle$. Let e be any event in the causal past of C with respect to $\rightarrow$. Since $\rightarrow \subseteq \rightarrow'$, e is also in the causal past of C with respect to $\rightarrow'$. Therefore, $C[e.proc] \not\prec e$. So, C is consistent in $\langle E, \rightarrow \rangle$. $\Box$

2.5 More on Consistent Cuts

This sections presents definitions and results related to consistent cuts that only apply to locally ordered computations. In case of locally ordered computations, every event has a unique next event and a unique previous event:

- $\langle e.next \rangle$ **:** For $e \in E_i \cup \{\perp_i\}$, $e.next$ denotes the immediate successor to e in the total order defined by the $\prec_i$ relation. If no successor exists, $e.next = null$, where $null$ does not belong to $E \cup \perp$.

- $\langle e.prev \rangle$ **:** For $e \in E_i \cup \{\perp_i\}$, $e.prev$ denotes the immediate predecessor to e in the total order defined by the $\prec_i$ relation. If no predecessor exists, $e.prev = null$, where $null$ does not belong to $E \cup \perp$.

Note that $null$ (not an event) should not be confused with $\perp_i$ (the initialization event). In fact, $\perp_i.prev = null$.

The following result shows us an alternative way of proving that a cut is consistent.

Lemma 3 *In a computation* $\langle E, \rightarrow \rangle$*:*

$$\text{a cut } C \text{ is consistent} \equiv \quad \forall i, j : C[i].next, C[j] \in E : C[i].next \not\rightarrow C[j]$$

Proof: (1) Suppose C is consistent. We show by contradiction that:

$$\forall i, j : C[i].next, C[j] \in E : C[i].next \not\rightarrow C[j].$$

Suppose there is an i and j such that $C[i].next \rightarrow C[j]$. Then $C[i].next \in C.past$. But $C[i] \prec C[i].next$ contradicting the consistency of C.

(2) Suppose C is not consistent. We prove by contradiction that:

$$\exists i, j : C[i].next, C[j] \in E : C[i].next \rightarrow C[j].$$

Since C is not consistent, there is an event, say e, such that $e \in C.past$ and $C[e.proc] \prec e$. Therefore, $C[e.proc].next \preceq e$. By transitivity, $C[e.proc].next \in C.past$. Therefore, $\exists j : C[j] \in E : C[e.proc].next \rightarrow C[j]$. $\Box$

The next is a well-known result about the set of consistent cuts in a computation [Mat89].

Lemma 4 *The set of consistent cuts of a computation forms a lattice under* $\leq$*.*

Proof: Let C_1 and C_2 be any two consistent cuts. Let C be a consistent cut defined such that for each i, $C[i]$ is the minimum of $C_1[i]$ and $C_2[i]$. Clearly, this ensures that for any lower bound C' of C_1 and C_2, $C' \leq C$. Therefore, to show that C is the greatest lower bound of C_1 and C_2, it remains to show that C is consistent.

Suppose C is not consistent. Using Lemma 3, there must exist i and j such that $C[i].next \rightarrow C[j]$. Without loss of generality, suppose $C[i] = C_1[i]$. Clearly, $C[j] \neq C_1[j]$ since, otherwise, the consistency of C_1 is contradicted. Therefore, $C[j] = C_2[j]$. By the definition of C, $C_2[j] \preceq C_1[j]$. Therefore, by transitivity, $C_1[i].next \rightarrow C_1[j]$. This contradicts the consistency of C_1. This shows that C is consistent and, therefore, C is the greatest lower bound of C_1 and C_2.

In a similar way, we can demonstrate the existence of the least upper bound of C_1 and C_2. $\Box$

It can further be verified that the lattice is a distributive one. This result allows us to define the consistent cut that is the greatest lower bound or least upper bound of a non-empty set of consistent cuts. Consider the set A of consistent cuts containing an event e. Lemma 1 tells us that A is non-empty. By Lemma 4, there is a consistent cut that is the greatest lower bound of A. Further, it is easy to check that this consistent cut contains e. Therefore, we can define the following:

- **$\langle lcc(e) \rangle$:** For each event e in a computation, let $lcc(e)$ denote the *least consistent cut* that contains e.

Informally, $lcc(e)$ represents the earliest global state for which e has occurred. It is verifiable that $lcc(e)$ is the maximum cut contained in $\{e\}.past$ (similar to the proof of Lemma 1). Therefore, we have:

Lemma 5 *$lcc(e).past = \{e\}.past$.*

This result shows us that $lcc(e)$, which corresponds to the earliest global state for which e has occurred, is formed by the "frontier" of the causal past of e (that is, the set of last events per process in the causal past of e).

There can be only one global state that occurs immediately before $lcc(e)$ and it is represented by the following cut:

- **$\langle lccprev(e) \rangle$:** For each event e in a computation, let $lcc\text{-}prev(e)$ denote the cut formed from $lcc(e)$ by deleting e, and then by adding $e.prev$ if $e.prev \notin \bot$.

To show that $lcc\text{-}prev(e)$ is indeed the global state that occurs immediately before $lcc(e)$, we must show that it is consistent.

Lemma 6 *For any event e in a computation, $lcc\text{-}prev(e)$ is consistent.*

Proof: We prove by contradiction. Let f be any event in $lcc\text{-}prev(e).past$ such that $lcc\text{-}prev(e)[f.proc] \prec f$. Clearly, by the definition of $lcc\text{-}prev(e)$ we have $f.proc \neq$

e.proc. Therefore, *lcc-prev*$(e)[f.proc] = lcc(e)[f.proc]$ (by the defn. of *lcc-prev*(e)). Further, since f is in *lcc-prev*$(e).past$, it is also in $lcc(e).past$. This contradicts the consistency of $lcc(e)$. $\Box$

2.6 Runs

In this section we state some definitions and results specific to runs. In the case of a run, we can make a stronger assertion about the set of consistent cuts than the fact that it forms a lattice (Lemma 4):

Lemma 7 *The set of consistent cuts of a run is totally ordered under* $\leq$.

Proof: We prove by contradiction. Suppose that C_1 and C_2 are two consistent cuts in a computation $\langle E, \rightarrow\rangle$ such that $C_1 \not\leq C_2$ and $C_2 \not\leq C_1$. Therefore, there must exist i and j such that $C_1[i] \prec C_2[i]$ and $C_2[j] \prec C_1[j]$. Since $\rightarrow$ is a total order, without loss of generality, let $C_2[i] \rightarrow C_1[j]$. Therefore, $C_2[i]$ is in $C_1.past$ contradicting the consistency of C_1. $\Box$

This corresponds to the intuition that since the events in a run are totally ordered, the global states must also occur in sequence. Further, for any global state that occurs, there is a unique event that can lead up to it. This unique event is defined as:

- $\langle ge(C)\rangle$ **:** For each consistent cut C of a run $\langle E, \rightarrow\rangle$, such that $C \neq \emptyset$, let the greatest event in C with respect to $\rightarrow$ be denoted by $ge(C)$.

The next result shows us that there is a one-to-one correspondence between the consistent cuts (excluding $\emptyset$) and the events in a run.

Lemma 8 *In a given run, ge and lcc are inverses of each other.*

Proof: Consider a computation, $\langle E, \rightarrow\rangle$. It follows directly from Lemma 5 that $\forall e \in E : ge(lcc(e)) = e$. Therefore, it remains to prove that $lcc(ge(C)) = C$, for a consistent cut $C \neq \emptyset$. We prove by contradiction. Suppose $lcc(ge(C)) \neq C$, then there is an i such that $lcc(ge(C))[i] \neq C[i]$. By the definition of lcc, we must have: $lcc(ge(C))[i] \prec C[i]$. By the definition of ge, $C[i] \rightarrow ge(C)$ and so, $C[i] \in lcc(ge(C)).past$. This contradicts the consistency of $lcc(ge(C))$. $\Box$

2.7 Interval Graphs

Let $\langle E, \rightarrow \rangle$ be a locally ordered computation and let $\alpha_1, \alpha_2, \cdots \alpha_n$ be a set of local predicates. Each local predicate α_i defines a partition of the sequence of events $\langle E_i \cup \{\perp_i\}, \prec_i \rangle$ into "intervals" in which α_i is alternately true and false. We define this formally as:

- $\langle$**interval**$\rangle$ **:** An interval I is a non-empty subset of an $E_i \cup \{\perp_i\}$ corresponding to a maximal subsequence in the sequence of events in $\langle E_i \cup \{\perp_i\}, \prec_i \rangle$, such that all events in I have the same value for α_i.

We next introduce a few notations and definitions related to intervals.

- $\langle \mathcal{I}_i \rangle$ **:** Let $\mathcal{I}_i$ denote the set of intervals of $E_i \cup \{\perp_i\}$.
- $\langle \mathcal{I} \rangle$ **:** Let $\mathcal{I} = \bigcup_i \mathcal{I}_i$.
- $\langle I.proc \rangle$ **:** $(I.proc = i) \;\equiv\; (I \subseteq (E_i \cup \{\perp_i\}))$
- $\langle I.first \rangle$ **:** For an interval I, $I.first$ denotes the minimum event in I with respect to $\prec_{I.proc}$.
- $\langle I.last \rangle$ **:** For an interval I, $I.last$ denotes the maximum event in I with respect to $\prec_{I.proc}$.
- $\langle \prec_i$ **(for intervals)**$\rangle$ **:** For any i, let the $\prec_i$ relation on events apply to intervals as well, such that $I_1 \prec_i I_2 \;\equiv\; I_1.first \prec_i I_2.first$.

- $\langle I.next \rangle$ **:** $I.next$ denotes the immediate successor interval of interval I with respect to $\prec_{I.proc}$, or $null$ if none exists ($null$ is distinct from all intervals).

- $\langle I.prev \rangle$ **:** $I.prev$ denotes the immediate predecessor interval of interval I with respect to $\prec_{I.proc}$, or $null$ if none exists ($null$ is distinct from all intervals).

- $\langle$**local predicate (on intervals)**$\rangle$ **:** A local predicate α_i also applies to an interval $I \subseteq E_i \cup \{\perp_i\}$ such that $\alpha_i(I) \equiv \alpha_i(I.first)$.

- $\langle$**true interval**$\rangle$ **:** An interval I is called a *true interval* if $\alpha_{I.proc}(I)$.

- $\langle$**false interval**$\rangle$ **:** An interval I is called a *false interval* if $\neg\alpha_{I.proc}(I)$.

The following relation on the set of intervals, $\mathcal{I}$, is defined so that $I_1 \mapsto I_2$ represents the intuitive notion that: "I_1 must enter before I_2 can leave".

- $\langle \mapsto \rangle$ **:** $\mapsto$ is a relation on intervals defined as :

$$I_1 \mapsto I_2 \equiv \begin{cases} I_1.first \rightarrow I_2.next.first & \text{if } I_1.prev \neq null \text{ and } I_2.next \neq null \\ true & \text{if } I_1.prev = null \text{ or } I_2.next = null \end{cases}$$

Note that, the conditions $I_1.prev \neq null$ and $I_2.next \neq null$ imply that $I_1.first \notin \perp$ (so $\rightarrow$ is defined) and $I_2.next.first$ is defined. If $I_1.prev = null$ or $I_2.next = null$, then we define $I_1 \mapsto I_2$ to be true. Since the execution starts in the first interval of every process and ends in the last interval of every process, this corresponds to the intuition that I_1 must enter before I_2 can leave. Further, note that the relation $\mapsto$ is reflexive, corresponding to the intuition that an interval must enter before it can leave. Unlike many of the relations we deal with, it is to be noted that the relation $\mapsto$ is not a partial order and may have cycles.

The set of intervals $\mathcal{I}$ together with the relation $\mapsto$ forms a graph, called an interval graph.

- **⟨interval graph⟩ :** $\langle \mathcal{I}, \mapsto \rangle$ is called the *interval graph* of computation $\langle E, \rightarrow \rangle$ under the set of local predicates $\alpha_1, \alpha_2, \cdots \alpha_n$.

The interval graph represents a higher granularity view of a computation, in which the intervals may be viewed as "large events". However, there is one important difference — an interval graph may be cyclic, while a computation is partially ordered.

Chapter 3

The Predicate Control Problem

In this chapter we describe our results in studying the predicate control problem and design algorithms, some of which will be used in the experimental study of controlled re-execution in Chapter 5.

3.1 Overview

We first define the predicate control problem formally in Section 3.2. Informally, the predicate control problem is to determine how to add edges to a computation (that is, make the computation stricter) so that it maintains a global predicate.

In its full generality, the predicate control problem deals with *any* predicates and it would, therefore, be expected that it is hard to solve. We establish in Section 3.3 that it is NP-Complete. However, as we will see, there are useful predicates for which the problem can be solved efficiently.

The first class of predicates that we study in Section 3.4 is the class of "disjunctive predicates" that are specified as a disjunction of local predicates. Intuitively, these predicates can be used to express a global condition in which at least one local condition has occurred, or, in other words, in which a bad combination of local conditions has not occurred. For example: "at least one server is available". First, we show that a necessary condition to solve the problem for disjunctive pred-

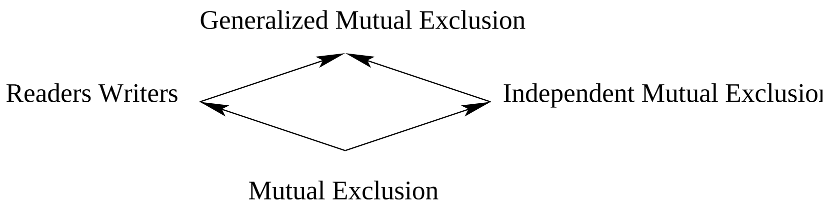

Figure 3.1: Variants of Mutual Exclusion Predicates

icates is the absence of a clique of n false intervals in the interval graph. Next, we design an algorithm for solving the problem when the necessary condition is satisfied, thus establishing that the condition is also sufficient. The algorithm is of complexity $O(np)$, where n is the number of processes and p is the number of false intervals in the interval graph.

The next class of predicates is that of "mutual exclusion predicates" which state that no two processes are in critical sections at the same time. Mutual exclusion predicates are particularly useful in software fault tolerance since they correspond to races, a very common type of synchronization faults. In Section 3.5, our two results show that the necessary and sufficient conditions for solving predicate control for mutual exclusion predicates is the absence of a non-trivial cycle of critical sections in the interval graph. We also design an algorithm of complexity $O(np)$, where p is the number of critical sections in the computation.

We can generalize the mutual exclusion predicates to "readers writers predicates" specifying that only "write" critical sections must be exclusive, while "read" critical sections need not be exclusive. Another generalization is the "independent mutual exclusion predicates" where critical sections have "types" associated with them, such that no two critical sections of the same type can execute simultaneously. Finally, "generalized mutual exclusion predicates" allow read and write critical sections *and* multiple types of critical sections. Figure 3.1 illustrates the relative generality of the four problems.

In Section 3.6, we find that the necessary and sufficient conditions for solving predicate control for readers writers predicates is the absence of a non-trivial cycle of critical sections with at least one write critical section. For independent mutual exclusion, however, we discover that the problem is NP-Complete in general. We show this in Section 3.7, and also show that a sufficient condition for solving the problem is the absence of a non-trivial cycle of critical sections with two critical sections of the same type. However, in general, this condition is not necessary. The results for generalized mutual exclusion follow along similar lines. We do not describe individual algorithms for readers writers and independent mutual exclusion predicates. Instead, in Section 3.8, we describe a general $O(np)$ algorithm that solves predicate control for general mutual exclusion predicates (under the sufficient conditions). This algorithm can be applied to solving readers writers predicates (in general) and independent mutual exclusion predicates (under the sufficient conditions).

3.2 Problem Statement

In order to state the problem, we first require the concept of a "controlling computation". Informally, given a predicate and a computation, a controlling computation is a stricter computation for which all consistent cuts satisfy the predicate. Formally, it is defined as:

- $\langle$**controlling computation**$\rangle$ **:** Given a computation $\langle E, \rightarrow \rangle$ and a global predicate ϕ, a computation $\langle E, \rightarrow^c \rangle$ is called a *controlling computation* of ϕ in $\langle E, \rightarrow \rangle$, if: (1) $\rightarrow \subseteq \rightarrow^c$, and
  (2) for all consistent cuts C in $\langle E, \rightarrow^c \rangle$: $\phi(C)$

Given this definition, the predicate control problem is:

The Predicate Control Problem: Given a computation $\langle E, \rightarrow \rangle$ and a global predicate ϕ, is there a controlling computation of ϕ in $\langle E, \rightarrow \rangle$?

The search problem corresponding to the predicate control problem is to find such a controlling computation, if one exists.

3.3 Predicate Control is NP-Complete

We prove that predicate control is NP-Complete in two steps. We first define the predicate control problem for runs and show that it is equivalent to the predicate control problem. We next show that the predicate control problem for runs is NP-Complete.

The Predicate Control Problem for Runs: Given a computation $\langle E, \rightarrow \rangle$ and a global predicate ϕ, is there a controlling *run* of ϕ in $\langle E, \rightarrow \rangle$?

Lemma 9 *The Predicate Control Problem is equivalent to the Predicate Control Problem for Runs.*

Proof: Let $\langle E, \rightarrow \rangle$ be a computation and ϕ be a global predicate. We have to show that a controlling computation of ϕ in $\langle E, \rightarrow \rangle$ exists if and only if a controlling run of ϕ in $\langle E, \rightarrow \rangle$ exists.

Since a run is a special case of a computation, the "if" proposition follows.

For the "only if" proposition, suppose a controlling computation $\langle E, \rightarrow^c \rangle$ of ϕ in $\langle E, \rightarrow \rangle$ exists. Let $<^c$ be any linearization of $\rightarrow^c$ (a *linearization* of a partial order is a totally ordered superset). If C is a consistent cut of $\langle E, <^c \rangle$, then C is also a consistent cut of $\langle E, \rightarrow^c \rangle$ (Lemma 2). Since, $\langle E, \rightarrow^c \rangle$ is a controlling computation, we have $\phi(C)$. Therefore, $\langle E, <^c \rangle$ is also a controlling computation of ϕ in $\langle E, \rightarrow \rangle$. $\square$

Lemma 10 *The Predicate Control Problem for Runs is NP-Complete.*

Proof: First, we show that the problem is in NP. A non-deterministic algorithm need only guess a run $\langle E, <^c \rangle$ such that $<^c$ is a linearization of $\rightarrow$. To check that a run is a controlling computation, we first enumerate all the consistent cuts in the run. By Lemma 8, each consistent cut is the *lcc* of some event, and, therefore, we can achieve the enumeration by finding the *lcc* of each event in the run. Using Lemma 5 and the fact that the causal past of a cut is unique, it can be easily shown that the *lcc* of an event can be constructed in polynomial time. Further, by definition, it can be checked in polynomial time that the given predicate satisfies each *lcc*.

To show that the problem is NP-Hard, we transform SAT. Let $U = \{u_1, \cdots u_m\}$ be a set of boolean variables, and $D = \{d_1, \cdots d_k\}$ be a set of clauses forming an instance of SAT. Let ψ be the boolean function over the set of truth assignments such that $\psi(t) \equiv t$ satisfies D. (ψ can be represented as a boolean formula in conjunctive normal form).

For each variable $u_i \in U$, construct a singleton set of events $E_i = \{e_i\}$. Construct one more set $E_{m+1} = \{e_{m+1}, e'_{m+1}\}$. Let $E = \bigcup_{i \in \{1, \cdots m+1\}} E_i$. Let relation $\rightarrow$ be defined so that $e_{m+1} \rightarrow e'_{m+1}$ and no other events are related. Clearly $\rightarrow$ is a partial order on E and $\rightarrow_i$ is a total order. Therefore, $\langle E, \rightarrow \rangle$ is a (locally ordered) computation.

For each i, let α_i be a local predicate on $E_i \cup \{\perp_i\}$ as follows. Let $\alpha_{m+1}(\perp_{m+1}) = \mathit{true}$, $\alpha_{m+1}(e_{m+1}) = \mathit{false}$, and $\alpha_{m+1}(e'_{m+1}) = \mathit{true}$. For each $i \leq m$ let $\alpha_i(\perp_i) = \mathit{false}$ and $\alpha_i(e_i) = \mathit{true}$. Note that for each cut C, there is a truth assignment t_C such that $t_C(u_i) \equiv \alpha_i(C[i])$. Define a global predicate $\phi = \alpha_{m+1}(C[m+1]) \vee \psi(t_C)$.

Thus, we have a polynomial construction from an instance (U, D) of SAT to an instance $(\langle E, \rightarrow \rangle, \phi)$ of the Predicate Control Problem for Runs. We next show that this is indeed a transformation.

Part 1: Suppose that $\langle E, <^c \rangle$ is a controlling run of ϕ in $\langle E, \rightarrow \rangle$. Let C be any consistent cut containing e (by Lemma 1). Since $\phi(C) = \mathit{true}$ and $\alpha_{m+1}(C[m+1]) = \mathit{false}$, we must have $\psi(t_C)$. Therefore, t_C is a truth assignment that satisfies the set of clauses D.

Part 2: Suppose that t is a truth assignment of variables in U that satisfies D.

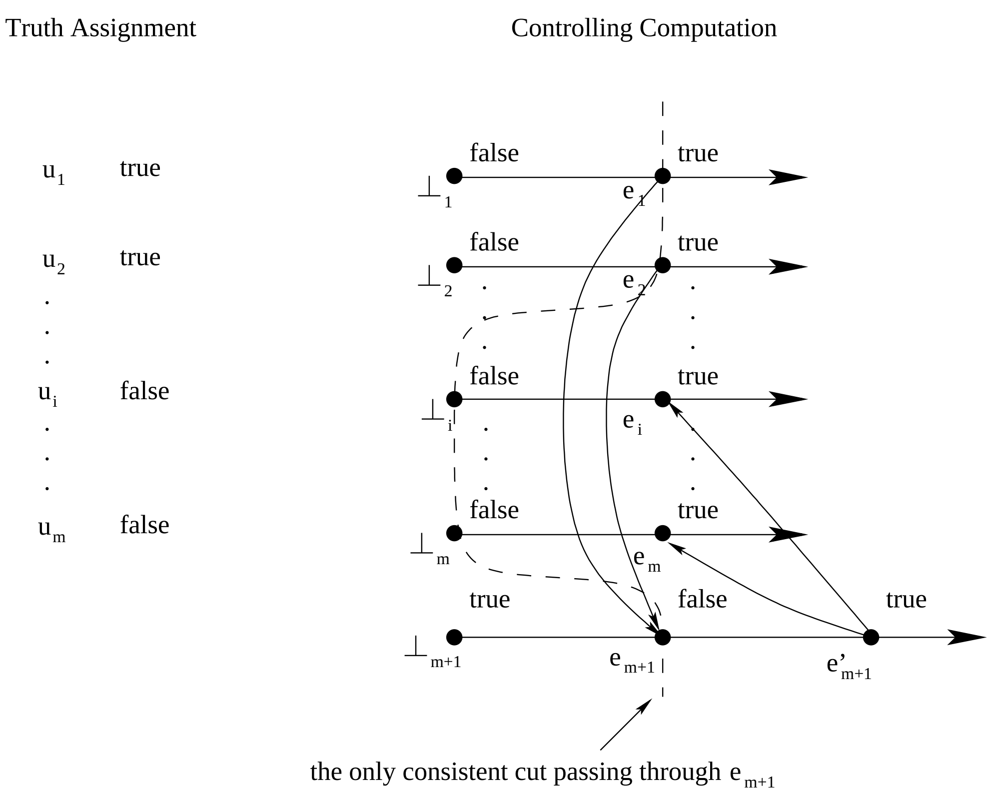

Figure 3.2: Proof: The Predicate Control Problem for Runs is NP-Complete

Define $\rightsquigarrow^c$ such that $\rightarrow \subset \rightsquigarrow^c$ and $\rightsquigarrow^c$ contains the following edges in addition to edges in $\rightarrow$: (1) for each $i \leq m$ such that $t(u_i) = \mathit{true}$, the edge $e_i \rightsquigarrow^c e_{m+1}$, and (2) for each $i \leq m$ such that $t(u_i) = \mathit{false}$, the edge $e'_{m+1} \rightsquigarrow^c e_i$. Let $\rightarrow^c$ be the transitive closure of $\rightsquigarrow^c$. Since each E_i except E_{m+1} does not have both incoming and outgoing $\rightsquigarrow^c$ edges, there can be no cycles in $\rightsquigarrow^c$, and, therefore, $\rightarrow^c$ is a partial order and $\langle E, \rightarrow^c \rangle$ is a computation. Let $<^c$ be any linearization of $\rightarrow^c$.

Claim: The run $\langle E, <^c \rangle$ is a controlling run of ϕ in $\langle E, \rightarrow \rangle$.

Proof: Let C be a consistent cut of the run $\langle E, <^c \rangle$. We show that $\phi(C) = \mathit{true}$. If $\alpha_{m+1}(C[m+1]) = \mathit{true}$, we directly have $\phi(C) = \mathit{true}$. Therefore, we consider the remaining possibility: $\alpha_{m+1}(C[m+1]) = \mathit{false}$. By the definition of α_{m+1}, we have: $C[m+1] = e_{m+1}$. We next show that $t_C = t$. Consider any $i \leq m$. We consider two cases depending on the value of $t(u_i)$:

Case 1: $t(u_i) = \mathit{true}$:

Therefore, $e_i <^c e_{m+1}$ (by the definition of $\leadsto^c$ and $\leadsto^c \subset <^c$). Since C is consistent and $C[m+1] = e_{m+1}$, we must have $C[i] = e_i$. By the definition of α_i, we have $\alpha_i(C[i]) = \mathit{true}$. Therefore, $t_C(u_i) = \mathit{true}$.

Case 2: $t(u_i) = \mathit{false}$:

Therefore, $e'_{m+1} <^c e_i$ (by the definition of $\leadsto^c$ and $\leadsto^c \subset <^c$). Since C is consistent and $C[m+1] = e_{m+1}$, we must have $C[i] = \bot_i$. By the definition of α_i, we have $\alpha_i(C[i]) = \mathit{false}$ and, therefore, $t_C(u_i) = \mathit{false}$.

This proves that $t_C = t$. Since t satisfies the set of clauses, D, we have $\psi(t_C) = \mathit{true}$. Therefore, $\phi(C) = \mathit{true}$. $\langle$ *End: Proof of Claim* $\rangle$ □

Theorem 1 *The Predicate Control Problem is NP-Complete.*

Proof: Follows directly from Lemmas 9 and 10. □

3.4 Disjunctive Predicates

- $\langle$**disjunctive predicates**$\rangle$ **:** Given n local predicates $\alpha_1, \alpha_2, \cdots, \alpha_n$, the *disjunctive predicate* ϕ_{disj} is defined as:

$$\phi_{disj}(C) \equiv \bigvee_{i \in \{1, \cdots, n\}} \alpha_i(C[i])$$

Some examples of disjunctive predicates are:

(1) At least one server is available:	$avail_1 \vee avail_2 \vee \ldots avail_n$
(2) x must happen before y:	*after* x $\vee$ *before* y
(3) At least one philosopher is thinking:	$think_1 \vee think_2 \vee \ldots think_n$
(4) $(n-1)$-mutual exclusion:	$\neg cs_1 \vee \neg cs_2 \vee \ldots \neg cs_n$

Note how we can even achieve the fine-grained control necessary to cause a specific event to happen before another as in property (3). This was done using local predicates to check if the event has happened yet. $(n-1)$-mutual exclusion is a special case of k-mutual exclusion, which states that at most k processes can be in the critical sections at the same time.

We next determine a necessary condition for solving the predicate control problem for disjunctive predicates.

Theorem 2 *Let $\langle \mathcal{I}, \mapsto \rangle$ be the interval graph of a computation $\langle E, \rightarrow \rangle$ under local predicates $\alpha_1, \alpha_2, \cdots, \alpha_n$. If $\langle \mathcal{I}, \mapsto \rangle$ contains a clique of n false intervals, then there is no controlling computation of ϕ_{disj} in $\langle E, \rightarrow \rangle$.*

Proof: Let $\langle \mathcal{I}, \mapsto \rangle$ have a clique X of n false intervals. Since for any two distinct false intervals I and I' in X, $I \mapsto I'$ and $I' \mapsto I$, we must have $I.proc \neq I'.proc$. Therefore, each of the false intervals in X is on a distinct E_i. Let $I_1, I_2, \cdots I_n$ be the false intervals of X, such that $I_i.proc = i$). We prove by contradiction that there is no controlling computation of ϕ_{disj} in $\langle E, \rightarrow \rangle$.

Suppose that $\langle E, \rightarrow^c \rangle$ is a controlling computation of ϕ_{disj} in $\langle E, \rightarrow \rangle$. Let $<^c$ be any linearization of $\rightarrow$. Therefore, by Lemma 2, $\langle E, <^c \rangle$ is also a controlling computation. We next show that there is a consistent cut C in the run $\langle E, <^c \rangle$ such that $\neg\phi_{disj}(C)$, thus, contradicting the fact that $\langle E, <^c \rangle$ is a controlling computation.

Consider two cases:

Case 1: $\forall i : I_i.next = null$:

Let C be the maximum cut in the run $\langle E, \rightarrow \rangle$. It is easy to verify that C is consistent (as in any computation). Since each I_i is the last interval and is also a false interval, $\neg(\phi_{disj}(C))$.

Case 2: $\exists i : I_i.next \neq null$:

Let e be the minimum event in $\langle E, <^c \rangle$ such that $e = I_i.next.first$ for some i. Let $C' = lcc(e)$. Consider any $j \in \{1, \cdots n\}$ such that $j \neq i$. Consider two cases:

Case a: $I_j.prev = null$: In this case $I_j.first = \perp_j$. Therefore, $I_j.first \preceq_j C'[j]$.

Case b: $I_j.prev \neq null$: Since $I_j \mapsto I_i$, we have $I_j.first \rightarrow e$. Further, since $\rightarrow \subseteq <^c$, we have $I_j.first <^c e$. Therefore, by the consistency of C', we have $I_j.first \preceq_j C'[j]$.

Therefore, in both cases: $I_j.first \preceq_j C'[j]$ –[A]

Consider two cases:

Case a: $I_j.next = null$: Clearly, $C'[j] \preceq_j I_j.last$.

Case b: $I_j.next \neq null$: Since $e <^c I_j.next.first$ (by the definition of e) and since $e = ge(C')$ (by Lemma 8), we have $C'[j] <^c e <^c I_j.next.first$. Therefore, $C'[j] \preceq_j I_j.last$.

Therefore, in both cases: $C'[j] \preceq I_j.last$ –[B]

By [A] and [B], we have $C'[j] \in I_j$ and therefore $\neg\alpha_j(C'[j])$. Let $C = lcc\text{-}prev(e)$. Since, $\forall j : j \neq i : C[j] = C'[j]$, we have: $\neg\alpha_j(C[j])$. Further, since $e = I_i.next.first$, $C[i] = I_i.last$, and therefore, $\neg\alpha_i(C[i])$. So, $\neg\phi_{disj}(C)$. Also, by Lemma 6, C is consistent.

□

Algorithm Description

In order to show that the necessary condition is also sufficient, we design an algorithm that finds a controlling computation whenever the condition is true. The algorithm is shown in Figure 3.3.

The algorithm takes as input the n sequences of intervals for each of the processes. The output contains a sequence of added edges. The central idea is that this sequence of added edges links true intervals into a continuous "chain" from the start of the computation to the end. Any cut must either intersect this chain in a true interval, in which case it satisfies the disjunctive predicate, or in an edge, in which case the cut is made inconsistent by the added edge.

The purpose of each iteration of the main loop in lines L12-L23 is to add one true interval, *anchor*, to the *chain* of true intervals. The intervals $I_1, \cdots, I_n$ form a "frontier" of intervals that start at the beginning of the computation and continue until one of them reaches the end. In each iteration, a *valid_pair*, consisting of a true interval, *anchor*, and a false interval, *crossed*, is selected at line L13, such that *anchor* does not have to leave before *crossed* leaves. If no such *valid_pair*, we can (and will) show that the necessary condition must be true so that an n-sized clique of false intervals exists in the interval graph. Next, the *anchor* is added to the *chain* of true intervals at line L19. Finally, the frontier advances such that *crossed* is crossed and all intervals which must enter as a result of this are entered (lines L20 - L23). After the loop terminates, the output is constructed from the chain of true intervals by connecting them in reverse order.

The time complexity of the algorithm is $O(np)$ where p is the number of false-intervals in the computation The naive implementation of the algorithm would be $O(n^2p)$ because the outer while loop iterates $O(np)$ times and calculating the set $ValidPairs()$ can take $O(n^2)$ time to check every pair of processes. However, an optimized implementation avoids redundant comparisons in computing the set $ValidPairs()$. Since, in this approach, each new false-intervals has to be compared with $n-1$ existing false-intervals, the time complexity is $O(np)$.

Types:

event: ($proc$: int; v: vector clock)

interval: ($proc$: int; α: boolean; $first$: event; $last$: event)

Input:

$\mathcal{I}_1, \mathcal{I}_2, \cdots, \mathcal{I}_n$: list of *interval*, initially non-empty

Output:

$\mathcal{O}$: list of (*event*, *event*), initially *null*

Vars:

$I_1, I_2, \cdots, I_n$: *interval*, initially $\forall i: \; I_i = \mathcal{I}_i.head$

valid_pairs: set of (*interval*, *interval*)

chain: list of *interval*

anchor, *crossed*: *interval*

prev, *curr*: *interval*

Notation:

$$N_i = \begin{cases} \min X \in \mathcal{I}_i : \; I_i \preceq_i X \; \wedge \; X.\alpha = \mathit{false}, & \text{if exists} \\ \mathit{null}, & \text{otherwise} \end{cases}$$

$$\mathit{select}(Z) = \begin{cases} \text{arbitrary element of set } Z, & \text{if } Z \neq \emptyset \\ \mathit{null}, & \text{otherwise} \end{cases}$$

Procedure:

while ($\forall i: \; N_i \neq \mathit{null}$) do
    $\mathit{valid\_pairs} := \{(I_i, N_j) | \; (I_i.\alpha = \mathit{true}) \; \wedge \; (N_i \not\rightarrow N_j)\}$
    if ($\mathit{valid\_pairs} = \emptyset$)
        exit("no controlling computation exists")
    $(\mathit{anchor}, \mathit{crossed}) := \mathit{select}(\mathit{valid\_pairs})$
    if ($\mathit{anchor.prev} = \mathit{null}$) then
        $\mathit{chain} := \mathit{null}$
    $\mathit{chain.add\_head}(\mathit{anchor})$
    for ($i \in \{1, \cdots n\}$: $i \neq \mathit{crossed.proc}$) do
        while ($I_i.next \neq \mathit{null} \; \wedge \; I_i.next \mapsto \mathit{crossed}$)
            $I_i := I_i.next$
    $I_{crossed.proc} := \mathit{crossed.next}$
$\mathit{anchor} := \mathit{select}(\{I_i : \; i \in \{1, \cdots n\} \; \wedge N_i = \mathit{null}\})$
if ($\mathit{anchor.prev} = \mathit{null}$) then
    $\mathit{chain} := \mathit{null}$
$\mathit{chain.add\_head}(\mathit{anchor})$
$\mathit{prev} := \mathit{chain.delete\_head}()$
while ($\mathit{chain} \neq \mathit{null}$) do
    $\mathit{curr} := \mathit{chain.delete\_head}()$
    $\mathcal{O}.\mathit{add\_head}(\; (\mathit{prev.first}, \mathit{curr.next.first}) \;)$
    $\mathit{prev} := \mathit{curr}$

Figure 3.3: Algorithm for Predicate Control of the Disjunctive Predicate

We have assumed, of course, that the algorithm is correct. We now establish this. In doing so, we also prove that the necessary condition – the existence of an n-sized clique of false intervals – is also a sufficient one for solving predicate control for disjunctive predicates.

We first show that the algorithm is well-specified so that it is indeed an algorithm – all the terms used are well-defined and it terminates.

Lemma 11 *The algorithm in Figure 3.3 is well-specified.*

Proof: We first check that all terms in the algorithm are well-defined. The second condition, $I_i.next \mapsto crossed$, in L21 is well-defined assuming that the first condition, $I_i.next \neq null$, is checked prior to it. The only remaining non-trivial check is for the term $curr.next.first$ at line L31. To show that it is well-defined, first note that, for all the intervals *anchor* added to *chain* at line L19, $anchor.next \neq null$ since $anchor.\alpha = true$ (by lines L13 and L16) and $N_{anchor.proc} \neq null$ (by terminating condition L12). Therefore, immediately after line L27, for all intervals X in *chain*, we have: $(X = chain.head) \lor (X.next \neq null)$. Lastly, note that the interval *chain.head* immediately after line L27 is never assigned to *curr* (by L28).

We next check termination. The while loop in lines L21-L22 terminates since I_i advances once per iteration. The while loop in lines L12-L23 terminates since $I_{crossed.proc}$ advances by at least one interval in each iteration. The loop in lines L29-L32 terminates since *chain* reduces by one interval in each iteration. □

Next, we prove two useful invariants maintained by the algorithm.

Lemma 12 *The algorithm in Figure 3.3 maintains the following invariants on* $I_1, I_2 \cdots, I_n$ *(except in lines L20-L23 while* I_i*'s are being updated):*

INV1: $\forall i, j$: *if* $I_i.next \neq null$ *and* $I_j.prev \neq null$ *then* $I_i.next \not\mapsto I_j.prev$

INV2: $\forall i$: *if* $I_i.prev \neq null$ *then* $I_i.\alpha = true \;\lor\; (\exists j:\; I_i \mapsto I_j.prev \;\land\; I_j.\alpha = true)$

Proof: (structural induction treating L20-L23 as atomic)

Base Case: Initially, $\forall j : I_j = \mathcal{I}_j.head$ and, therefore, $\forall j : I_j.prev = null$ and INV1 and INV2 follow.

Inductive Case: The only statements that update any I_i are L20-L23. Therefore, we assume that INV1 and INV2 are true before an occurrence of L20-L23. We denote the values of the variables I_i at this point by I'_i. We must prove that INV1 and INV2 are true after the occurrence of L20-L23. We denote the values of the variables I_i at this point by I_i

To prove that INV1 holds, let $I_i.next \neq null$ and $I_j.prev \neq null$ and consider two cases:

Case 1: $I_j \neq I'_j$**:** Therefore, the variable I_j was updated at either L22 or L23.

> **Case a:** I_j **updated at** L23**:** Therefore, $I_j.prev = crossed$. Also, by the terminating condition of while loop L21 and since $I_i.next \neq null$, $I_i.next \not\mapsto crossed$. Therefore, $I_i.next \not\mapsto I_j.prev$.
>
> **Case b:** I_j **updated at** L22**:** Therefore, $I_j \mapsto crossed$. Since $I_j.prev \neq null$ and $crossed.next \neq null$ (since $anchor \not\mapsto crossed$ - line L13, L16), we have: $I_j.first \rightarrow crossed.next.first$. –[A]
>
> Also, by the terminating condition of loop L21 and since $I_i.next \neq null$, $I_i.next \not\mapsto crossed$. –[B]
>
> From [A] and [B], we have $I_i.next.first \not\rightarrow I_j.first$. Therefore, $I_i.next \not\mapsto I_j.prev$.

Case 2: $I_j = I'_j$**:** We know that $I'_i.next \neq null$ since $I_i.next \neq null$ and $I'_i \preceq I_i$ (since variable I_i is always advanced). We also know that $I'_j.prev \neq null$ since $I'_j = I_j$. Therefore, by the inductive hypothesis, we have: $I'_i.next \not\mapsto I'_j.prev$. Since $I_j = I'_j$, we further have: $I'_i.next \not\mapsto I_j.prev$. Therefore, $I'_i.next.first \not\rightarrow I_j.first$. Since $I'_i \preceq I_i$, we have $I'_i.next.first \underline{\rightarrow} I_i.next.first$. Therefore, by transitivity, $I_i.next.first \not\rightarrow I_j.first$. It follows that $I_i.next \not\mapsto I_j.prev$.

We next prove that INV2 holds after lines L20-L23. Let $I_i.prev \neq null$. We have two cases:

Case 1: $I_i = I_i'$: By the inductive hypothesis, let j be such that $I_i \mapsto I_j'.prev \wedge I_j'.\alpha = true$. If $I_j = I_j'$, then INV2 clearly holds. Suppose $I_j \neq I_j'$. Therefore, either I_j was updated at L22 or at L23.

Case a: I_j updated at L23: Therefore, $I_j = crossed.next$. Since $crossed.\alpha = false$, we have $I_j.\alpha = true$. Further, since $I_i \mapsto I_j'.prev$ and $I_j' \preceq_j crossed$, we have: $I_i \mapsto I_j.prev$. Therefore, INV2 holds.

Case b: I_j updated at L22: From the terminating condition of the loop at L21, we have:

$I_j'.next \mapsto crossed$ and so, $I_j'.next.first \rightarrow crossed.next.first$. –[A]

Since $I_i \mapsto I_j'.prev$ and $I_i.prev \neq null$, we have:

$I_i.first \rightarrow I_j'.first$. –[B]

Using [A] and [B] and transitivity, we have:

$I_i.first \rightarrow crossed.next.first$ and so, $I_i \mapsto crossed$. Further, $crossed.next.\alpha = true$ and so INV2 follows (instantiating $crossed.proc$ for j).

Case 2: $I_i \neq I_i'$: In this case, either I_i was updated in L23 or L22. If at L23, then $I_i.\alpha = true$ and INV2 follows. If at L22, then by the loop condition at L21, we have: $I_i \mapsto crossed$. Therefore, INV2 follows (instantiating $crossed.proc$ for j).

□

We next show that if the algorithm exits abnormally at line L15, failing to produce a controlling computation, then no controlling computation exists for the problem instance.

Lemma 13 *If the algorithm in Figure 3.3 exits at line* L15*, then no controlling computation exists.*

Proof: If $\forall i, j : N_i \mapsto N_j$ then we have a clique of n false intervals and the result follows from Theorem 2. Therefore, let i and j be such that $N_i \not\mapsto N_j$. Since the set $valid\_pairs = \emptyset$ (line L14), we must have $I_i.\alpha = false$ (from line L13). –[A]

Therefore, by the definition of N_i, we have: $I_i = N_i$. So, since $N_i \not\mapsto N_j$, we have: $I_i.prev \neq null$ –[B]

Applying INV2 and using [A] and [B] let k be such that: $I_i \mapsto I_k.prev \wedge I_k.\alpha = true$. Since $N_i \not\mapsto N_j$ and $N_i = I_i$ and $I_i \mapsto I_k.prev$ we have: $N_k \not\mapsto N_j$. This contradicts the fact that $valid\_pairs = \emptyset$ (line L14). $\Box$

Finally, we show that the output does form a controlling computation.

Lemma 14 *If the algorithm in Figure 3.3 terminates normally, then $\langle E, \rightarrow^c \rangle$ is a controlling computation of ϕ_{disj} in $\langle E, \rightarrow \rangle$, where $\rightarrow^c$ is the transitive closure of the union of $\rightarrow$ with the set of edges in $\mathcal{O}$.*

Proof: First, we show that $\langle E, \rightarrow^c \rangle$ is a computation. Just before loop L29-32, we have $\forall I \in chain : I.prev \neq null$ (by lines L17-L18 and and L25-L26). Therefore, in line L31, $prev.first \in E$ (i.e. $prev.first \notin \bot$). Clearly, the same holds for $curr.next.first$. Therefore, $\mathcal{O}$ determines a set of edges on E. Let $\rightsquigarrow^c$ represent this set of edges.

We show by contradiction that $\rightarrow^c$ is an irreflexive partial order. Since $\rightarrow^c$ is transitive by definition, we assume that it is reflexive. Therefore, $\rightarrow \cup \rightsquigarrow^c$ has a cycle, say $\mathcal{C}$. Among the edges in $\mathcal{C}$ let $X.first \rightsquigarrow^c Y.next.first$ be the edge such that X is the latest to be added to *chain*. Let $U.first \rightsquigarrow^c V.next.first$, be the next $\rightsquigarrow^c$ edge in the cycle $\mathcal{C}$ (note that $U = X$ if there is only one $\rightsquigarrow^c$ edge in $\mathcal{C}$). Consider the algorithm step in which X is the *anchor* and added to *chain*. Since Y was the *anchor* immediately before X, and since the *anchor* is not affected by the update to I_i's in lines L20-L23, we have: $I_{Y.proc} = Y$. –[A]

Since U cannot be added after X (by the definition of X), we have: $U \preceq I_{U.proc}$. –[B]

Since $U.first \rightsquigarrow^c V.next.first$ is the next $\rightsquigarrow^c$ edge in the cycle $\mathcal{C}$ following the edge $X.first \rightsquigarrow^c Y.next.first$, we have: $Y.next.first \rightarrow U.first$. (Note: $Y.next.first \neq U.first$ since both Y and U are true intervals). Therefore, using [A] and [B], we have: $I_{Y.proc}.next.first \rightarrow I_{U.proc}.first$ –[C]

Since $\rightsquigarrow^c$ is a set of edges on E, we know that $U.prev \neq null$. Therefore, using [B], we have: $I_{U.proc}.prev \neq null$ Therefore, applying INV1: $I_{Y.proc}.next \not\rightarrow I_{U.proc}.prev$ and, so: $I_{Y.proc}.next.first \not\rightarrow I_{U.proc}.first$, which contradicts [C]. Therefore, $\rightarrow^c$ is an irreflexive partial order and $\langle E, \rightarrow^c \rangle$ is a computation.

It remains to prove that $\langle E, \rightarrow^c \rangle$ is a controlling computation. Let C be a consistent cut in $\langle E, \rightarrow^c \rangle$. Let I be the last interval to be added to *chain* such that: $I.first \preceq C[I.proc]$ (such an I must exist, since the initial conditions together with lines L17-L18 ensure that for first interval X, $X.first \in \perp$). Consider two cases:

Case 1: ***I is the last interval added to* chain:** Since I was added in line L27, $I.next = null$ (by L24). Therefore, $C[I.proc] \preceq I.last$. Together with the definition of I, this implies that $C[I.proc] \in I$. Therefore $\alpha_{I.proc}(C[I.proc])$ and so, $\phi_{disj}(C)$.

Case 2: ***I is not the last interval added to* chain:** Let I' be the next interval to be added to *chain*. Therefore, $I'.first \rightsquigarrow^c I.next.first$ (by L31). –[A]

By the definition of I, we have: $C[I'.proc] \prec I'.first$. Therefore, since C is consistent in $\langle E, \rightarrow^c \rangle$, $I'.first \notin C.past$. In particular, $I'.first \not\rightarrow^c C[I.proc]$. From [A], we must have: $C[I.proc] \prec I.next.first$. Together with the definition of I, we have $C[I.proc] \in I$. Therefore $\alpha_{I.proc}(C[I.proc])$ and so, $\phi_{disj}(C)$.

□

Our final theorem in this section states the sufficient condition for solving disjunctive predicate control as demonstrated by the correctness of our algorithm.

Theorem 3 *Let $\langle \mathcal{I}, \mapsto \rangle$ be the interval graph of a computation $\langle E, \rightarrow \rangle$ under local predicates $\alpha_1, \alpha_2, \cdots, \alpha_n$. If $\langle \mathcal{I}, \mapsto \rangle$ does not contain a clique of n false intervals, then there is a controlling computation of ϕ_{disj} in $\langle E, \rightarrow \rangle$.*

Proof: By Lemmas 11, 13, and 14, the algorithm in Figure 3.3 determines a controlling computation if $\langle \mathcal{I}, \mapsto \rangle$ does not contain a clique of n false intervals. $\Box$

3.5 Mutual Exclusion Predicates

- $\langle$**mutual exclusion predicates**$\rangle$ **:** Let $critical_1, critical_2, \cdots, critical_n$ be n local predicates and let $critical(e) \equiv critical_{e.proc}(e)$. The *mutual exclusion predicate* ϕ_{mutex} is defined as:

$$\phi_{mutex}(C) \;\equiv\; \forall\ distinct\ i, j :\ \neg\ (\ critical(C[i])\ \wedge\ critical(C[j])\ )$$

- $\langle$**critical section**$\rangle$ **:** A *critical section* (*non-critical section*) denotes a true (false) interval in $\langle E, \rightarrow \rangle$ with respect to $critical_1, critical_2, \cdots, critical_n$.

Mutual exclusion is one of the most common forms of synchronization in distributed applications. In particular, the results for mutual exclusion predicates will be used in the controlled re-execution method in Chapter 5.

The next two theorems demonstrate the necessary and sufficient conditions for solving predicate control for the mutual exclusion predicates.

Theorem 4 *Let $\langle \mathcal{I}, \mapsto \rangle$ be the interval graph of a computation $\langle E, \rightarrow \rangle$ under local predicates $critical_1, critical_2, \cdots, critical_n$. If $\langle \mathcal{I}, \mapsto \rangle$ contains a non-trivial cycle of critical sections, then there is no controlling computation of ϕ_{mutex} in $\langle E, \rightarrow \rangle$.*

Proof: We prove the result by contradiction. Let X be a non-trivial cycle of critical sections in $\langle \mathcal{I}, \mapsto \rangle$ and let $\langle E, \rightarrow^c \rangle$ be a controlling computation of ϕ_{mutex} in $\langle E, \rightarrow \rangle$. Let $<^c$ be a linearization of $\rightarrow^c$ forming a controlling run $\langle E, <^c \rangle$ (by Lemma 2).

We next show that there is a consistent cut C in the run $\langle E, <^c \rangle$ such that $\neg\phi_{mutex}(C)$, thus, contradicting the fact that $\langle E, <^c \rangle$ is a controlling computation. Consider two cases:

Case 1: $\forall CS \in X : CS.next = null$ **:**

Let C be the maximum cut in the run $\langle E, \rightarrow \rangle$. It is easy to verify that C is consistent (as in any computation). Since X has at least two critical sections, $\neg\phi_{mutex}(C)$.

Case 2: $\exists CS \in X : CS.next \neq null$ **:**

Let e be the minimum event in $\langle E, <^c \rangle$ such that $e = CS.next.first$ for some $CS \in X$. Let CS' be the critical section preceding CS in the cycle X. Therefore, $CS' \mapsto CS$. Let $C' = lcc(e)$. Consider two cases:

Case a: $CS'.prev = null$:

Since $CS'.first \in \bot$, we have $CS'.first \preceq C'[CS'.proc]$.

Case b: $CS'.prev \neq null$:

Since $CS' \mapsto CS$, we have $CS'.first \rightarrow e$ and so, $CS'.first <^c e$. Therefore, since C' is consistent, $CS'.first \preceq C'[CS'.proc]$

In both cases, $CS'.first \preceq C'[CS'.proc]$. –[A]

Consider two cases:

Case a: $CS'.next = null$:

Clearly, $C'[CS'.proc] \preceq CS'.last$.

Case b: $CS'.next \neq null$:

Since $e <^c CS'.next.first$ (by the definition of e) and since $e = ge(C')$

(by Lemma 8), we have $C'[CS'.proc] <^c e <^c CS'.next.first$. Therefore, $C'[CS'.proc] \preceq CS'.last$.

In both cases, $C'[CS'.proc] \preceq CS'.last$ –[B]

By [A] and [B], we have $C'[CS'.proc] \in CS'$ and therefore $critical(C'[CS'.proc])$. Let $C = lcc\text{-}prev(e)$. Since, $\forall j : j \neq e.proc : C[j] = C'[j]$, we have: $critical(C[CS'.proc])$. Further, since $e = CS.next.first$, $C[CS.proc] = CS.last$, and therefore, $critical(C[CS.proc])$. So, $\neg\phi_{mutex}(C)$. Also, by Lemma 6, C is consistent.

$\Box$

Theorem 5 *Let $\langle \mathcal{I}, \mapsto \rangle$ be the interval graph of a computation $\langle E, \rightarrow \rangle$ under local predicates $critical_1, critical_2, \cdots, critical_n$. If $\langle \mathcal{I}, \mapsto \rangle$ does not contain a non-trivial cycle of critical sections, then there is a controlling computation of ϕ_{mutex} in $\langle E, \rightarrow \rangle$.*

Proof: Since there are no non-trivial cycles of critical sections in $\langle \mathcal{I}, \mapsto \rangle$, we can topologically sort the critical sections to form a sequence: $CS_1, CS_2, \cdots CS_m$ such that $CS_i \mapsto CS_j \;\Rightarrow\; i \leq j$. –[A]

Define a relation $\rightsquigarrow^c$ on events in E as follows. For each $i \in \{1, \cdots, m-1\}$, if $CS_i.next \neq null$ and $CS_{i+1}.prev \neq null$, then $CS_i.next.first \rightsquigarrow^c CS_{i+1}.first$. Further, no other events are related by $\rightsquigarrow^c$.

Define $\rightarrow^c$ to be the transitive closure of $\rightarrow \cup \rightsquigarrow^c$. We first show that $\langle E, \rightarrow^c \rangle$ is a computation and next show that it is a controlling computation.

If $\langle E, \rightarrow^c \rangle$ is not a computation, then $\rightarrow^c$ is not an irreflexive partial order. Since $\rightarrow^c$ is transitive by definition, it must be reflexive. Therefore, $\rightarrow \cup \rightarrow^c$ must have a cycle, say X. Since $\langle E, \rightarrow \rangle$ is a computation, there can be no cycle that involves no $\rightsquigarrow^c$ edges. Therefore, let $CS_i.next.first \rightsquigarrow^c CS_{i+1}.first$ be the $\rightsquigarrow^c$ edge in X such that i has the maximum value. Let $CS_j.next.first \rightsquigarrow^c CS_{j+1}.first$ be the next $\rightsquigarrow^c$ edge in the cycle X (note that it is possible that $i = j$). Therefore, $CS_{i+1}.first \underline{\rightarrow} CS_j.next.first$. Since by definition two critical sections are not contiguous, the equality cannot hold. Therefore, $CS_{i+1}.first \rightarrow CS_j.next.first$ and so, $CS_{i+1} \mapsto CS_j$. By [A], we have: $i + 1 \leq j$ and so $i < j$. This contradicts the choice of i as maximum. Therefore, $\langle E, \rightarrow^c \rangle$ is a computation.

Before showing that $\langle E, \rightarrow^c \rangle$ is a controlling computation, we first prove a claim:

Claim 1: For all critical sections CS_i and CS_j such that $i < j$, if $CS_i.next \neq null$ and $CS_j.prev \neq null$ then $CS_i.next.first \rightarrow^c CS_j.first$.

Proof: (by induction on $j - i$)

Base Case: $j - i = 1$

Directly from the definition of $\rightsquigarrow^c$.

Inductive Case: $j - i > 1$

Consider two critical sections CS_i and CS_j such that $i < j$ and $CS_i.next \neq null$ and $CS_j.prev \neq null$. We have two cases:

Case 1: $CS_{j-1}.prev = null$**:**

Therefore, by definition, $CS_{j-1} \mapsto CS_i$. By [A], we have $j - 1 \leq i$. Together with $i < j$, this implies $j = i+1$, which violates the inductive case assumption.

Case 2: $CS_{j-1}.prev \neq null$**:**

By the inductive hypothesis, we have: $CS_i.next.first \rightarrow^c CS_{j-1}.first$. –[IH]

Consider two sub-cases:

Case a: $CS_{j-1}.next = null$**:**

Therefore, by definition, $CS_j \mapsto CS_{j-1}$. By [A], we have $j \leq j-1$ giving a contradiction.

Case b: $CS_{j-1}.next \neq null$**:**

Since by the choice of j, $CS_j.prev \neq null$, we have by the definition of

$\rightsquigarrow^c$: $CS_{j-1}.next.first \rightsquigarrow^c CS_j.first$. Together with [IH], by transitivity, this gives: $CS_i.next.first \rightarrow^c CS_j.first$.

□ (Proof of Claim)

We now show that $\langle E, \rightarrow^c \rangle$ is a controlling computation. Suppose it is not. Then, let C be a consistent cut in $\langle E, \rightarrow^c \rangle$ such that $\neg\phi_{mutex}(C)$. Therefore, we have distinct i and j such that $critical(C[i])$ and $critical(C[j])$.

Let $C[i] \in CS_k$ and $C[j] \in CS_l$. Without loss of generality, $k < l$. (Since $i \neq j$, $k \neq l$).

Consider two cases:

Case 1: $CS_k.next = null \ \vee \ CS_l.prev = null$**:**

Therefore, by definition, $CS_l \mapsto CS_k$. By [A], $l \leq k$, contradicting $k < l$.

Case 2: $CS_k.next \neq null \ \wedge \ CS_l.prev \neq null$**:**

Therefore, by Claim 1, we have: $CS_k.next.first \rightarrow^c CS_l.first$. Since $C[i] \in CS_k$, we have: $C[i].next \preceq CS_k.next.first$ and since $C[j] \in CS_j$, we have: $CS_l.first \preceq C[j]$. Therefore, by transitivity, $C[i].next \rightarrow^c C[j]$, violating the consistency of C.

□

Algorithm Description

The proof of the above theorem provides a simple algorithm for finding a controlling computation based on topologically sorting the interval graph of critical sections. We provide a more efficient algorithm in Figure 3.4, making use of the fact that the critical sections in a process are totally ordered. The central idea used in the algorithm is to maintain a frontier of critical sections $(CS_1, \cdots, CS_n)$ that advances from the start of the computation to the end. Instead of finding a minimal critical

Types:

event:	(*proc*: int; v: vector clock)
interval:	(*proc*: int; *critical*: boolean; *first*: event; *last*: event)

Input:

$\mathcal{I}_1, \mathcal{I}_2, \cdots, \mathcal{I}_n$:	list of *interval*, initially non-empty

Output:

$\mathcal{O}$:	list of (*event*, *event*), initially *null*

Vars:

$I_1, I_2, \cdots, I_n$:	*interval*, initially $\forall i: \ I_i = \mathcal{I}_i.head$
valid_cs:	set of *interval*
chain:	list of *interval*
crossed:	*interval*
prev, *curr*:	*interval*

Notation:

$$CS_i = \begin{cases} \min X \in \mathcal{I}_i : \ I_i \preceq_i X \ \wedge \ X.critical = true, & \text{if exists} \\ null, & \text{otherwise} \end{cases}$$

$$CS = \{ \ CS_i \ | \ (1 \leq i \leq n) \ \wedge \ (CS_i \neq null) \ \}$$

$$select(Z) = \begin{cases} \text{arbitrary element of set } Z, & \text{if } Z \neq \emptyset \\ null, & \text{otherwise} \end{cases}$$

Procedure:

while $(CS \neq \emptyset)$ do
    $valid\_cs := \{ \ c \in CS \ | \ \forall c' \in CS: \ (c' \neq c) \Rightarrow (c' \not\rightarrow c) \ \}$
    if $(valid\_cs = \emptyset)$
        exit("no controlling computation exists")
    $crossed := select(valid\_cs)$
    $chain.add\_head(crossed)$
    $I_{crossed.proc} := crossed.next$
if $(chain \neq null)$
    $prev := chain.delete\_head()$
while $(chain \neq null)$ do
    $curr := chain.delete\_head()$
    $\mathcal{O}.add\_head(\ (curr.next.first, prev.first) \ )$
    $prev := curr$

Figure 3.4: Algorithm for Predicate Control of the Mutual Exclusion Predicate

section of the whole interval graph, we merely find the minimal critical section in the current frontier. It is guaranteed to be a minimal critical section of the remaining critical sections in the interval graph at that point. Therefore, this procedure achieves a topological sort.

The main while loop of the algorithm executes p times in the worst case, where p is the number of critical sections in the computation. Each iteration takes $O(n^2)$, since it must compute the *valid_cs*. Thus, a simple implementation of the algorithm will have a time complexity of $O(n^2p)$. However, a better implementation of the algorithm would amortize the cost of computing *valid_cs* over multiple iterations of the loop. Each iteration would compare each of the critical sections that has newly reached the head of its list with the existing critical sections. Therefore, each of the p critical section reaches the head of the list just once, when it is compared with $n-1$ critical sections. The time complexity of the algorithm with this improved implementation is, therefore, $O(np)$. Note that a naive algorithm based directly on the constructive proof of the sufficient condition in Theorem 5 would take $O(p^2)$. We have reduced the complexity significantly by using the fact that the critical sections in a process are totally ordered.

We next prove the correctness of the algorithm.

Lemma 15 *The algorithm in Figure 3.4 is well-specified.*

Proof: To prove that the algorithm is well-specified, the only non-trivial check is for the term $curr.next.first$ at line L23. To show that $curr.next \neq null$ at this point, we prove that, after the termination of loop L13-L19, for all critical sections c in $chain$: $(c.next = null) \Rightarrow (c = chain.head)$. Let c be a critical section in $chain$ such that $c.next = null$. Consider the point when c is about to be added to $chain$ in line L18. Since $c \in valid\_cs$, we have $\forall c' \in valid\_cs : (c' \neq c) \Rightarrow (c' \not\mapsto c)$ However, since $c.next = null$, we have $\forall c' \in \mathcal{C} : c' \mapsto c$. Therefore, c is the only element of $valid\_cs$ at this point. It follows that after line L19, $CS = \emptyset$ and the loop terminates. Therefore, after the termination of loop L13-L19, $c = chain.head$.

The algorithm terminates since the loop L13-L19 advances one critical section in each iteration and the loop L20-L24 decreases *chain* by one critical section in each iteration. $\Box$

Lemma 16 *If the algorithm in Figure 3.4 exits at line L16, then no controlling computation exists.*

Proof: At the point of exit at line 16 $valid\_cs = \emptyset$. Therefore, the graph $\langle CS, \mapsto \rangle$ has no minimal element. Since $CS \neq \emptyset$ (by the terminating condition at line L13), there must be a non-trivial cycle in $\langle CS, \mapsto \rangle$. Therefore, by Theorem 4, no controlling computation exists. $\Box$

Lemma 17 *If the algorithm in Figure 3.4 terminates normally, then $\langle E, \rightarrow^c \rangle$ is a controlling computation of ϕ_{mutex} in $\langle E, \rightarrow \rangle$, where $\rightarrow^c$ is the transitive closure of the union of $\rightarrow$ with the set of edges in $\mathcal{O}$.*

Proof: We show that, at the termination of loop L13-L19, *chain* is a topological sort of the graph $\langle \mathcal{C}, \mapsto \rangle$. Then, since lines L20-L24 construct the same $\leadsto^c$ relation as in the proof of Theorem 5, $\langle E, \rightarrow^c \rangle$ is a controlling computation of ϕ_{mutex} in $\langle E, \rightarrow \rangle$.

Let *chain.set* denote the set of intervals in *chain*. First, we note that the algorithm maintains the following invariant:

INV1: $\forall c \in \mathcal{C} : \; c \prec CS_{c.proc} \Rightarrow c \in chain.set$

Next, consider the point just before L17, when *crossed* is about to be added to *chain*. Let c be any critical section in $\mathcal{C} - chain.set$ such that $c \neq crossed$. By INV1, we have $CS_{c.proc} \preceq c$. If $c \mapsto crossed$ then $CS_{c.proc} \mapsto crossed$, contradicting $crossed \in valid\_cs$. Therefore $c \not\mapsto crossed$. Therefore, *crossed* is minimal in $\langle C - chain.set, \mapsto \rangle$. Therefore, *chain* is a topological sort of $\langle C, \mapsto \rangle$. $\Box$

Theorem 6 *The algorithm in Figure 3.4 solves the predicate control problem for the mutual exclusion predicate.*

Proof: This follows directly from Lemmas 15, 16, and 17. □

3.6 Readers Writers Predicates

Let $critical_1, critical_2, \cdots, critical_n$ be n local predicates and let $critical(e) \equiv critical_{e.proc}(e)$ and let a *critical section* (*non-critical section*) denote a true (false) interval in $\langle E, \rightarrow \rangle$ with respect to $critical_1, critical_2, \cdots, critical_n$.

- $\langle$**read-critical/write-critical**$\rangle$ **:** A critical section is either *read-critical* or *write-critical* .

- $\langle write\_critical(I) \rangle$ **:** We define $write\_critical(I)$ to be true for a write-critical section I and false for all other intervals. We also say $write\_critical(e)$ for all events e in a write-critical section.

- $\langle read\_critical(I) \rangle$ **:** We define $read\_critical(I)$ to be true for a read-critical section I and false for all other intervals. We also say $read\_critical(e)$ for all events e in a read-critical section.

- $\langle$**readers writers predicate**$\rangle$ **:** The *readers writers predicate* ϕ_{rw} is defined as:

$$\phi_{rw}(C) \equiv \forall \; distinct \; i, j : \neg \; (\; critical(C[i]) \; \wedge \; write\_critical(C[j]) \;)$$

The readers writers predicate is a generalized form of mutual exclusion allowing critical sections to be read or write-critical. Two critical sections cannot enter at the same time if one of them is write-critical. The next two theorems establish the necessary and sufficient conditions for solving the predicate control problem for readers writers predicates. Since the algorithm that will be presented in Section 3.8

is applied to readers writers predicates without significant simplification, we do not present a specialized algorithm for readers writers predicates.

Theorem 7 *Let $\langle \mathcal{I}, \mapsto \rangle$ be the interval graph of a computation $\langle E, \rightarrow \rangle$ under local predicates $critical_1, critical_2, \cdots, critical_n$. If $\langle \mathcal{I}, \mapsto \rangle$ contains a non-trivial cycle of critical sections containing at least one write-critical section, then there is no controlling computation of ϕ_{rw} in $\langle E, \rightarrow \rangle$.*

Proof:

We prove the result by contradiction. In interval graph $\langle \mathcal{I}, \mapsto \rangle$, let X be a non-trivial cycle of critical sections containing a write-critical section W. Let $\langle E, \rightarrow^c \rangle$ be a controlling computation of ϕ_{mutex} in $\langle E, \rightarrow \rangle$. Let $<^c$ be a linearization of $\rightarrow^c$ forming a controlling run $\langle E, <^c \rangle$ (by 2).

We next show that there is a consistent cut C in the run $\langle E, <^c \rangle$ such that $\neg \phi_{rw}(C)$ contradicting the fact that $\langle E, <^c \rangle$ is a controlling computation. Consider two cases:

Case 1: $W.next = null$ **:**

Let CS be the critical section following W in X. Since X is non-trivial, $CS \neq W$. Consider two cases:

Case a: $CS.next = null$**:** Let C be the maximum cut in the run $\langle E, \rightarrow \rangle$. It is easy to verify that C is consistent (as in any computation). Since $W.next = null$ and $CS.next = null$, $\neg \phi_{rw}(C)$.

Case b: $CS.next \neq null$**:** Let $e = CS.next.first$ and $C' = lcc(e)$. Consider two cases:

$W.prev = null$: Since $W.first \in \bot$, we have $W.first \preceq C'[W.proc]$.

$W.prev \neq null$: Since $W \mapsto CS$, we have $W.first \rightarrow e$ and so, $W.first <^c e$. Therefore, since C' is consistent, $W.first \preceq C'[W.proc]$

In both cases, $W.first \preceq C'[W.proc]$ –[A]

Since $W.next = null$, we have: $C'[W.proc] \preceq W.last$ –[B]

By [A] and [B], $C'[W.proc] \in W$ and so $write\_critical(C'[W.proc])$. Let $C = lcc\text{-}prev(e)$. Since, $\forall j : j \neq e.proc : C[j] = C'[j]$, we have: $write\_critical(C[W.proc])$. Further, since $e = CS.next.first$, we have $C[CS.proc] = CS.last$, and therefore, $critical(C[CS.proc])$. So, $\neg\phi_{rw}(C)$. Also, by Lemma 6, C is consistent.

Case 2: $W.next \neq null$ **:**

Let $C' = lcc\text{-}prev(W.next.first)$. If there is a CS in X distinct from W such that: $CS.first \preceq C'[CS.proc] \preceq CS.last$, then since C' is consistent and $\neg\phi_{rw}(C)$, we are done. Therefore, assume that for all critical sections CS in X distinct from W: $(C'[CS.proc] \prec CS.first) \vee (CS.last \prec C'[CS.proc])$. –[D]

Let CS_1 and CS_2 be any two consecutive critical sections in X. Suppose that $CS_2.first \preceq C'[CS_2.proc]$. –[E]

Therefore, applying [D], we have: $CS_2.last \prec C'[CS_2.proc]$. –[F]

Consider two cases:

Case a: $CS_1.prev = null$**:** In this case, since $CS_1.first \in \perp$ we have: $CS_1.first \preceq C'[CS_1.proc]$.

Case b: $CS_1.prev \neq null$**:** Since $CS_1 \mapsto CS_2$ and since $CS_2.next \neq null$ (from [F]), we have: $CS_1.first \rightarrow CS_2.next.first$. This, together with [F] and transitivity gives us: $CS_1.first \in C'.past$. Since C' is consistent, $CS_1.first \preceq C'[CS_1.proc]$.

Therefore, in both cases: $CS_1.first \preceq C'[CS_1.proc]$. Combining this with our supposition [E], we have: $\forall CS_1, CS_2 \in X : CS_2.first \preceq C'[CS_2.proc] \Rightarrow$

$CS_1.first \preceq C'[CS_1.proc]$. Therefore, since $W.first \preceq C'[W.proc]$ (by the definition of C'), we have: $\forall CS \in X : CS.first \preceq C'[CS.proc]$. Combining this with [D] we have: $\forall CS \in X : CS = W \lor CS.next.first \preceq C'[CS.proc]$. –[G]

Let CS be the critical section following W in the cycle X.
Let $C = lcc\text{-}prev(CS.next.first)$. From [G], $CS.next.first \preceq C'[CS.proc]$, and so: $C[CS.proc] \prec C'[CS.proc]$. Therefore, by Lemma 7 we have: $C < C'$. So, we have: $C[W.proc] \preceq C'[W.proc]$. Using the definition of C': $C[W.proc] \preceq W.last$. –[H]

Consider three cases:

Case a: $W.prev = null$**:**

Therefore, $W.first \in \perp$ and so $W.first \preceq C[W.proc]$.

Case b: $W.proc = CS.proc$**:**

In this case, $W \prec CS$ and, therefore, $W.first \preceq C[w.proc]$.

Case c: $W.prev \neq null \land W.proc \neq CS.proc$ **:**

Since $W \mapsto CS$ and $CS.next \neq null$ (because of [G]), we have: $W.first \rightarrow CS.next.first$. Therefore, $W.first \in lcc(CS.next.first).past$. By the consistency of $lcc(CS.next.first)$, we have:
$W.first \preceq lcc(CS.next.first)[W.proc]$. Since $W.proc \neq CS.proc$, by the definition of $lcc\text{-}prev$:
$lcc\text{-}prev(CS.next.first)[W.proc] = lcc(CS.next.first)[W.proc]$.
Therefore, $W.first \preceq C[W.proc]$.

In all cases: $W.first \preceq C[W.proc]$. This, together with [H], gives us: $C[W.proc] \in W$. Therefore, $write\_critical(C[W.proc])$, and by the definition of C, we have $critical(C[CS.proc])$. Therefore, $\neg\phi_{rw}(C)$. Further C is consistent by definition.

□

Theorem 8 *Let $\langle \mathcal{I}, \mapsto \rangle$ be the interval graph of a computation $\langle E, \rightarrow \rangle$ under local predicates $critical_1, critical_2, \cdots, critical_n$. If $\langle \mathcal{I}, \mapsto \rangle$ does not contain a non-trivial cycle of critical sections containing at least one write-critical section, then there is a controlling computation of ϕ_{rw} in $\langle E, \rightarrow \rangle$.*

Proof: Let $\mathcal{C}$ be the set of critical sections in $\mathcal{I}$. Let $\mapsto_{\mathcal{C}}$ denote the relation $\mapsto$ restricted to the set $\mathcal{C}$. Let $\mathcal{S}$ be the set of strongly connected components of the graph $\langle \mathcal{C}, \mapsto_{\mathcal{C}} \rangle$. Let $\hookrightarrow$ be a relation defined on strongly connected components as:

$$SCC \ \hookrightarrow \ SCC' \ \ \equiv \ \ \exists CS \in SCC, CS' \in SCC' : \ CS \ \mapsto \ CS'$$

Clearly, $\hookrightarrow$ is a partial order on $\mathcal{S}$. Therefore, we can topologically sort the graph $\langle \mathcal{S}, \hookrightarrow \rangle$ to give a sequence of strongly connected components:

$$SCC_1, SCC_2, \cdots, SCC_l \ \text{ such that } \ \ SCC_i \ \hookrightarrow \ SCC_j \ \Rightarrow \ i \leq j \quad \text{–[A]}$$

Define a relation $\rightsquigarrow^c$ on events in E as the set of tuples:

$$\{ \ (CS_i.next.first, CS_j.first) \mid CS_i.next \neq null \ \wedge \ CS_j.prev \neq null \ \wedge \\ \exists k \in \{1, \cdots, l-1\} : CS_i \in SCC_k \ \wedge \ CS_j \in SCC_{k+1} \ \}$$

Define $\rightarrow^c$ to be the transitive closure of $\rightarrow \ \cup \rightsquigarrow^c$.

We first show by contradiction that $\langle E, \rightarrow^c \rangle$ is a computation. If it is not, then there must be a cycle, say X, in the graph $\langle E, \rightarrow \cup \rightsquigarrow^c \rangle$. Since $\langle E, \rightarrow \rangle$ is a computation, X must have at least one $\rightsquigarrow^c$ edge. Consider the set of $\rightsquigarrow^c$ edges in X, and let $CS_i.next.first \rightsquigarrow^c CS_j.first$ be the edge such that CS_i belongs to the maximum strongly connected component (among such edges) in the topological sorted order (i.e. $CS_i \in SCC_k$ and k has the maximum value). Let $CS_i \in SCC_k$, and so, $CS_j \in SCC_{k+1}$. Let $CS_a.next.first \rightsquigarrow^c CS_b.first$ be the next $\rightsquigarrow^c$ edge in the cycle X (note that it is possible that $a = i$). Therefore, $CS_j.first \ \underline{\rightarrow} \ CS_a.next.first$. Since, by definition, two critical sections cannot be contiguous, the equality cannot hold. Therefore, $CS_j.first \ \rightarrow \ CS_a.next.first$, and so, by definition, $CS_j \ \mapsto \ CS_a$. So, if $CS_a \in SCC_p$, we have: $SCC_{k+1} \ \hookrightarrow \ SCC_p$. Therefore, by [A], $k+1 \leq p$

and so $k < p$. This contradicts the choice of k as maximum. Therefore, $\langle E, \rightarrow^c \rangle$ is a computation.

Before showing that $\langle E, \rightarrow^c \rangle$ is a controlling computation, we first prove a claim:

Claim 1: For all critical sections $CS \in SCC_p$ and $CS' \in SCC_q$ such that $p < q$, if $CS.next \neq null$ and $CS'.prev \neq null$, then: $CS.next.first \ \rightarrow^c \ CS'.first$.

Proof: (by induction on $q - p$)

Base Case: (q - p = 1): Directly from the definition of $\rightsquigarrow^c$.

Inductive Case: (q - p ¿ 1): Let $CS \in SCC_p$ and $CS' \in SCC_q$ such that $p < q$, if $CS.next \neq null$ and $CS'.prev \neq null$. Let CS'' be any critical section in SCC_{q-1}. We have two cases:

Case 1: $(CS''.prev = null)$**:**

Therefore, by definition: $CS'' \mapsto CS$, and so $SCC_{q-1} \ \hookrightarrow \ SCC_p$. Using [A], we have: $(q-1) \leq p$. Together with $p < q$, this implies that: $q = p + 1$ which violates the inductive case assumption.

Case 2: $(CS''.prev \neq null)$**:**

Subcase 2.1: $(CS''.next = null)$

Therefore, by definition, $CS' \ \mapsto \ CS''$, and so $SCC_q \ \hookrightarrow \ SCC_{q-1}$. Using [A], it follows that $q \leq (q-1)$. Thus, we have a contradiction.

Subcase 2.2: $(CS''.next \neq null)$

Therefore, $CS''.next.first \rightsquigarrow^c CS'.first$. Further, by the inductive hypothesis, $CS.next.first \ \rightarrow^c \ CS''.first$. If follows, by transitivity, that: $CS.next.first \ \rightarrow^c \ CS''.first$.

□ (Proof of Claim 1)

We now show that $\langle E, \rightarrow^c \rangle$ is a controlling computation. Suppose it is not. Then, let C be a consistent cut of $\langle E, \rightarrow^c \rangle$ such that $\neg\phi_{rw}(C)$. Therefore, we have distinct i and j such that $write\_critical(C[i])$ and $critical(C[j])$. Let $C[i] \in CS$ and $C[j] \in CS'$. Further let, $CS \in SCC_p$ and $CS \in SCC_q$. Without loss of generality, let $p \leq q$. Consider two cases:

Case 1: $p = q$**:**

In this case, both CS and CS' are in the same strongly connected component of $\langle \mathcal{C}, \mapsto_{\mathcal{C}} \rangle$. Further CS is a write-critical section and CS' is a distinct critical section. Therefore, there is a non-trivial cycle of critical sections in $\langle \mathcal{I}, \mapsto \rangle$ containing a write-critical section, thus, contradicting the assumptions.

Case 2: $p < q$**:**

Subcase 2.1: $CS.next = null \ \vee \ CS'.prev = null$**:**

Therefore, by definition, $CS' \ \mapsto \ CS$, and so, $SCC_q \ \hookrightarrow \ SCC_p$. Using [A], we have: $q \leq p$ which is a contradiction.

Subcase 2.2: $CS.next \neq null \ \wedge \ CS'.prev \neq null$**:**

Therefore, by Claim 1, we have: $CS.next.first \ \rightarrow^c \ CS'.first$. Since $C[i] \in CS$, we have: $C[i].next \preceq CS.next.first$ and since $C[j] \in CS'$, we have: $CS'.first \preceq C[j]$. Therefore, by transitivity, $C[i].next \rightarrow^c C[j]$ violating the consistency of C.

□

3.7 Independent Mutual Exclusion Predicates

Let $critical_1, critical_2, \cdots, critical_n$ be n local predicates and let $critical(e) \equiv critical_{e.proc}(e)$ and let a *critical section* (*non-critical section*) denote a true (false) interval in $\langle E, \rightarrow \rangle$ with respect to $critical_1, critical_2, \cdots, critical_n$.

- $\langle$**k-critical**$\rangle$ **:** A critical section is k-critical for some $k \in \{1, \cdots m\}$.

- $\langle k\_critical(I)\rangle$ **:** We define $k\_critical(I)$ to be true for a k-critical section I and false for all other intervals. We also say $k\_critical(e)$ for all events e in a k-critical section.

- $\langle$**independent mutual exclusion predicate**$\rangle$ **:** The *independent mutual exclusion predicate* ϕ_{ind} is defined as:

$$\phi_{ind}(C) \ \equiv \ \forall \ distinct \ i, j : \forall \ k : \ \neg \ (\ k\_critical(C[i]) \ \wedge \ k\_critical(C[j]) \ )$$

The independent mutual exclusion predicate is a generalized form of mutual exclusion allowing critical sections to have types in $k \in \{1, \cdots m\}$. Two critical sections cannot enter at the same time if they are of the same type. The next result shows us that the problem becomes hard for this generalization. However, we will determine sufficient conditions for solving it that allow us to solve the problem efficiently under certain conditions.

Theorem 9 *The predicate control problem for the independent mutual exclusion predicate is NP-Complete.*

Proof:

Let PC-IND denote the predicate control problem for the independent mutual exclusion predicate. PC-IND is in NP for the same reasons that the general Predicate Control problem is in NP. We prove that PC-IND is NP-Hard by transforming 3SAT.

Let $U = \{u_1, u_2, \cdots u_w\}$ be a set of variables, and $C = \{c_1, c_2, \cdots c_x\}$ be a set of clauses, forming an instance of 3SAT. Let l_{i1}, l_{i2}, l_{i3} be the three literals in clause c_i.

To construct an instance of PC-IND from this instance of 3SAT, we first construct a set of events E as follows:

- For each clause c_i define 6 processes with event sets $E_{i1}, E_{i2}, E_{i3}, \cdots, E_{i6}$. Each E_{ip} has two events denoted by e_{ip1} and e_{ip2}. Let $E_i = E_{i1} \cup E_{i2} \cup \cdots \cup E_{i6}$.

- For each variable u_r define 2 processes with event sets E'_{r1}, E'_{r2}. Each E'_{rq} has two events denoted by e'_{rq1} and e'_{rq2}. Let $E'_r = E'_{r1} \cup E'_{r2}$.

Let $E = \bigcup_i E_i \quad \cup \quad \bigcup_r E'_r$ and $n = |E| = 2w + 6x$.

Let $\rightsquigarrow$ be a relation on E defined as follows:

- Within each E_i: $e_{i22} \rightsquigarrow e_{i31}$, $e_{i42} \rightsquigarrow e_{i51}$, $e_{i62} \rightsquigarrow e_{i11}$

- For each literal $l_{ip} \in c_i$ such that $l_{ip} = u_r$ for some variable u_r:
  $e_{i(2p)1} \rightsquigarrow e'_{r22}$, $e'_{r11} \rightsquigarrow e_{i(2p-1)1}$, $e_{i(2p-1)1} \rightsquigarrow e'_{r12}$, $e'_{r21} \rightsquigarrow e_{i(2p)1}$

- For each literal $l_{ip} \in c_i$ such that $l_{ip} = \overline{u_r}$ for some variable u_r:
  $e_{i(2p)1} \rightsquigarrow e'_{r12}$, $e'_{r21} \rightsquigarrow e_{i(2p-1)1}$, $e_{e(2p-1)1} \rightsquigarrow e'_{r22}$, $e'_{r11} \rightsquigarrow e_{i(2p)1}$

Let $\prec$ be a relation on E such that the local events in an E_{ip} or an E_{rq} are totally ordered by $\prec$ so that: $e_{ip1} \prec e_{ip2}$ and $e'_{rq1} \prec e'_{rq2}$.

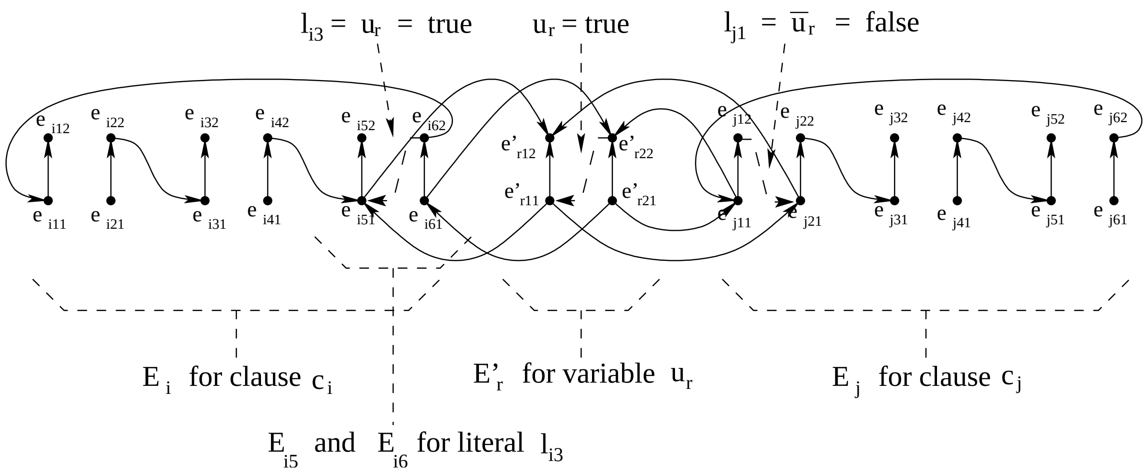

Figure 3.5: Proof: PC-IND is NP-Complete

Let $\rightarrow$ be the transitive closure of $\rightsquigarrow \cup \prec$. To show that $\rightarrow$ is irreflexive, we prove that there are no cycles in the graph $\langle E, \rightsquigarrow \cup \prec \rangle$. Any event in an E'_{rq} has either incoming or outgoing edges but not both. Therefore, no event in an E'_{rq} can be involved in a cycle. Since any distinct E_i and E_j have no edges between them, the only remaining possibility is a cycle within an E_i. Considering only edges within E_i, any E_{ip} has either incoming edges to its events or outgoing edges from its events but not both. Further, there are no cycles within an E_{ip}. Therefore, there is no cycle within an E_i. Thus, we have shown that $\rightarrow$ is irreflexive. Clearly, $\rightarrow$ is also transitive and so, $\langle E, \rightarrow \rangle$ is a computation.

Define local predicates $critical_1, critical_2, \cdots, critical_n$, such that each process consists of exactly one critical section containing only its first event. So, for each E_{ip}, there is exactly one critical section, $CS_{ip} = \{e_{ip1}\}$ and for each E'_{rq}, there is exactly one critical section $CS'_{rq} = \{e'_{rq1}\}$.

The critical sections in an E_i are $i\_critical$ and the critical sections in an E'_r are $(x+r)\_critical$. Therefore, there are $m = (x+w)$ independent types of critical sections. Define global predicate ϕ_{ind} based on these critical section definitions.

Thus, we have a polynomial construction from an instance $(U,\ C)$ of 3SAT to an instance $(\langle E, \rightarrow \rangle,\ \phi_{ind})$ of PC-IND. We next show that the construction is indeed a transformation.

Part 1: Let t be a truth assignment on variables in U such that t satisfies the set of clauses in C. Define a relation $\rightsquigarrow^{c1}$ on E as follows:

- For each variable u_r:
  if $t(u_r) = \textit{true}$ then $e'_{r22} \rightsquigarrow^{c1} e'_{r11}$
  if $t(u_r) = \textit{false}$ then $e'_{r12} \rightsquigarrow^{c1} e'_{r21}$

- For each literal l_{ip} in a clause c_i:
  if l_{ip} is *true* under t then $e_{i(2p)2} \rightsquigarrow^{c1} e_{i(2p-1)1}$
  if l_{ip} is *false* under t then $e_{i(2p-1)2} \rightsquigarrow^{c1} e_{i(2p)1}$

For any i, let $\mathcal{C}_i$ denote the set of 6 critical sections in E_i. Define $\mapsto_i^{c1}$ relation on $\mathcal{C}_i$ as:

$$\forall j,k \in \{1, \cdots, 6\} : \ CS_{ij} \ \mapsto_i^{c1} \ CS_{ik} \ \ \equiv \ \ e_{ij2} \rightsquigarrow^{c1} e_{ik1} \ \vee \ e_{ij2} \rightsquigarrow e_{ik1}$$

Depending on whether each l_{ip} is *true* or *false* under t, we have 8 cases (by the definition of $\rightsquigarrow^{c1}$). Since t satisfies clause c_i, at least one of l_{i1}, l_{i2}, l_{i3} must be *true* under t, and one of the cases is not possible. It is easy to verify that for each of the remaining 7 cases, there are no cycles in the graph $\langle \mathcal{C}_i, \mapsto_i^{c1} \rangle$.

Therefore, we can topologically sort $\langle \mathcal{C}_i, \mapsto_i^{c1} \rangle$. Consider a topological sort in which all the maximal critical sections are selected last. Therefore, if p_i is a permutation of $\{1, \cdots, 6\}$ representing the topological sort, then:

TOP1: $\forall j, k : \ CS_{ij} \ \mapsto_i^{c1} \ CS_{ik} \ \ \Rightarrow \ \ p_i(j) < p_i(k)$, and

TOP2: $\forall j : \ CS_{ij}$ is maximal in $\langle \mathcal{C}_i, \mapsto_i^{c1} \rangle \ \ \Rightarrow$

$$(\ \forall k : \ p_i(k) > p_i(j) \ \Rightarrow \ CS_{ik} \text{ is maximal in } \langle \mathcal{C}_i, \mapsto_i^{c1} \rangle \ )$$

Define a relation $\rightsquigarrow^{c2}$ on E that orders critical sections in $\mathcal{C}_i$ in the topologically sorted order as follows:

- In each E_i:
  $\forall j, k : \ (\ p_i(j) = p_i(k) - 1 \ \wedge \ CS_j \not\mapsto_i^{c1} CS_k \ ) \ \Rightarrow \ e_{ij2} \rightsquigarrow^{c2} e_{ik1}$

Let $\rightarrow^c$ be the transitive closure of $\rightarrow \ \cup \rightsquigarrow^{c1} \ \cup \ \rightsquigarrow^{c2}$. We next prove that $\rightarrow^c$ is irreflexive. This is equivalent to proving the following:

Claim 1: There is no cycle in $\prec \ \cup \rightsquigarrow \ \cup \ \rightsquigarrow^{c1} \ \cup \ \rightsquigarrow^{c2}$.

Proof: We consider three cases for a cycle and prove that each case is not possible.

Case 1: A cycle within an E'_r: The cycle cannot be within a single E'_{rq} since there is only one edge ($\prec$) between the two events. Since t assigns either *true* or *false* to variable u_r, there is only one edge ($\rightsquigarrow^{c1}$) between E'_{r1} and E'_{r2}. Therefore, there can be no cycles within an E'_r.

Case 2: A cycle within an E_i: The cycle cannot be within a single E_{ij} since there is only one edge ($\prec$) between the two events. We prove by contradiction that there cannot be a cycle intersecting multiple E_{ij}'s. Suppose there is such a cycle X. Among all the intersected E_{ij}'s, consider E_{ik} such that $p_i(k) =$

$max\{p_i(j) : E_{ij} \cap X \neq \emptyset\}$ (i.e. CS_{ik} is the last critical section in the topological sort of $\mapsto_i^{c1}$ among all intersected $CSij$'s). Let the cycle X intersect E_{il} immediately after E_{ik}. Therefore, there must be an edge ($\rightsquigarrow$, $\rightsquigarrow^{c1}$, or $\rightsquigarrow^{c2}$) from e_{ik2} to e_{il1}. Therefore, either it is a $\rightsquigarrow^{c2}$ edge or $CS_{ik} \mapsto_i^{c1} CS_{il}$. So, using the definition of $\rightsquigarrow^{c2}$ or TOP1, we have $p_i(k) < p_i(l)$. This contradicts the choice of k.

Case 3: All other cycles: For any distinct E_i and E_j there are no edges between them. Therefore, we have only to consider cycles that intersect at least one E'_r. Let X be such a cycle intersecting E'_r. We consider the case when $t(u_r) = true$. The case $t(u_r) = false$ follows along similar lines. By the definition of $\rightsquigarrow^{c1}$, we have $e'_{r22} \rightsquigarrow^{c1} e'_{r11}$. If this edge were excluded, no event in E'_r would have both incoming and outgoing edges. Therefore, the cycle X must include this edge. From the definition of $\rightsquigarrow$, there are two possible cases for the next edge in the cycle X:

Case 1: $e'_{r11} \rightsquigarrow e_{i(2p-1)1}$ is the next edge in cycle X, for some i and p:

In this case, the literal $l_{ip} = u_r$ and, therefore, is *true* under t. Therefore, by the definition of $\rightsquigarrow$ and $\rightsquigarrow^{c1}$, there are no outgoing edges from $E_{i(2p-1)}$ in $\rightsquigarrow \cup \rightsquigarrow^{c1}$. Therefore, $CS_{i(2p-1)}$ is maximal in $\langle \mathcal{C}_i, \mapsto_i^{c1} \rangle$.

Since the cycle X contains at least one maximal critical section in $\langle \mathcal{C}_i, \mapsto_i^{c1} \rangle$, consider the critical section CS_{ik} such that: $p_i(k) = max\{p_i(j) : CS_{ij}$ is maximal in $\langle \mathcal{C}_i, \mapsto_i^{c1} \rangle \wedge E_{ij} \cap X \neq \emptyset\}$ (i.e. CS_{ik} is the last maximal critical section in the topological sort of $\langle \mathcal{C}_i, \mapsto_i^{c1} \rangle$ that is intersected by cycle X). Since CS_{ik} is maximal in $\langle \mathcal{C}_i, \mapsto_i^{c1} \rangle$, E_{ik} can have no outgoing edges in $\rightsquigarrow \cup \rightsquigarrow^{c1}$. Therefore the outgoing edge from E_{ik} in the cycle must be: $e_{ik2} \rightsquigarrow^{c2} e_{iq1}$ such that $p(q) = p(k) + 1$. Since, CS_{iq} is also maximal in $\langle \mathcal{C}_i, \mapsto_i^{c1} \rangle$ (by TOP2), this contradicts the choice of $p_i(k)$.

Case 2: $e'_{r11} \rightsquigarrow e_{i(2p)1}$ is the next edge in cycle X, for some i and p:

In this case, the literal $l_{ip} = \overline{u_r}$ and, therefore, is *false* under t. Therefore, using $CS_{i(2p)}$ as the maximal critical section in X we arrive at a contradiction in a similar manner to Case 1.

End of Proof (Claim 1)

Therefore, $\langle E, \rightarrow^c \rangle$ is a computation. It is easy to verify that for any k such that $1 \leq k \leq (x + w)$, all of the k-critical sections can be arranged in a sequence $CS_1, \cdots, CS_m$ such that $\forall i : \; CS_i.next.first \; \rightarrow^c \; CS_{i+1}.first$ (where m is 6 or 2 depending on whether $1 \leq k \leq x$ or $x < k \leq (x+w)$). Therefore, we can prove that $\langle E, \rightarrow^c \rangle$ is a controlling computation along similar lines to the proof of Theorem 5.

Part 2: Let $\langle E, \rightarrow^c \rangle$ be a controlling computation of ϕ_{ind} in $\langle E, \rightarrow \rangle$. Before constructing a truth assignment on U, we prove:

Claim 2: For any two distinct k-critical sections CS and CS', either

(1) $CS.next.first \; \rightarrow^c \; CS'.first$, or

(2) $CS'.next.first \; \rightarrow^c \; CS.first$,

but not both.

Proof: Clearly both (1) and (2) cannot be true since otherwise there would be a cycle in $\rightarrow^c$. Suppose both (1) and (2) are false. Let C be the consistent cut that is the least upper bound (*lub*) of $lcc(CS.first)$ and $lcc(CS'.first)$ (using Lemma 4). Since (1) is false, $CS.next.first \notin lcc(CS'.first).past$ and since $CS.first \prec CS.next.first$, we have $CS.next.first \notin lcc(CS.first).past$ (using Lemma 5). Therefore, $CS.next.first \notin C.past$ (by the definition of *lub*). Together with $CS.first \preceq C[CS.proc]$ (by the definition of *lub*), we have: $C[CS.proc] \in CS$. In a similar way, it can be shown that: $C[CS'.proc] \in CS'$. Therefore,

$k\_critical(C[CS.proc]) \ \wedge \ k\_critical(C[CS'.proc])$, and so: $\neg\phi_{ind}(C)$. This contradicts the fact that $\langle E, \rightarrow^c\rangle$ is a controlling computation. *End of Proof (Claim 2)*

We construct a truth assignment t on U such that for a variable u_r:

- $t(u_r) = true$, if $CS'_{r2}.next.first \ \rightarrow^c \ CS'_{r1}.first$, and
- $t(u_r) = false$, if $CS'_{r1}.next.first \ \rightarrow^c \ CS'_{r2}.first$

By Claim 2, this is a valid truth assignment.

Further, let x be a boolean function on literals defined as follows:

- $x(l_{ip}) = true$, if $CS_{i(2p)}.next.first \ \rightarrow^c \ CS_{i(2p-1)}.first$, and
- $x(l_{ip}) = false$, if $CS_{i(2p-1)}.next.first \ \rightarrow^c \ CS_{i(2p)}.first$

Again, by Claim 2, this is a valid definition.

Claim 3: ($x(l_{ip}) = true$) $\equiv$ (l_{ip} is *true* under t)

Proof:

Case 1: $x(l_{ip}) = true$**:**

Therefore, substituting in the definition of x, we get:

$e_{i(2p)2} \ \rightarrow^c \ e_{i(2p-1)1}$ –[A]

Case a: $l_{ip} = u_r$ for some r**:**

Assume $t(u_r) = false$, then (substituting):

$e'_{r12} \ \rightarrow^c \ e'_{r21}$ –[B]

From the definition of $\rightarrow$, we have:

$e'_{r21} \ \rightarrow \ e_{i(2p)1}$ and $e_{i(2p-1)1} \ \rightarrow \ e'_{r12}$ –[C]

From [A], [B], and [C], we have a cycle contradicting the fact that $\langle E, \rightarrow^c\rangle$ is a computation. Therefore, $t(u_r) = true$, and so, l_{ip} is true under t.

Case b: $l_{ip} = \overline{u_r}$ for some r**:**

Assume $t(u_r) = true$, then (substituting):

$e'_{r22} \;\rightarrow^c\; e'_{r11}$ –[D]

From the definition of $\rightarrow$, we have:

$e'_{r11} \;\rightarrow\; e_{i(2p)1}$ and $e_{i(2p-1)1} \;\rightarrow\; e'_{r22}$ –[E]

From [A], [D], and [E], we have a cycle contradicting the fact that $\langle E, \rightarrow^c \rangle$ is a computation. Therefore, $t(u_r) = \mathit{false}$, and so, l_{ip} is true under t.

Case 2: $x(l_{ip}) = \mathit{false}$**:**

Along similar lines to Case 1, we can show that l_{ip} is false under t.

End of Proof (Claim 3)

Now, we show that the truth assignment t satisfies the set of clauses C. Consider any clause c_i. If all of $x(l_{i1})$, $x(l_{i2})$, and $x(l_{i3})$ are *false*, then by the definition of x, $\prec$ and $\rightarrow$, we have a cycle in $\rightarrow^c$: $e_{i12}, e_{i21}, e_{i22}, e_{i31}, e_{i32}, e_{i41}, e_{i42}, e_{i51}, e_{i52}, e_{i61}, e_{i62}$. Therefore, at least one literal, say $x(l_{ip})$ is *true*. It follows from *Claim 3* that l_{ip} is *true* under t. Therefore t satisfies c_i. □

Although the problem is NP-Complete, the next result states a sufficient condition under which it can be solved. The condition is the absence of cycles containing two critical sections of the same type. Under these conditions, we can construct an efficient algorithm. Since the algorithm that will be presented in Section 3.8 is applied to independent mutual exclusion predicates without significant simplification, we do not present a specialized algorithm in this section.

Theorem 10 *Let $\langle \mathcal{I}, \mapsto \rangle$ be the interval graph of a computation $\langle E, \rightarrow \rangle$ under local predicates $critical_1, critical_2, \cdots, critical_n$. If $\langle \mathcal{I}, \mapsto \rangle$ does not contain a non-trivial cycle of critical sections containing two k-critical section for some k, then there is a controlling computation of ϕ_{rw} in $\langle E, \rightarrow \rangle$.*

Proof: The proof is along similar lines to the proof of Theorem 8. □

3.8 Generalized Mutual Exclusion Predicates

Using the definitions of the previous two sections:

- **⟨generalized mutual exclusion predicate⟩ :** The *generalized mutual exclusion predicate* ϕ_{gen_mutex} is defined as:

  $$\phi_{gen\_mutex}(C) \equiv \forall\ distinct\ i,j : \\ \neg\ (\ critical(C[i])\ \wedge\ write\_critical(C[j])\ )\ \wedge \\ \forall\ k :\ \neg\ (\ k\_critical(C[i])\ \wedge\ k\_critical(C[j])\ )$$

Generalized mutual exclusion predicates allow critical sections to have types and be read/write-critical. Clearly the predicate control problem is NP-Complete for generalized mutual exclusion predicates. Further, a similar sufficient condition can be proved combining the sufficient conditions for readers writers and independent mutual exclusion predicates.

Algorithm Description

Based on the proof of the sufficient conditions, we can design a simple algorithm based on determining the strongly connected components in the critical section graph and then topologically sorting them. Instead, we present a more efficient algorithm in Figure 3.6.

In order to understand how the algorithm operates, we require the concept of a "general interval". A general interval is a sequence of intervals in a process that belong to the same strongly connected component of the interval graph. For the purposes of the algorithm, it is convenient to treat such sequences of intervals as a single general interval. We now define general intervals and define a few notations.

- **⟨general interval⟩ :** Given an interval graph $\langle \mathcal{I}, \mapsto \rangle$, a *general interval* is a contiguous sequence of intervals in an $\langle \mathcal{I}_i, \prec_i \rangle$ subgraph.

- $\langle g.first/g.last \rangle$ **:** For a general interval g, let $g.first$ ($g.last$) represent the first and last intervals in g.

- $\langle g.set \rangle$ **:** Let $g.set$ represent the set of intervals in the sequence g.

- $\langle \mapsto$ **(for general intervals)**$\rangle$ **:** Let $\mathcal{G}$ denote the set of general intervals in a computation. We define the (overloaded) relation $\mapsto$ for general intervals as:

$$\forall g, g' \in \mathcal{G} : \ g \mapsto g' \ \equiv \ g.first \mapsto g'.last$$

- $\langle \alpha(g) \rangle$ **:** We say that a general interval g is *true* under a local predicate α if $\alpha(g.first)$. In particular, if g is *true* under a local predicate *critical*, we call g a *general critical section*.

- $\langle \hookrightarrow \rangle$ **:** Let $\mathcal{G}_1 \subseteq \mathcal{G}$ be a set of general intervals. Let $\mathcal{S}_{\mathcal{G}_1}$, the set of strongly connected components in the graph $\langle \mathcal{G}_1, \mapsto \rangle$. We define a relation $\hookrightarrow$ on $\mathcal{S}_{\mathcal{G}_1}$ as follows:

  $$\forall s, s' \in \mathcal{S}_{\mathcal{G}_1} : \ s \hookrightarrow s' \ \equiv \ \exists g \in s, g' \in s' : \ g \mapsto g'$$

  Clearly, $\langle \mathcal{S}_{\mathcal{G}_1}, \hookrightarrow \rangle$ has no cycles.

The algorithm maintains a frontier of general critical sections that advances from the beginning of the computation to the end. In each iteration, the algorithm finds the strongly connected components (scc's) of the general critical sections in the frontier. Then, it picks a minimal strongly connected component, *candidate*, from among them (line L22). However, the *candidate* is not necessarily a minimal scc of the entire critical section graph. In fact, it need not even be an scc of the entire graph. To determine if it is, we find the *mergeable* set of critical sections that immediately follow the general critical sections and belong to the same scc (line L23). If *mergeable* is not empty, the critical sections in *mergeable* are merged with the general critical sections in *candidate* to give larger general critical sections (line

Types:

$event$: $(proc\text{: int; } v\text{: vector clock})$

$gen\_interval$: $(proc\text{: int; } critical\text{: boolean; } first\text{: event; } last\text{: event})$

str_conn_comp: set of $gen\_interval$

Input:

$\mathcal{I}_1, \mathcal{I}_2, \cdots, \mathcal{I}_n$: list of $gen\_interval$, initially non-empty

Output:

$\mathcal{O}$: list of ($event$, $event$), initially $null$

Vars:

$I_1, I_2, \cdots, I_n$: $gen\_interval$, initially $\forall i: \ I_i = \mathcal{I}_i.head$

scc_set: set of str_conn_comp

$valid\_scc$: set of str_conn_comp

$chain$: list of str_conn_comp

$candidate$: str_conn_comp

$prev\_scc, curr\_scc$: str_conn_comp

$prev, curr$: $gen\_interval$

Notation:

$$CS_i = \begin{cases} \min X \in \mathcal{I}_i : \ I_i \preceq_i X \ \wedge \ X.critical = true, & \text{if exists} \\ null, & \text{otherwise} \end{cases}$$

$$CS = \{ \ CS_i \ | \ (1 \le i \le n) \ \wedge \ (CS_i \ne null) \ \}$$

$$get\_scc(CS) = \text{set of strongly connected components in } \langle CS, \mapsto \rangle$$

$$select(Z) = \begin{cases} \text{arbitrary element of set } Z, & \text{if } Z \ne \emptyset \\ null, & \text{otherwise} \end{cases}$$

$not\_valid(X)$ = X has a non-trivial cycle of critical sections with either one write critical section or two k-critical sections

$$merge(c, c') = (c.last := c'.last; \ c.next := c'.next)$$

Procedure:

while ($CS \ne \emptyset$) do
  $scc\_set := get\_scc(CS)$
  $valid\_scc := \{ \ s \in scc\_set \ | \ \forall s' \in scc\_set : \ (s' \ne s) \Rightarrow (s' \not\mapsto s) \ \}$
  $candidate := select(valid\_scc)$
  $mergeable := \{ \ c.next.next \ | \ c \in candidate \ \wedge \ c.next.next \ne null \ \wedge \ \exists c' \in candidate : \ c.next.next \mapsto c' \}$
  if ($mergeable = \emptyset$)
    if ($not\_valid(candidate)$)
      exit("cannot find controlling computation")
    $chain.add\_head(candidate)$
    for ($c \in candidate$) do
      $I_{c.proc} := c.next$
  else
    for ($c \in mergeable$) do
      $merge(CS_{c.proc}, c)$
if ($chain \ne null$)
  $prev\_scc := chain.delete\_head()$
while ($chain \ne null$) do
  $curr\_scc := chain.delete\_head()$
    for ($curr \in curr\_scc$, $prev \in prev\_scc$) do
  $\mathcal{O}.add\_head(\ (curr.next.first, prev.first) \ )$
  $prev\_scc := curr\_scc$

Figure 3.6: Algorithm for Generalized Mutual Exclusion Predicate Control

L32) and the procedure is repeated. If *mergeable* is empty, then it can be shown that *candidate* is a minimal scc of the graph. Therefore, we check that it meets the sufficient conditions of validity (line L25), and then append it to the *chain* (line L27).

Finally, after the main loop terminates, the scc's in the *chain* are connected using added edges which define the controlling computation (lines L33-L39). Note how the use of general critical sections allows us to reduce the number of edges that need to connect two consecutive scc's as compared to the simple algorithm that would be defined by the proof of Theorem 8.

The main while loop of the algorithm executes p times in the worst case, where p is the number of critical sections in the computation. Each iteration takes $O(n^2)$, since it must compute the scc's. Thus, a simple implementation of the algorithm will have a time complexity of $O(n^2p)$. However, a better implementation of the algorithm would amortize the cost of computing scc's over multiple iterations of the loop. Each iteration would compare each of the critical sections that have newly reached the heads of the lists with the existing critical sections. Therefore, each of the p critical section reaches the head of the list just once, when it is compared with $n-1$ critical sections to determine. The time complexity of the algorithm with this improved implementation is, therefore, $O(np)$. Note that a naive algorithm based directly on the constructive proof of the sufficient condition in Theorem 8 would take $O(p^2)$. We have reduced the complexity significantly by using the fact that the critical sections in a process are totally ordered.

Finally, we prove the correctness of the algorithm.

Lemma 18 *The algorithm in Figure 3.6 is well-specified.*

Proof: To prove that the algorithm is well-specified, the only non-trivial check is for the term $curr.next.first$ at line L37. To show that $curr.next \neq null$ at this point, we prove that, after the termination of loop L13-L19, for each strongly connected component (scc) s in *chain* : $(\exists c \in s : \ c.next = null) \Rightarrow (s = chain.head)$. Let s be an scc in *chain* and let $c \in s$ be a critical section such that $c.next = null$. Consider the point when s is about to be added to *chain* in line L27. Since $s \in valid\_scc$, we have $\forall s' \in scc\_set : \ (s' \neq s) \Rightarrow (s' \not\hookrightarrow s)$ However, since $c.next = null$, we have $\forall c' \in \mathcal{C} : \ c' \mapsto c$. Therefore, $\forall s' \in scc\_set : \ s' \hookrightarrow s$. Therefore, s is the only element of *valid_scc* at this point. It follows that after line L29, $CS = \emptyset$ and the loop terminates. Therefore, after the termination of loop L19-32, $s = chain.head$.

We next show that the algorithm terminates. The loop L19-L32 terminates since in each iteration either one I_i advances over a critical section (line L29) or two distinct critical sections merge into one (line L32). The loop L34-L38 terminates since *chain* decreases by one critical section in each iteration. □

Lemma 19 *If the algorithm in Figure 3.6 exits at line* L26*, then there is a non-trivial cycle of critical sections with:*

(1) at least one write-critical, or

(2) two k-critical sections for some $k \in \{1, \cdots, m\}$.

Proof: Follows directly from the condition at line L25. □

Lemma 20 *If the algorithm in Figure 3.6 terminates normally, then* $\langle E, \rightarrow^c \rangle$ *is a controlling computation of* ϕ_{gen_mutex} *in* $\langle E, \rightarrow \rangle$*, where* $\rightarrow^c$ *is the transitive closure of the union of* $\rightarrow$ *with the set of edges in* $\mathcal{O}$.

Proof:

At any point in the algorithm, let $\mathcal{C}$ denote the set of regular (not general) critical sections in the input computation. Note that $\mathcal{C}$ is a constant throughout the algorithm execution. Let $\mathcal{S}$ be the set of strongly connected components (scc) in the graph $\langle \mathcal{C}, \mapsto \rangle$. We use the relations $\mapsto$ and $\hookrightarrow$ for both regular and general intervals and scc's since the usage is clear from context. Let $\mathcal{C}_\mathcal{G}$ denote the set of general critical sections in $\mathcal{I} = \bigcup_i \mathcal{I}_i$ at any stage of the algorithm (note that $\mathcal{C}_\mathcal{G}$ is a variable since it changes at the *merge* step during the algorithm).

Let *scc* be a function that maps each critical section in $\mathcal{C}$ to the strongly connected component in $\mathcal{S}$ that contains it. First, we prove that the following are invariants.

INV1: $\forall g \in \mathcal{C}_\mathcal{G} : \ \forall x, y \in g.set : \ scc(x) = scc(y)$

INV2: g is reachable from g' in the graph $\langle \mathcal{C}_\mathcal{G}, \mapsto \rangle \ \Rightarrow$

$\forall x \in g.set, x' \in g'.set : \ x$ is reachable from x' in the graph $\langle \mathcal{C}, \mapsto \rangle$

It is easy to prove that INV2 is a consequence of INV1 (by induction on the length of the reachability path). Therefore, we have only to prove that INV1 is an invariant. We do so by structural induction.

Initially, each general critical section in $\mathcal{C}_\mathcal{G}$ contains only one critical section and so INV1 holds. The only step at which $\mathcal{C}_\mathcal{G}$ changes is the *merge* at line L32. Consider the execution point just before line L32. Since $c \in mergeable$ we have:

c.prev.prev $\in$ *candidate*, and –[A]

let $c' \in$ *candidate* such that $c \mapsto c'$. –[B]

Since *candidate* is an scc of the subgraph $\langle CS, \mapsto \rangle$ of $\langle \mathcal{C}_\mathcal{G}, \mapsto \rangle$, c' and *c.prev.prev* are in the same scc, say s, of $\langle \mathcal{C}_\mathcal{G}, \mapsto \rangle$. Clearly, *c.prev.prev* $\mapsto c$. Together with [B], this implies that c is also in s. Therefore:

c and *c.prev.prev* are reachable from each other in graph $\langle \mathcal{C}_\mathcal{G}, \mapsto \rangle$. –[C]

Since *candidate* $\subseteq CS$, using [A], we have *c.prev.prev* $= CS_{c.proc}$. Let x and y be any two critical sections in *g.set*, where g is the general critical section formed by merging *c.prev.prev* with c. By the inductive hypothesis, the case when both x and y belong to c or to *c.prev.prev* is trivial. Therefore, without loss of generality, assume that $x \in c$ and $y \in$ *c.prev.prev*. By the inductive hypothesis, and since INV2 follows from INV1, we use INV2 and [C] to give: x and y are reachable from each other in graph $\langle \mathcal{C}, \mapsto \rangle$. Therefore, $scc(x) = scc(y)$.

Now, consider the point of execution just before line L27 when *candidate* is added to *chain*. Let *chain.set* be the set of general critical sections in the scc's of the sequence *chain*. It is easy to show that the following is an invariant:

INV3: $\forall c \in \mathcal{C}_{\mathcal{G}} : \ c \prec CS_{c.proc} \Rightarrow c \in chain.set$

Suppose there is a general critical section c in $(\mathcal{C}_{\mathcal{G}} - (candidate \cup chain.set))$ such that for some $c' \in candidate$, $c \mapsto c'$. Consider two cases:

Case 1: $CS_{c.proc} \in candidate$**:** Since by definition, $c \notin candidate$, we have $c \neq CS_{c.proc}$. Using INV3, we have $CS_{c.proc} \prec c$ and, since c is a critical section $CS_{c.proc}.next.next \preceq c$. Together with $c \mapsto c'$, this gives $CS_{c.proc}.next.next \mapsto c'$. This contradicts the fact that $mergeable = \emptyset$ just before line L27 (by the check at L25).

Case 2: $CS_{c.proc} \notin candidate$**:** Using INV3, we have $CS_{c.proc} \preceq c$. Since $c \mapsto c'$, we have $CS_{c.proc} \mapsto c'$. Let $s \in scc\_set$ such that $CS_{c.proc} \in s$. Therefore $s \hookrightarrow candidate$ and $s \neq candidate$. This contradicts the fact that *candidate* was selected from *valid_scc* (lines L21, L22).

Therefore, $\forall c \in (\mathcal{C}_{\mathcal{G}} - (candidate \cup chain.set)) : \ \forall c' \in candidate : \ c \not\mapsto c'$ –[D]

This proves that *candidate* is an scc in $(\mathcal{C}_{\mathcal{G}} - chain.set)$ and, furthermore, that it is a minimal scc in $(\mathcal{C}_{\mathcal{G}} - chain.set)$.

Let $reg(candidate)$ be the set of regular critical sections in *candidate* (i.e. $\{x \in \mathcal{C} \mid \exists g \in candidate : x \in g.set\}$). Similarly, let $reg(chain.set)$ be the set of regular critical sections in *chain.set*. By the definition of *ord* on $\mathcal{C}_{\mathcal{G}}$, we have: $\forall c, c' \in \mathcal{C}_{\mathcal{G}} : \ (\exists x \in c, x' \in c' : \ x \mapsto x') \Rightarrow (c \mapsto c')$. Therefore, by [D], we have:

$\forall x \in (\mathcal{C} - (reg(candidate) \cup reg(chain.set))) : \ \forall x' \in reg(candidate) : \ x \not\mapsto x'$ –[E]

By INV2, we know that all critical sections in $reg(candidate)$ are reachable from one another. Therefore, using [E], $reg(candidate)$ is an scc in $\langle \mathcal{C} - reg(chain.set), \mapsto \rangle$. Further, using [E] again, $reg(candidate)$ is a minimal scc in $\langle \mathcal{C} - reg(chain.set), \mapsto \rangle$.

Since the minimal scc is chosen at each step, *chain* represents a topological sort of the graph $\langle \mathcal{S}, \hookrightarrow \rangle$. Therefore, from the construction of $\mathcal{O}$ from *chain* in lines L33-L38 of the algorithm, the $\rightarrow^c$ relation is the same as that defined in the proof of Theorem 8 (note that the optimization of connecting general critical sections by the $\leadsto^c$ edges instead of regular critical sections, as in the proof of Theorem 8, does not affect the equality of $\rightarrow^c$ defined by each). Therefore, in a very similar way to the proof of Theorem 8, we can prove that $\langle E, \rightarrow^c \rangle$ is a controlling computation of ϕ_{gen_mutex} in $\langle E, \rightarrow \rangle$. $\Box$

Theorem 11 *The algorithm in Figure 3.6 solves the predicate control problem for the generalized mutual exclusion predicate if the input computation has no non-trivial cycles of critical sections containing two k-critical sections for some $k \in \{1, \cdots, m\}$.*

Proof: By Lemma 18, the algorithm always terminates, and by Lemma 20, if it terminates normally then it outputs a controlling computation. By Lemma 19, if it terminates at line L26, then there is a non-trivial cycle with:

(1) at least one write-critical, or

(2) two k-critical sections for some $k \in \{1, \cdots, m\}$.

In case (1), Theorem 7 indicates that no controlling computation exists. Therefore, it is only in case (2) that the algorithm fails to find a controlling computation that does exist, (which is to be expected since the problem is NP-Complete by Theorem 9). $\Box$

Chapter 4

The Predicate Detection Problem

In this chapter, we present our study of predicate detection in the extended model of computation.

4.1 Overview

The extended model of computations [ACG93] extends the happened before model by allowing partially ordered events within a process. While this seems like a small difference, it has broad implications. In Section 4.2, we give the background for the extended model of distributed computations and discuss why the extended model should be preferred to the happened before model for certain applications.

We focus on the important class of "conjunctive predicates" which can be solved efficiently in the happened-before model [GW94]. Conjunctive predicates are specified as conjunctions of local predicates and express the combined occurrence of local events. Some examples of these predicates are: "all servers are unavailable" and "no process has a token". In Section 4.3, we formally state the problem of conjunctive predicate detection (predicate detection for conjunctive predicates).

Unfortunately, our first result is that the problem is NP-Complete in general. We prove this in Section 4.4. This is an indication of the difficulty of solving predicate detection in the extended model. However, this is to be expected since a general computation (in the extended model) represents, in general, an exponential number of happened before computations. We call this set of happened before computations as the set of "local linearizations" of the general computations. In Section 4.5, we demonstrate the equivalence between detecting a predicate in a general computation and detecting it in the corresponding set of local linearizations.

The next results in Section 4.6 show that it is indeed possible to find an efficient algorithm in the special case of "receive-ordered" and "send-ordered" computations. A receive ordered computation has totally ordered receive events, while other events may be partially ordered (and similarly for send-ordered computations). These computations are important since some natural programming styles create computations that fall in these categories. Note that obtaining efficient algorithms under such constraints is a big improvement over the alternative method of detection in an exponential-sized set of local linearizations.

Finally, in Section 4.7, we deal with general computations under no restrictions. Although, the problem is NP-Complete, we show that the algorithms of the previous section can be utilized by first decomposing a general computation into a set of receive-ordered or send-ordered computations. Though exponential in time, by restricting the search to the set of receive-ordered or send-ordered computations, this approach is still a great improvement over the alternative method of detection in the set of all local linearizations.

4.2 A Case for the Extended Model

The *happened before model* [Lam78] has been used to model distributed computations, capturing the notions of *logical time* and *potential causality*. As a result, these two notions have long been considered the same. In this section, we argue that these two notions are different and, in fact, starkly contradictory in nature. They arise in different applications and require different models. Although happened before suffices to model logical time, it is not good for modeling potential causality.

Historical Perspective

The history of modeling distributed computations may be divided into three stages. The first stage started with Lamport's introduction of happened before to model logical time [Lam78]. The second started with Mattern's observation that happened before may also be used to model potential causality [Mat89]. The third stage consisted of the gradual discovery that using happened before to model potential causality leads to problems owing to inherent false causality [CS93, SBN$^+$97] and the definition of extended models of computation [ACG93, HW96].

Stage 1: Happened Before and Logical Time

Many applications, such as mutual exclusion and deterministic replay, need to know the order in which events happen in time. In a distributed system, events on different processes do not share a common clock. This makes it impossible to determine their order in real time using time-stamping mechanisms. Lamport [Lam78] introduced logical time to order distributed events in a manner that approximates their real time order.

To model logical time, Lamport defined the *happened before* relation, denoted by $\rightarrow$, as the smallest relation satisfying the following: (1) If a and b are events in the same process, and a comes before b according to the total ordering specified by the local clock on that process, then $a \rightarrow b$. (2) If a is the send event of a message 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 $a \rightarrow c$. Further, two events a and b are *concurrent*, denoted by $a \| b$, if and only if they are incomparable using the happened before relation (i.e. $(a \not\rightarrow b) \ \wedge \ (b \not\rightarrow a)$).

Lamport went on to define a logical clock mechanism which constructed a total order of events that is a linearization of the happened before relation. This total order may be viewed as a possible ordering of events in real time and is sufficient for applications that need a notion of logical time.

Stage 2: Happened Before and Potential Causality

If an event happens before another event, it has the potential for causing that event. Many applications, such as recovery and debugging, require the tracking of such causal dependencies. Mattern [Mat89] realized that such applications would benefit by a mechanism to quickly determine the happened before relation between events.

The totally ordered logical clock mechanism that proved useful for applications requiring a notion of logical time is not good for applications requiring the notion of potential causality. In Mattern's own words: "For some applications (like mutual exclusion as described by Lamport himself in [Lam78]) this defect is not noticeable. For other purposes (e.g., distributed debugging), however, this is an important defect." Mattern, therefore, proposed a vector clock mechanism that allows the happened before relation between events to be deduced.

Stage 3: False Causality Problems and Extended Models

An event that happens before another event need not necessarily cause it. This is implicit when we say that happened before tracks *potential* causality. Therefore, an inherent problem in using happened before in applications that require causality tracking is that sometimes events that are independent are believed to have a causal dependency. This phenomenon is called *false causality*. While any approximation of causality must have false causality, the happened before model was found to fall particularly short in this respect. We cite three examples of application domains where this has happened.

Firstly, happened before has been used as the basis of causally and totally ordered communication support. Cheriton and Skeen [CS93] observed that when two send events have a false causal dependency between them, the resulting effect is to make the receipt of one message unnecessarily wait for the receipt of the other. This overhead was mentioned as one of the limitations of causally and totally ordered communication support.

Secondly, happened before has been used in tools to detect data races in multi-threaded programs. Savage, et al [SBN+97] pointed out that false causality between events causes some data races to go undetected. This is because if one event happens before another, it may falsely be believed to cause that event and thus a potential data race between the independent events may be missed.

The third application domain is predicate detection, the focus of this chapter. As we noted in Chapter 1, using the happened before model would miss the detection of certain predicates owing to the false causality between events. Since predicate detection has applications in distributed debugging and distributed monitoring, this translates to certain faulty conditions being missed.

The incidence of the false causality problem in a number of applications was noted by Ahuja, et al [ACG93]. They proposed a *passive-space and time* view of a computation that partially orders the events within a process. They also propose a more general vector clock mechanism than the one proposed by Mattern. These vector clocks trade the cost incurred and the accuracy of causality identification.

In a later study, Hrischuk and Woodside [HW96] also proposed more refined approximations of causality than the happened before model. A distinguishing feature of their *scenario causality* model is the use of typed nodes and typed edges in the computation graph.

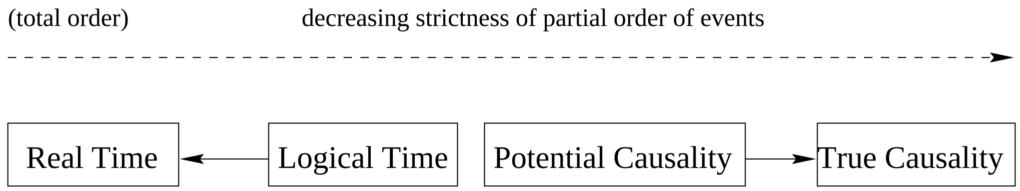

Figure 4.1: Spectrum of models

Logical Time is *not* Potential Causality

We have seen that happened before has come to be used to model both logical time and potential causality in distributed computations. In spite of demonstrations to the contrary [ACG93, HW96], this has led to a wide-spread belief that the two concepts are the same. We now present a brief argument that the concepts are not only different but opposite in nature.

The applications that motivated Lamport to model logical time were notably different from those that motivated Mattern to model potential causality. The first kind of applications, represented by mutual exclusion, require as close an approximation to real time order as possible. Happened before is, in fact, the *best* possible approximation of real time order that one can make in a message-passing distributed system without external channels and global clocks. The kind of applications that require potential causality, represented by distributed recovery, require as close an approximation as possible to the true causality between events.

As illustrated in Figure 4.1, real time and true causality fall at opposite ends of a spectrum of partial orders based on their strictness. While logical time tries to approximate the real time total order that lies on one end of the spectrum, potential causality tries to approximate true causality at the other end. Happened before was initially introduced to model logical time and therefore closely approximates real time. By a historical accident, it was also used to model potential causality. In fact, in attempting to approach real time, the happened before model creates more false causality than necessary. A better model for potential causality would be a less strict partial order that approaches true causality in the spectrum indicated in Figure 4.1. In both of the studies [ACG93, HW96] on modeling potential causality, the authors describe a spectrum of choices for models that approximate true causality. In [ACG93], flexible models are proposed that allow trading off the cost with the accuracy of the approximation.

Closing Comments

A computation, in the sense that we have used it, is something that has actually happened. It may, therefore, seem peculiar that a single potential causality model may capture multiple possible interleavings of local events (or, multiple possible happened before computations). The same fact may be noticed in the happened before model as compared to the real time ordering model. The happened before model allows multiple possible real time orderings. In both cases, the explanation for this seeming paradox is that *what actually happened lies in the eyes of the beholder*. If the observer is interested in applications that require logical time, he would pay attention to the ordering of local events based on real time. However, if the observer were interested in applications that require potential causality, he would only pay attention to the causal ordering of local events.

The crucial deciding factor in choosing the happened before or the potential causality model is the type of application being considered. A litmus test for making this decision would be to ask the following question: *Would a global clock help?* A "yes" would indicate happened before, while a "no" would indicate potential causality. For example, in mutual exclusion, a global clock would allow the timestamping of critical section requests in real time and lead to a fair granting order. In fact, even without a global clock, if there were multiple threads in a process, we would prefer to treat local events as being totally ordered using the local process clock. In an application like race detection, for example, a global clock would have no advantage because any more ordering would merely induce more false causality and reduce our chances of detecting a race.

The purpose of this section has been to draw attention to the false causality problem which limits the effectiveness of the happened before model in modeling potential causality. A more effective model of potential causality allows events within a process to be partially ordered. We now present our study of the predicate detection problem in this *extended model* of computation.

4.3 Problem Statement: Conjunctive Predicate Detection

In this chapter, we consider *general* computations corresponding to the *extended* model of computation. All of the definitions in our model (Chapter 2) apply to general computations, except for those in Sections 2.5, 2.6, and 2.7. The definition of the predicate detection problem in this model is:

The Predicate Detection Problem: Given a general computation $\langle E, \rightarrow \rangle$ and a global predicate ϕ, is there a consistent cut C such that $\phi(C)$?

We focus on the class of conjunctive predicates defined as:

- $\langle$**conjunctive predicates**$\rangle$ **:** Given n local predicates $\alpha_1, \alpha_2, \cdots, \alpha_n$, the *conjunctive predicate* ϕ_{conj} is defined as:

$$\phi_{conj}(C) \equiv \bigwedge_{i \in \{1, \cdots, n\}} \alpha_i(C[i])$$

The reason that conjunctive predicates are important is that their detection is sufficient for the detection of *any* global predicate which can be written as a boolean expression of local predicates [Gar96]. This is true since the boolean expression can be expressed in disjunctive normal form giving rise to the disjunction of conjunctive predicates. Each of the conjunctive predicates can then be detected independently.

For example, for a distributed application with x, y, and z being variables on three processes, the predicate:

$$even(x) \wedge ((y < 0) \vee (z > 6))$$

can be rewritten as:

$$(even(x) \wedge (y < 0)) \vee (even(x) \wedge (z > 6))$$

Each of the disjuncts is a conjunctive predicate. It can also be shown that, even if the global predicate is not a boolean expression of local predicates, but is satisfied by a finite number of consistent cuts, then it can also be expressed as a disjunction of conjunctive predicates.

Note that the reason that we do not solve the predicate detection problem for disjunctive predicates (defined in the Chapter 3) is that it is trivial to do so. One has simply to scan each process to see if the corresponding local predicate is true for any event. If so, the disjunctive predicate is detected. It is interesting to note that the negation of a conjunctive predicate is a disjunctive predicate. As a result, if a conjunctive predicate indicates a failure, then it is its negation, a disjunctive predicate, which must be given as input to the predicate control problem to maintain. Thus, the detection of conjunctive predicates is complementary to the control of disjunctive predicates.

The problem of conjunctive predicate detection was efficiently (in $O(mn^2)$ time, where m is a bound on the number of events in a process) and optimally solved for locally ordered computations in [GW94]. However, in the extended model, the problem becomes expectedly harder as we establish in the following section.

4.4 Conjunctive Predicate Detection is NP-Complete

Let CPG denote the predicate detection problem for the Conjunctive Predicate in General computations.

Theorem 12 *CPG is NP-Complete.*

Proof: CPG is in NP because, given a cut C, $\phi_{conj}(C)$ can be checked in polynomial time (by the definition of predicates) and a naive check for consistency based directly on its definition can be checked in polynomial time.

To show that CPG is NP-Hard, we transform 3SAT. Let $U = \{u_1, u_2, \cdots u_w\}$ be a set of variables, and $D = \{c_1, c_2, \cdots c_x\}$ be a set of clauses, forming an instance of 3SAT. Let l_{i1}, l_{i2}, l_{i3} be the three literals in clause c_i.

For each variable u_r, define a set of events $E'_r = \{e'_{r1}, e'_{r2}\}$. For each clause c_i, define a set of events $E_i = \{e_{i11}, e_{i21}, e_{i31}, e_{i12}, e_{i22}, e_{i32}\}$. Let $E = \bigcup_i E_i \cup \bigcup_r E'_r$. On each E_i, define a relation $\prec_i = \{(e_{i11}, e_{i12}), (e_{i21}, e_{i22}), (e_{i31}, e_{i32})\}$. let $\prec = \bigcup_i \prec_i$. Define $\rightsquigarrow$ as the smallest relation on E such that:

- for each literal $l_{ip} = u_r$ for some variable u_r: $e_{ip2} \rightsquigarrow e'_{r2}$.

- for each literal $l_{ip} = \overline{u_r}$ for some variable u_r: $e_{ip2} \rightsquigarrow e'_{r1}$.

Let $\rightarrow$ be the transitive closure of $\prec \cup \rightsquigarrow$. It is easy to verify that the graph $\langle E, \prec \cup \rightsquigarrow\rangle$ has no cycles. Therefore, $\langle E, \rightarrow\rangle$ is a computation.

Define local predicates $\alpha_1, \cdots, \alpha_x, \alpha'_1, \cdots, \alpha'_w$ such that:

- for each E_i: $\alpha_i(\perp_i) = \mathit{false}$, $\forall p : \alpha_i(e_{ip1}) = \mathit{true}$, and $\forall p : \alpha_i(e_{ip2}) = \mathit{false}$.

- for each E'_r: $\alpha'_r(\perp'_r) = \mathit{false}$, $\alpha'_r(e'_{r1}) = \mathit{true}$, and $\alpha'_r(e'_{r2}) = \mathit{false}$.

Define ϕ_{conj} based on these local predicates. Thus, we have constructed an instance of CPG, $\langle\langle E, \rightarrow\rangle, \phi_{conj}\rangle$, from an instance of 3SAT, $\langle U, D\rangle$. We next show that this is indeed a transformation.

Part 1: Suppose that t is a truth assignment assignment of variables in U that satisfies C. We prove that ϕ_{conj} is detected in $\langle E, \rightarrow\rangle$. Define a cut C as follows. For each E_i, let l_{ip} be a literal which is true under t (since t satisfies D) and let $C[i] = e_{ip1}$. For each E'_r, if $t(u_r) = true$ then $C[r'] = e'_{r1}$, and if $t(u_r) = false$ then $C[r'] = e'_{r2}$. Clearly, $\phi_{conj}(C)$. We next show by contradiction that C is consistent.

Suppose C is not consistent. Let e be an event in $C.past$ such that $C[e.proc] \prec e$. By the definition of $\prec$, we know that e must belong to some E_i corresponding to a clause c_i. Note that e cannot be a $\bot_i$ or a e_{ip1} since $C[e.proc] \prec e$. Therefore, suppose $e = e_{ip2}$ so that $C[i] = e_{ip1}$. Since $e_{ip2} \in C.past$, let e' be an event in C such that $e_{ip2} \rightarrow e'$. Consider two cases:

- **Case 1:** $l_{ip} = u_r$ for some variable u_r:

  By the definition of $\rightarrow$, we must have: $e' = e'_{r2}$. Since $C[i] = e_{ip1}$, l_{ip} is true under t. Therefore, $t(u_r) = true$ and so $C[r'] = e'_{r1}$. This contradicts the fact that $e' = e'_{r2}$ belongs to C.

- **Case 2:** $l_{ip} = \overline{u_r}$ for some variable u_r:

  By the definition of $\rightarrow$, we must have: $e' = e'_{r1}$. Since $C[i] = e_{ip1}$, l_{ip} is true under t. Therefore, $t(u_r) = false$ and so $C[r'] = e'_{r2}$. This contradicts the fact that $e' = e'_{r1}$ belongs to C.

Therefore, in both cases we get a contradiction. So, C is consistent.

Part 2: Suppose C is a consistent cut in $\langle E, \rightarrow \rangle$ such that $\phi_{conj}(C)$. Define a truth assignment t as follows: $t(u_r) = true$ if $C[r'] = e'_{r1}$ and $t(u_r) = false$ if $C[r'] = e'_{r2}$.

Consider a clause c_i. We know from the definition of local predicate α_i that $C[i] = e_{ip1}$ for some p. Consider two cases:

- **Case 1:** $l_{ip} = u_r$:

  Since $C[i] = e_{ip1}$ and $e_{ip1} \prec e_{ip2}$ and $e_{ip2} \leadsto e'_{r2}$, the consistency of C would be violated if $C[r'] = e'_{r2}$. Therefore, $C[r'] = e'_{r1}$. So, $t(u_r) = true$. Thus, literal l_{ip} is true under t, and so, clause c_i is satisfied by t.

- **Case 2:** $l_{ip} = \overline{u_r}$:

  Since $C[i] = e_{ip1}$ and $e_{ip1} \prec e_{ip2}$ and $e_{ip2} \leadsto e'_{r1}$, the consistency of C would be violated if $C[r'] = e'_{r1}$. Therefore, $C[r'] = e'_{r2}$. So, $t(u_r) = false$. Thus, literal l_{ip} is true under t, and so, clause c_i is satisfied by t.

Thus t satisfies C. $\Box$

4.5 Local Linearizations

In practice, a computation is formed by the transitive closure of a local ordering relation and a remote ordering relation. We now make these relations explicit.

- $\langle (x)^+ \rangle$ **:** The notation $(x)^+$ denotes the transitive closure of a relation x.
- $\langle$**locally precedes, remotely precedes**$\rangle$ **:** We assume that a computation $\langle E, \rightarrow \rangle$ has associated with it two relations *locally precedes* ($<$) and *remotely precedes* ($\rightsquigarrow$) such that:
  - $<$ is an irreflexive partial order on E such that $e < f \;\Rightarrow\; e.proc = f.proc$.
  - $\rightsquigarrow$ is an irreflexive partial order on E such that $e \rightsquigarrow f \;\Rightarrow\; e.proc \neq f.proc$.
  - $\rightarrow \;=\; (< \;\cup \rightsquigarrow)^+$. Although not required, it is expected that $<$ and $\rightsquigarrow$ do not contain transitively redundant edges.
- $\langle\langle E, \rightarrow, <, \rightsquigarrow \rangle\rangle$ **:** When the distinction is important, we will use the notation $\langle E, \rightarrow, <, \rightsquigarrow \rangle$ to denote a computation $\langle E, \rightarrow \rangle$ with locally precedes relation $<$ and remotely precedes relation $\rightsquigarrow$.
- $\langle$**$<$ (extended)**$\rangle$ **:** We extend $<$ to $E \;\cup\; \bot$ such that $\forall e \in E_i : \;\bot_i <_i e$.

We next define the concept of a "local linearization" of a general computation. Informally, a local linearization is a locally ordered computation that can be obtained by linearizing the partially ordered processes in a general computation. In general, a general computation has an exponential number of local linearizations.

- $\langle$**linearization**$\rangle$ **:** A *linearization* of a partial order is a total order that contains it.

- $\langle$**local linearization**$\rangle$ **:** A *local linearization* of a (general) computation $\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle$ is a locally ordered computation $\langle E, \rightarrow, <, \rightsquigarrow \rangle$ such that $\forall i : <_i$ is a linearization of ${<^s}_i$. (Note: an arbitrary choice of linearizations of ${<^s}_i$'s may not result in a computation because of possible cycles).

- $\langle Lin(\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle)\rangle$ **:** Let $Lin(\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle)$ represent the set of local linearizations of a computation $\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle$

General computations concisely represents many locally ordered computations, thus allowing improved detection. The following result confirms that solving predicate detection in a general computation is equivalent to solving predicate detection for all of the computation's local linearizations. First, we define the new problem:

The Conjunctive Predicate Detection Problem in Local Linearizations (CPL): Given a general computation $\langle E, \rightarrow \rangle$, is there a consistent cut C in any local linearization of $\langle E, \rightarrow \rangle$ such that $\phi_{conj}(C)$?

Theorem 13 *CPG is equivalent to CPL.*

Proof: Let the general computation $\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle$ and the conjunctive predicate ϕ_{conj} define instances of CPG and CPL.

Part 1: Suppose the cut C is a solution to CPL. Then, it follows as a corollary of Lemma 2 that C is consistent in $\rightarrow^s$, and so, also a solution of CPG.

Part 2: Suppose the cut C is a solution to CPG. We prove that C is also a solution to CPL. We construct $\langle E, \rightarrow^s, <, \rightsquigarrow \rangle$, a local linearization of $\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle$ as follows. If each $\langle E_i, {\rightarrow^s}_i \rangle$ is already a total order, then we simply let $<_i = {\rightarrow^s}_i$ and we are done. So assume that two events e and f of the same process are incomparable in $\rightarrow^s$. From $\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle$, we construct another general computation, $\langle E, \rightarrow^{s\prime}, <^{s\prime}, \rightsquigarrow \rangle$ such that:

(1) $<^s \subseteq <^{s\prime}$,

(2) C is consistent in $<^{s\prime}$, and

(3) e and f are comparable in $\rightarrow^{s\prime}$.

By repeating this procedure, we will eventually obtain a local linearization of $\langle E, \rightarrow^s, <^s, \rightsquigarrow\rangle$ in which C is consistent, thus proving the result. We make e and f comparable in $\rightarrow^{s\prime}$ as follows:

- **Case 1:** $e \rightarrow^s C[k]$ for some k:

  We add (e, f) to $<^s$ giving $<^{s\prime}$. Clearly, no cycles are caused, since e and f were incomparable in $\rightarrow^s$. We show that C is consistent in $\rightarrow^{s\prime}$. If not, then there must exist an event g in the causal past (w.r.t. $\rightarrow^{s\prime}$) of C such that $C[g.proc] \prec' g$ (where $\prec'$ is defined with respect to $\rightarrow^{s\prime}$). Since C is consistent in $\rightarrow^s$, it cannot be that both of the following are true: g is in the causal past (w.r.t. $\rightarrow^s$) of C and $C[g.proc] \prec g$. Therefore, we have two cases:

  - **Case a:** g is not in the causal past (w.r.t. $\rightarrow^s$) of C and $C[g.proc] \prec g$:

    Since g is in the causal past (w.r.t. $\rightarrow^{s\prime}$) of C, we must have $g \rightarrow^s e$ and $f \rightarrow^s h$ for some $h \in C$. Therefore, by transitivity, $g \rightarrow^s C[k]$. Since $C[g.proc] \prec g$, this contradicts the consistency of C in $\rightarrow^s$.

  - **Case b:** $C[g.proc] \not\prec g$:

    Since $C[g.proc] \prec' g$, we must have $C[g.proc] \prec e$ and $f \prec g$. Since $e \rightarrow^s C[k]$, this contradicts the consistency of C in $\rightarrow^s$.

- **Case 2:** $e \not\rightarrow^s C[k]$ for all k:

  We add (f, e) to $<^s$ giving $<^{s\prime}$. Clearly, no cycles are caused, since e and f were incomparable in $\rightarrow^s$. We show that C is consistent in $\rightarrow^{s\prime}$. If not, then there must exist an event g in the causal past (w.r.t. $\rightarrow^{s\prime}$) of C such that $C[g.proc] \prec' g$ (where $\prec'$ is defined with respect to $\rightarrow^{s\prime}$). Since C is consistent in $\rightarrow^s$, it cannot be that both of the following are true: g is in the causal past (w.r.t. $\rightarrow^s$) of C and $C[g.proc] \prec g$. Therefore, we have two cases:

– **Case a:** g is not in the causal past (w.r.t. $\rightarrow^s$) of C:

Since g is in the causal past (w.r.t. $\rightarrow^{s\prime}$) of C, we must have $g \rightarrow^s f$ and $e \rightarrow^s h$ for some $h \in C$. This contradicts the condition that $e \not\rightarrow^s C[k]$ for all k.

– **Case b:** g is in the causal past (w.r.t. $\rightarrow^s$) of C and $C[g.proc] \not\prec g$:

Since $C[g.proc] \prec' g$, we must have $C[g.proc] \prec f$ and $e \prec g$. Thus, by transitivity, e is in the causal past (w.r.t $\rightarrow^s$) of C. This contradicts the condition that $e \not\rightarrow^s C[k]$ for all k.

□

4.6 Solving Conjunctive Predicate Detection Under Constraints

The result of the previous section tells us that if we exhaustively detect a predicate in each of $Lin(\langle E, \rightarrow^s, <^s, \rightsquigarrow\rangle)$ then we have also done so for the computation $\langle E, \rightarrow^s, <^s, \rightsquigarrow\rangle$. Since this would be very inefficient, we identify two classes of general computations for which we may apply a special predicate detection algorithm to a specially chosen representative from $Lin(\langle E, \rightarrow^s, <^s, \rightsquigarrow\rangle)$ in order to efficiently detect a predicate.

- $\langle$**send/receive events**$\rangle$ **:** In a general computation $\langle E, \rightarrow^s, <^s, \rightsquigarrow\rangle$, if $s \rightsquigarrow t$, then we call s a *send* event and we call t a *receive* event.

- $\langle Snd/Rcv\rangle$ **:** Let Snd be the set of send events and Rcv be the set of receive events in E.

- $\langle Snd_i/Rcv_i\rangle$ **:** Snd_i and Rcv_i denote the sets of send and receive events, respectively, in E_i.

- $\langle$**receive-ordered computation**$\rangle$ **:** A computation $\langle E, \rightarrow^s, <^s, \rightsquigarrow\rangle$ is *receive-ordered* if $\forall i :\ Rcv_i$ is totally ordered under $({<^s}_i)^+$.

- $\langle$**send-ordered computation**$\rangle$: A computation $\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle$ is *send-ordered* if $\forall i$: Snd_i is totally ordered under $({<^s}_i)^+$.

The constraint of being send-ordered or receive-ordered is sometimes natural for computations derived from common programming styles. For example, a scenario that arises very often, especially in client-server systems, is:

```
repeat
   receive a request ;
   create a thread to process the request
until done
```

It is clear that such a scenario is receive-ordered even though the sends and per-request processing may be independent.

Another scenario that is often used to model synchronous rounds is:

```
repeat
   receive and process messages
               until time = end-of-round ;
   send messages
until done
```

If the sends in a round occur in a fixed order or use the same port, then they are totally ordered while receives and local processing may be independent. Thus, a distributed computation resulting from such a program would be send-ordered.

Let CPR and CPS denote the CPG problem specialized to receive-ordered and send-ordered computations respectively.

From the set of local linearizations of a given computation, we pick a special representative that satisfies the following property:

- $\langle$**P1**$\rangle$ **:** Let $\langle E, \rightarrow, <, \leadsto \rangle$ be a local linearization of $\langle E, \rightarrow^s, <^s, \leadsto \rangle$. Then let **P1** be the following property.

$$\textbf{P1:}\ \ \forall i:\ \forall e \in E_i:\ \forall f \in Rcv_i:\ (f\ (<)^+\ e)\ \Rightarrow\ (f \rightarrow^s e)$$

This ensures that we linearize the partial order $<^s$ on each process such that a receive event is ordered after all the events that are concurrent with it. The property is well-defined because no two receive events are concurrent.

In order to ensure this property, we apply a special linearization algorithm, RECEIVE-SORT shown in Figure 4.2, for each process. The algorithm is a modification of a standard topological sort algorithm that gives a higher priority to non-receive events so that all events concurrent to a receive event precede it in the total ordering. The algorithm applies this modified topological sort to the graph $\langle E_i, {<^s}_i \rangle$. It is easy to show that this correctly produces a linearization of the partial order for each process. It is also easy to show that the linearizations produced by the algorithm form a local linearization that satisfies Property P1:

Lemma 21 *Let $\langle E, \rightarrow^s, <^s, \leadsto \rangle$ be a receive-ordered computation and let $<= \bigcup_i <_i$, where $<_i$ is the linearization of ${<^s}_i$ produced by applying algorithm RECEIVE_SORT. Then, $\langle E, \rightarrow, <, \leadsto \rangle$ is a local linearization of $\langle E, \rightarrow^s, <^s, \leadsto \rangle$ satisfying* **P1**.

Proof: We first show that the graph of $\langle E, \rightarrow, <, \leadsto \rangle$ satisfies **P1**. We next show that the graph $\langle E, < \cup \leadsto \rangle$ has no cycles and, therefore, $\langle E, \rightarrow, <, \leadsto \rangle$ is a computation.

Input:

E_i	set of events
Rcv_i	set of receive events in E_i

Output:

$\mathcal{Q}_i$	queue of events in E_i, initially $\emptyset$

Notation:

$select(Z)$	arbitrary element from non-empty set Z

Variables:

$F = E_i$	set of events
$M = \emptyset$	set of events
$R = \emptyset$	set of events
e, f	events
$k[E_i]$	array of integers ranging over events in E_i

Algorithm:

```
for each event e in E_i do
    k[e] := no. of incoming edges in <^s_i for e
    if (k[e] = 0) then
        if (e ∈ Rcv_i) then R := R ∪ {e}
        else M := M ∪ {e}
while (F ≠ ∅) do
    if (M ≠ ∅) then f := select(M)
    else f := select(R)
    enqueue(Q_i, f)
    F := F − {f}
    for each event e ∈ F such that f <^s_i e do
        k[e] := k[e] − 1
        if (k[e] = 0) then
            if (e ∈ Rcv_i) then R := R ∪ {e}
            else M := M ∪ {e}
```

Figure 4.2: Algorithm RECEIVE-SORT

The following four invariants are maintained by the algorithm at line L9:

INV1: $\forall e \in F : \ k[e] =$ number of events that immediately precede e in $\langle F, {<^s}_i \rangle$.

INV2: M is the set of minimal events in $\langle F, {<^s}_i \rangle$ that are not in Rcv_i.

INV3: R is the set of minimal events in $\langle F, {<^s}_i \rangle$ that are in Rcv_i.

INV4: F contains no event in $\langle E_i, {<^s}_i \rangle$ that transitively precedes an event in $M \cup R$.

It is easy to prove INV1 from the algorithm, and then prove INV2 and INV3 using it. INV4 follows from INV1 by induction on the length of the transitive path.

To show that **P1** holds, let $e \in E_i$ and $f \in Rcv_i$ be two events such that $f \ (<)^+ \ e$. Since $\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle$ is receive-ordered and since $<$ is a linearization of $<^s$, the case when $e \in Rcv_i$ follows easily. So, consider the case of $e \notin Rcv_i$. Consider the iteration of the loop L6-L15 in which f was enqueued in $\mathcal{Q}_i$. Since $f \in Rcv_i$, f must have been chosen at line L8 (using INV2 and INV3). So $M = \emptyset$. Further, $R = \{f\}$ (using INV3, INV4, and that the computation is receive-ordered). We conclude that f is the *minimum* event in $\langle F, {<^s}_i \rangle$. Since $f \ (<)^+ \ e$, e has not been enqueued yet. Therefore, $f \ (<^s)^+ \ e$.

We next prove by contradiction that there are no cycles in the graph $\langle E, < \cup \rightsquigarrow \rangle$. Suppose there is such a cycle, $\mathcal{C}$. Clearly $\mathcal{C}$ cannot be within a single E_i. Therefore, there must be at least one receive event in $\mathcal{C}$. Let $\mathcal{R}$ be the set of receive events in $\mathcal{C}$. Since $\rightarrow^s$ is an irreflexive partial order ($\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle$ is a computation), let r be any maximal event in $\langle \mathcal{R}, \rightarrow^s \rangle$. Let $s \rightsquigarrow r'$ be the next $\rightsquigarrow$ edge in the cycle $\mathcal{C}$. Therefore, $r \ (<)^+ \ s$. Therefore, by **P1**, we have $r \rightarrow^s s$. Therefore, $r \rightarrow^s r'$ contradicting the choice of r as maximal. □

From Property P1, we derive the following useful property:

- $\langle$**P2**$\rangle$: Let $\langle E, \rightarrow, <, \rightsquigarrow \rangle$ be a local linearization of $\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle$. Then let **P2** be the following property.

P2: $\forall$ distinct i, j : $\forall e \in E_i$: $\forall f, g \in E_j$:

$$((e \rightarrow^s f) \ \wedge \ (f \ (\leq)^+ \ g)) \ \Rightarrow \ (e \rightarrow^s g)$$

Lemma 22 *Let $\langle E, \rightarrow, <, \rightsquigarrow \rangle$ be a local linearization of a receive-ordered computation $\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle$. Then* **P1** $\Rightarrow$ **P2**.

Proof: Let $i \neq j$ and $e \in E_i$ and $f, g \in E_j$ and let $e \rightarrow^s f$ and $f \ (\leq)^+ \ g$. Since $e \rightarrow^s f$ and $e.proc \neq f.proc$, there must be some receive event r in E_j such that $e \rightarrow^s r$ and $r \ (\leq^s)^+ \ f$. Since $<$ is a linearization of $<^s$, we have $r \ (\leq)^+ \ f$, and, therefore, $r \ (\leq)^+ \ g$. If $r = g$, we are done. If $r \ (<)^+ \ g$, **P1** implies that $r \rightarrow^s g$. Therefore, by transitivity, $e \rightarrow^s g$. $\Box$

We now apply algorithm PRED-DETECT in Figure 4.3 to the special representative local linearization chosen using algorithm RECEIVE-SORT.

Our final result shows that applying PRED-DETECT to the representative local linearization is sufficient for detecting the predicate in the general computation. The main idea of the algorithm is similar to that used in [GW94] to optimally solve the problem for locally ordered computations. We start with the lowest cut and move upwards. If, in a cut C, we find that for some i and j, $C[i].next \rightarrow^s C[j]$ then using Property P2, we are guaranteed that $C[i].next$ is ordered before (in $\rightarrow^s$) every event following $C[j]$ in the total order $<$. So $C[i]$ can never be part of a consistent cut, and can be safely discarded. If no such pair of events can be found, then the cut is consistent. The algorithm discards at least one event in each iteration, and so, must terminate.

Theorem 14 *Let $\langle E, \rightarrow, <, \rightsquigarrow \rangle$ be the local linearization of a receive-ordered computation $\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle$ produced by applying algorithm RECEIVE-SORT to each $\langle E_i, {<^s}_i \rangle$. Then applying algorithm PRED-DETECT to $\langle E, \rightarrow, <, \rightsquigarrow \rangle$ and ϕ_{conj} solves CPR for $\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle$ and ϕ_{conj}.*

Proof: We first show that if PRED-DETECT returns *detected* = *true* then there is a consistent cut in $\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle$ that satisfies ϕ_{conj}. It is easy to verify that the

Input:

$\mathcal{Q}_1, \mathcal{Q}_2, \ldots \mathcal{Q}_n$ event queues ($\mathcal{Q}_i$ contains events of $E_i \cup \{\bot_i\}$ in $<_i$ order)
$\alpha_1, \alpha_2, \ldots \alpha_n$ local predicates

Output:

detected boolean

Notation:

$all \; = \; \{1, 2, \ldots, n\}$

Variables:

$low, newlow$ subsets of all
k, l integers in all

Algorithm:

$low \;:=\; all$
while $((low \neq \emptyset) \;\wedge\; (\forall k : \; \neg \mathcal{Q}_k.empty))$ **do**
  $newlow \;:=\; \emptyset$
  for k in low **do**
    if $(\; \neg x_k(\mathcal{Q}_k.head) \;)$ **then**
      $newlow \;:=\; newlow \;\cup\; \{k\}$
    else
      for l in all **do**
        if $((\mathcal{Q}_k.head.next, \mathcal{Q}_l.head \in E) \;\wedge\; (\mathcal{Q}_k.head.next \rightarrow^s \mathcal{Q}_l.head) \;)$ **then**
          $newlow \;:=\; newlow \;\cup\; \{k\}$
        if $((\mathcal{Q}_l.head.next, \mathcal{Q}_k.head \in E) \;\wedge\; (\mathcal{Q}_l.head.next \rightarrow^s \mathcal{Q}_k.head) \;)$ **then**
          $newlow \;:=\; newlow \;\cup\; \{l\}$
  $low \;:=\; newlow$
  for k in low **do**
    $\mathcal{Q}_k.delete\_head$
$detected \;:=\; (\forall k : \; \neg \mathcal{Q}_k.empty)$

Figure 4.3: Algorithm PRED-DETECT

following invariants are maintained after line L13:

INV1: $\forall k \notin low: \ \alpha_k(\mathcal{Q}_k.head)$

INV2: $\forall k, l \notin low: \ (\mathcal{Q}_k.head.next, \mathcal{Q}_l.head \in E) \ \Rightarrow \ (\mathcal{Q}_k.head.next \not\rightarrow^s \mathcal{Q}_l.head)$

Since, on termination, $detected = true$, by the terminating condition at L2, we must have $low = \emptyset$ after line 13 of the final iteration. Consider the cut C such that $\forall i: \ C[i] = \mathcal{Q}_i.head$ after line 13 of the final iteration. By INV1, we know that C satisfies ϕ_{conj} and by INV2, we know that C is consistent (by Lemma 3) in $\langle E, \rightarrow, <, \rightsquigarrow \rangle$. Further, by Lemma 2, C is consistent in $\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle$.

Next, we show that if PRED-DETECT returns $detected = false$, then there is no consistent cut in $\langle E, \rightarrow^s, <^s, t \rangle$hat satisfies ϕ_{conj}. Suppose there is such a consistent cut, C. Since $detected = false$, some queue must be empty at termination. Therefore, at least one of the $C[i]$'s must have been deleted from its queue. Consider the first iteration in which a $C[i]$ was deleted. Since $\phi_{conj}(C)$, $C[i]$ could not have been added to *newlow* at L6. Therefore, $C[i]$ must have been added in either L10 or L12. in either case, $C[i].next \neq null$ and $C[i].next \rightarrow^s Q_j.head$ for some j. Since $C[i]$ is the first to be deleted, $Q_j.head (\leq)^+ C[j]$. By Lemmas 21 and 22, we know that **P2** holds. Therefore, $C[i].next \rightarrow^s C[j]$ contradicting the consistency of C (by Lemma 3). $\Box$

Having established that CPR can be solved efficiently, it remains to solve CPS. Consider a send-ordered computation $\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle$. Let $\langle E, \rightarrow^{s\prime}, <^{s\prime}, \rightsquigarrow' \rangle$ be the computation obtained by inverting all the edges in the computation so that $<^{s\prime} = \{(e, f) | (f, e) \in <^s\}$ and $\rightsquigarrow' = \{(e, f) | (f, e) \in \rightsquigarrow\}$. In this computation, the receive events become send events and vice-versa. Therefore, in order to solve CPS in a send-ordered computation $\langle E, \rightarrow^s, <^s, \rightsquigarrow \rangle$, we can solve CPR instead in the receive-ordered computation $\langle E, \rightarrow^{s\prime}, <^{s\prime}, \rightsquigarrow' \rangle$. Thus, the algorithms we described for CPR can be used to solve CPS.

If m is a bound on $|E_i|$ and e is the size of the $<^s$ relation, then we can deduce that the time complexity of applying RECEIVE-SORT to each process is $O(mn + e)$ and the time complexity of PRED-DETECT is $O(mn^2)$. So the time complexity to solve CPR or CPS is $O(mn^2 + e)$.

4.7 Solving Conjunctive Predicate Detection Without Constraints

Having efficiently solved CPR and CPS, we now take another look at the general problem CPG. We know that CPG is NP-Complete. So, a polynomial solution to CPG is unlikely. Two naive exponential solutions are possible. Let m be a bound on $|E_i|$.The first solution enlists every cut and checks if it is consistent, taking $O(m^n n^2)$ time. The second applies a predicate detection algorithm (such as in [GW94]) to every local linearization of the general computation, which takes $O(m^{mn} mn^2)$ time.

However, these solutions do not perform any better for general computations which are "close" to being send-ordered or receive-ordered. For example, in a general computation which has two possible linearizations of receive events in one process, we would expect not to have to pay the full price of the above naive solutions. We now provide a solution that degrades gracefully for computations that are close to being send-ordered or receive-ordered.

Let k_i be a bound on the number of linearizations of the $<^s$ relation restricted to the set of receive events in E_i. If we linearize for each process, we can construct a receive-ordered computation by adding the ordering of receive events imposed by the linearizations. For all such possible combinations of linearizations, there would be $k = k_1 \times k_2 \times \ldots k_n$ possible receive-ordered computations. We know (as in Theorem 13) that applying predicate detection to each such receive-ordered computations would be equivalent to applying predicate detection to the original computation. So we can solve CPG by applying our algorithm for CPR to k receive-ordered computations. taking $O(k(mn^2 + e))$ time. Notice that this degrades to the second naive approach in the worst case but achieves good results if k is small, or the original computation is close to being receive-ordered. A similar approach could be used if the computation were close to being send-ordered. Further, by decomposing it into receive-ordered computations instead of locally ordered computations, we save an exponential number of applications of a predicate detection algorithm as compared to the second naive approach.

Chapter 5

Controlled Re-execution: An Experimental Study

In this chapter, we describe an experimental evaluation of the controlled re-execution method, based on the predicate control algorithms of Chapter 3.

5.1 Overview

First, in Section 5.2, we describe the application domain and assumptions for our study.

Section 5.3 describes the controlled re-execution method along with two alternative re-execution methods: simple re-execution and locked re-execution. We also make a *qualitative* evaluation of controlled re-execution relative to simple and locked re-execution. We conclude that controlled re-execution has the disadvantages of making the "piece-wise determinism" assumption and involving an extra tracing cost during normal operation, and the advantages of being able to give a guarantee of recoverability and it has a higher likelihood of recovery than the other two approaches. It remains to show quantitatively: (1) how much the cost of tracing is, and (2) how much better is controlled re-execution at being able to recover.

Section 5.4 gives the details of our experimental setting, including implementation, environment, benchmarks, and parameters. We used synthetic benchmarks to allow flexibility in choosing important parameter values such as frequency of communication and frequency of file accesses.

Section 5.5 presents the experimental results. We conclude that the tracing overhead is less than 1% even for applications with relatively high communication and file access frequencies (with respect to real scientific applications). We also conclude that controlled re-execution has a significantly higher ability to tolerate races than either the simple or the locked re-execution methods under varying values of communication and file access frequencies.

In Section 5.6, we discuss extending controlled re-execution to other scenarios. This is a discussion of what we would expect if some of the assumptions made in Section 5.2 are weakened. Important extensions that we discuss include extensions to read-write races and various blocking and non-blocking communication primitives.

In Section 5.7, we give a brief summary.

5.2 The Context

The targeted application domain consists of long-running, non-interactive distributed applications over a local area network. The applications communicate using messages and share files on a network file system. Some examples of applications in this class include weather forecast systems and fluid dynamics code in aerophysics. Messages are used in these applications both to communicate data and for synchronization. Shared files are used to log intermediate results for persistence, for debugging traces, and as an alternate communication mechanism. We chose this application domain as a starting point for studying the effectiveness of controlled re-execution since their long-running and non-interactive nature make them suitable for rollback recovery in general, and further, their simple and well-defined interfaces for communication and file i/o facilitate controlled re-execution.

Message passing is point-to-point and reliable. No assumption is made regarding the order of delivery. We assume, for simplicity, that all send invocations are non-blocking and all receive invocations are blocking. We will consider other blocking/non-blocking semantics in Section 5.6.

Files in the network file system are accessible from all the processes in the application. While the file system provides a view of basic read and write operations, the programming interface is through higher-granularity file access operations. Each file access consists of a clearly demarcated region of code (e.g. a method of a persistent object in object-oriented programs, or a procedure in procedural programs). A file access consists of reads and writes to the file as well as other operations including communication operations, but excluding other file accesses. A file access is called a *write file access* if it contains at least one write operation and a *read file access* otherwise. We assume that all write file accesses are synchronous (i.e. all writes reach the file before the end of the access). This assumption is necessary because adding synchronizations would be unable to prevent races for asynchronous writes since the actual time at which the data reaches the file on disk is indeterminate. Further, we assume that the network file system supports read and write locking. For simplicity, we initially restrict our attention to write-only files (disallowing reads). We will discuss the general case of read-write files in Section 5.6.

The types of synchronization faults that we are concerned with are races (or data races) on files. A *race* occurs when two file accesses are concurrent and at least one of them is a write file access. In order to prevent races, file accesses must be synchronized using either messages or file locks.

5.3 Re-execution Methods

In the context of races, rollback recovery consists of three phases:

1. Detection: detect a race failure.

2. Restoration: restore a *consistent* global state.

3. Re-execution: re-execute to avoid a recurrence of the race.

Both detection and restoration have been widely studied problems. Race detection has been studied in the context of debugging concurrent programs [CL95, FL97, Net91, SBN$^+$97, Tai97]. Restoration has been the central problem of much of the work in rollback recovery for fail-stop faults (as opposed to software faults) [EAWJ99]. Our focus is on the re-execution phase. Before describing the proposed controlled re-execution method, we first describe two natural re-execution methods.

5.3.1 Simple Re-execution

The simplest re-execution method merely allows the application to execute normally. There are two reasons why a race might not recur if a program is simply re-executed. Firstly, if the relative timing of events in different processes changes (for example, owing to different system and network load conditions) then the two file accesses which caused a race in the previous execution may be separated in time in the re-execution. The second reason is that applications sometimes have non-deterministic events which cause the re-execution to differ from the previous execution. Some examples of such non-deterministic events are message receive events (since a different message may be delivered), thread scheduling events, and signal handler events. Note, however, that the simple re-execution method can give no guarantees that a re-execution will be race-free.

5.3.2 Locked Re-execution

One method to guarantee that races do not recur is to use file locks to synchronize all file accesses. However, this locked re-execution method may introduce deadlocks, if the added file locks interfere with the synchronizations of the application. A

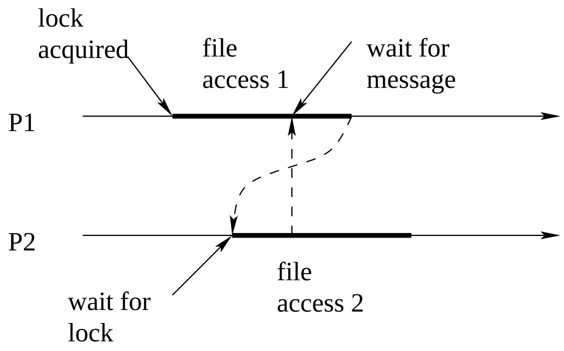

Figure 5.1: A Deadlock Scenario

deadlock is due to a cyclic *wait-for* dependency caused by added file locks and application blocking message receives. A simple wait-for cycle is shown in Figure 5.1.

Deadlock scenarios arise because of file accesses which contain communications that synchronize. The traditional methodology of locking is to follow a discipline of programming that forbids such wait-for synchronizations inside locked critical sections. However, since the imposition of locks during re-execution is adventitious rather than planned, such situations may occur. For example, the file access procedure might involve information gathered from multiple processes. As another example, a file access procedure might make multiple logs for tracing purposes interspersed with communications in the code being traced.

Since the locked re-execution may, in general, have different relative timings and non-deterministic choices than the first execution, it cannot be guaranteed before-hand that the re-execution will be deadlock-free.

5.3.3 Controlled Re-execution

Controlled re-execution aims at preventing both races and deadlocks. In controlled re-execution, explicit *control messages* are used to synchronize the file accesses in a pre-determined order that does not conflict with the existing synchronizations. Further, the previous execution is replayed rather than simply re-executed (i.e. all non-deterministic events, such as message receives, are replayed [NM92]). Thus, the replayed execution together with the added control messages form a *stricter partial order* of events than the first execution. In other words, the problem of adding synchronization edges appropriately is a direct application of the predicate control problem for the general mutual exclusion predicate discussed in Chapter 3. File accesses are treated as critical sections for the purposes of the algorithm. All that remains is determine how to: (1) trace during normal execution to obtain the input computation for predicate control, and (2) add control messages corresponding to the extra edges in the output computation of predicate control.

5.3.3.1 Tracing

In order to represent the partial order relationship between events (required as input to the predicate control algorithm), a *vector clock* mechanism [Mat89, Fid91] is used. A vector clock is an n-sized vector with one entry per process in the system. We refer the reader to [Gar96] for a detailed discussion on vector clock mechanisms. For our purposes it is sufficient to know that for a computation $\langle E, \rightarrow \rangle$, a vector clock mechanism assigns to each event $e \in E$ a vector clock $e.v$ such that:

$$\forall e, f \in E :\ e \rightarrow f \ \Leftrightarrow\ e.v < f.v$$

where $e.v < f.v \ = \ (\forall k :\ e.v[k] \leq f.v[k]) \ \wedge\ (\exists k :\ e.v[k] < f.v[k])$.
Further, we can compare two events in constant time using the following property of vector clocks:

$$\forall e, f \in E :\ e \rightarrow f \ \Leftrightarrow\ (e.v[e.proc] \leq f.v[e.proc]) \ \wedge\ (e.v[f.proc] < f.v[f.proc])$$

In order to trace the file accesses, we must interrupt execution at the begin and end events for the file access. The vector clock value at these events is recorded. By maintaining the sequence of these vector clocks for each process, we have the input interval sequences for the predicate control algorithm in Figure 3.6.

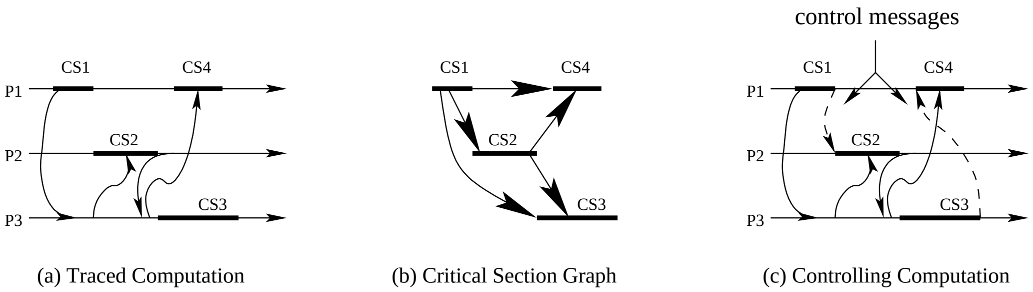

Figure 5.2: Controlled Re-execution

In addition to tracing for control, we must also trace all non-deterministic events so that they may be replayed during re-execution. The ability to trace such events is referred to as the *piece-wise determinism assumption* since it allows a process execution to be divided into deterministic pieces. The same assumption is also made in rollback recovery systems based on message logging [EAWJ99] and trace-replay systems for debugging [CS98, LMC87, NM92, RZ97, Net93]. For the applications under consideration, we assume that receive events, which may receive a different message in a re-execution, are the only non-deterministic events (for now, we are assuming that files are write-only; we deal with reads and other types of non-deterministic events in Section 5.6). Therefore, information to identify a message (e.g. sender's id and position in sender's sequence of sent messages) must be traced at each receive event to be used in replaying the same message delivery order during re-execution.

To summarize the key points of the tracing mechanism, the execution is interrupted at three points: (1) receive events, (2) begin file access events, and (3) end file access events. The traced information allows for replay. Furthermore, the traced information provides the input computation for the predicate control algorithm. An example of such a traced computation is depicted in Figure 5.2(a).

5.3.3.2 Control

Once the predicate control algorithm of Figure 3.6 is applied, the output from the algorithm consists of a set of edges that must be added to the original computation to obtain a controlling computation of mutual exclusion. Suppose (e, f) is one such added edge. Note that, according to the algorithm, e is the end event for some file access and f is the begin event of some other file access. Therefore, during re-execution, when process $e.proc$ reaches the end file access event e (identifiable by comparing with vector clock $e.v$), it sends a control message to process $f.proc$. The control message has an empty message body and has a *message envelope* consisting of sender and receiver identification, vector clock $e.v$, and a message type (tag) value that is distinct from all application messages. When the process $f.proc$ reaches the begin file access event f (identifiable by $f.v$), it first performs a blocking receive, waiting for a control message from $e.proc$ with vector clock value $e.v$. Once the message is received, it continues normal execution. [1]

Figure 5.2 illustrates a simple scenario of controlled re-execution in which all critical sections (file accesses) correspond to a single file. The traced computation in Figure 5.2(a) has races since the application synchronizations do not ensure mutual exclusion. The predicate control algorithm uses the critical section graph shown in Figure 5.2(b) to determine the edges to be added. Based on the added edges, control messages are sent during re-execution as shown in Figure 5.2(c).

There is a seeming contradiction in replaying the non-deterministic choices of a failed execution in order to prevent a recurrence of a failure. However, it must be stressed that it is only the process executions that are replayed and not the sequence of operations on a file. Since the failure outcome of the race is data interleaving

[1] Owing to the manner in which the algorithm orders critical sections, an optimization can be made : the vector clock value $e.v$ need not be sent with the control message. The receiving process $f.proc$ merely blocks waiting for the *next* message to be received from $e.proc$. It is verifiable that, even without any assumptions on message ordering, this ensures that the message sent from e is received at f.

on the write-only files, replaying the execution with added control synchronizations ensures that the files are not corrupted during re-execution. The case of read-write files will be discussed in Section 5.6.

5.3.4 A Qualitative Evaluation

Having described the operation of the three re-execution methods, we now make a qualitative evaluation of the advantages and disadvantages of controlled re-execution with respect to simple and locked re-execution.

5.3.4.1 Disadvantages of Controlled Re-execution:

1. **Tracing Cost:** Unlike simple and locked re-execution, controlled re-execution requires tracing of information for control and replay. Tracing imposes a cost both in space and in time. This cost is particularly important since it is imposed during failure-free operation.

   The cost in time does not involve the cost of writing the traces synchronously to disk. This is because we are not considering failure scenarios, such as power failures, in which volatile storage is lost. The cost in space involves the size of one vector clock value for each receive event, begin file access event, and end file access event. This is in contrast to the cost already imposed in order to implement the restoration phase of recovery, typically involving checkpointing and message logging.

2. **Piece-wise Determinism Assumption:** Controlled re-execution assumes that all non-deterministic events can be identified, traced, and replayed. For example, by assuming that the only source of non-determinism is the receive events and tracing the order of message delivery. This assumption is not required by either simple or locked re-execution.

While maintaining the piece-wise determinism assumption is a challenging problem, the application domain under consideration has fewer sources of non-determinism than more general applications that interact more frequently with their environments. Sources of non-determinism usually arise from message ordering, thread scheduling, interrupts, or system calls. Dealing with such sources of non-determinism has been a widely-studied problem in the context of message logging protocols [EAWJ99]. When the restoration mechanism used for rollback recovery involves message logging, the piece-wise determinism assumption is already made regardless of the re-execution scheme used. Therefore, controlled re-execution imposes no additional assumptions in such cases.

5.3.4.2 Advantages of Controlled Re-execution:

1. **Ability to Recover:** Suppose an application has failed and restored a consistent global state, which one of the three methods has a greater likelihood of providing a safe re-execution? The likelihood of simple re-execution providing a safe re-execution (no races) depends on timing and the likelihood of file access overlap. For locked re-execution, the likelihood of a safe re-execution (no deadlocks) depends on the likelihood of cyclic file access/communication synchronizations. For controlled re-execution, the likelihood of a safe re-execution (no cycles in the critical section graph) depends on the likelihood of cyclic communication synchronizations.

   For the application domain under consideration, the non-determinism owing to message reordering is expected to be low, since a receive usually corresponds to a unique message. Thus, it is expected that a re-execution has the same synchronization pattern as the failed execution. Therefore, the only situation in which the mutual exclusion predicate control algorithm (for a single file) fails is when a safe re-execution is impossible (refer to Section 3.5). Thus, the likelihood of recovery in controlled re-execution is at least as high as that in simple and locked re-execution. It remains to determine how much higher the likelihood actually is.

2. **Guarantee:** Independent of the expected recoverability, controlled re-execution also has the advantage of being able to guarantee before-hand whether it will be able to provide a safe re-execution. In the case of simple and locked re-execution there are no such guarantees and the re-execution must be attempted before its safety is determined. Thus, repeated re-execution - detection - restoration cycles are avoided in the case of controlled re-execution.

   This advantage is especially important in the context of a complementary framework consisting of all three schemes. In such a framework, if one re-execution method fails, another may be attempted. The advantage of an *a priori* guarantee is useful in such a framework, since it is immediately clear whether controlled re-execution should be used.

In order to make a complete evaluation of controlled re-execution, we require a quantitative evaluation of: (1) how much of a disadvantage is posed by the tracing cost, and (2) how much of an advantage is involved in the improved ability to recover. This is the objective of the experimental study described in the following sections.

5.4 Experimental Setting

5.4.1 Implementation and Environment

We implemented the three re-execution methods in the form of a library that augments the functionality of MPI (Message Passing Interface). The relevant calls – send, receive, begin file access, and end file access – are intercepted by the augmented library before passing control to the normal routines.

Our implementation was based on mpich 1.1 [GL96], a portable implementation of MPI. To provide synchronous writes over a network, we exported a remote interface to a local Unix file system. Tracing was implemented both in-memory and using asynchronous writes to local files. Control messages were implemented using blocking MPI messages with a unique *tag* value.

Since our experiments focus on the re-execution phase, we implemented simple strategies for the detection and restoration phases of recovery. For failure detection, we performed an integrity check on the files to check for data corruption. Restoration was implemented by restarting all processes, thus restoring the initial consistent global state. This is valid for our experiments since the synthetic benchmarks we use (described in the next section) are not sensitive to the starting point of the re-executions.

The experimental environment consists of 4 Sun Ultra 10 Workstations (440 MHz) running Solaris 2.7 and connected by a 100 Mbps Ethernet. Each workstation has 128MB of DRAM and a 4GB local disk.

5.4.2 Synthetic Benchmarks

For our experiments, we used synthetic benchmarks which allow various critical parameters, such as the frequency of communication and the frequency of file access to be varied independently. Each synthetic application is generated by running a loop, each iteration of which picks an event with a pre-specified probability from among a set of possible events. The possible events are: sends, receives, file access begins, file access ends, file writes, and local events. The file accesses ensure that file writes occur within the context of a file access and, in the default case, file accesses are not nested. The recipient of a message is chosen uniformly at random. Receives are blocking, waiting on a specific process in such a way as not to create communication deadlocks (determined by a sample non-blocking execution). The file corresponding to each file access is selected uniformly at random. Similar synthetic applications have also been used in the study of message logging recovery protocols [ARV98] and logical clock schemes [TRA96].

The main reason for selecting synthetic benchmarks is because it gives us flexibility in selecting application parameters such as communication and file access frequencies. We selected the parameters values for the synthetic benchmarks based on measurements of the NPB 2.0 Benchmark Suite [BHS+95] consisting of five computational fluid dynamics codes and the NAS Application I/O (BTIO) Benchmark [Fin97] consisting of one of the NPB 2.0 Benchmark codes including file i/o. In a similar manner as the BTIO Benchmark, we instrumented the remaining four NPB 2.0 Benchmark codes with file i/o. Further, we referred to two other studies of scientific application codes: [Rao99] for measurements on the same NPB Benchmark Suite and [AHKM96] for a detailed study of eight scientific applications that use messages and file i/o. Furthermore, we referred to [KN94] for a study of dynamic file-access characteristics of a production parallel scientific workload.

A further advantage of using synthetic benchmarks is that the NPB benchmark applications do not account for possible non-determinism in the application codes. This biases our comparitive study of recoverability in favor of controlled re-execution and against simple re-execution. The synthetic benchmarks allow us flexibility in including non-determinism since a different re-execution can be generated based on the same parameters to model a non-deterministic application (alternatively, a single synthetic application can be replayed multiple times to model a deterministic application).

5.4.3 Parameters

The applications under consideration have two main characteristics: the frequency of communications and the frequency of file accesses. These are represented by the following two parameters:

- **mean communication period:** the average time between consecutive communication events (receive or send events) on a process, averaged over all processes.

- **mean file access period:** the average time between consecutive file accesses on a process, averaged over all processes.

Note that the parameters are controlled indirectly by varying the relative probabilities of communication or file access events for the generated synthetic application. Measurements are then made to determine their values.

There are other parameters for the synthetic applications for which we set default values, unless otherwise specified. These parameters are the message size (default: 100 bytes), the file write size (1 KB), the mean file access time (90 - 110 ms), the mean file-i/o period (30 - 35 ms), the number of files (1), and the fraction of file-i/o that are writes (1). The defaults were chosen based on the real application parameters (from the sources listed above) and the assumptions for the common-case.

5.5 Experiments

5.5.1 The Costs of Controlled Re-execution

Unlike simple and locked re-execution, controlled re-execution imposes the extra penalty of tracing during normal execution. First, we give the details of exactly what extra information is traced and at what points. There are three execution points at which a trace needs to be taken:

- **receive events:** At each receive, the following information is traced: the sender's id, the message tag (used to classify messages into types in MPI), and the sender's entry in the message's vector clock.

- **begin file access events:** At each begin file access event, the following information is traced : the vector clock value, whether it is a read or a write file access (in the general case), and the file id for the file access.
- **end file access events:** At each end file access event, the vector clock value is traced.

Tracing may be implemented in-memory or using asynchronous writes to files. We do not use synchronous writes since we assume that a failure is detected in time to save volatile memory to stable storage. The trace may be periodically shortened by a garbage collection mechanism, under the assumption that the execution has progressed far enough that the deleted information will never be required.

Tracing has a cost both in time and space. We use two simple metrics for these costs:

- **time overhead**: This is the percentage increase in total application running time when tracing is performed. The time overhead is measured both for in-memory and asynchronous file tracing. Note that the time overhead is for an application that does not incorporate other recovery mechanisms, such as checkpointing and message logging (the overhead calculated in consideration of existing recovery mechanisms is smaller than our measure).
- **space cost**: This is the amount of space required per unit time (in MB/hour).

We measured the tracing costs for applications with relatively dense communication and file access patterns. Note that mean communication period values are in the range 10 - 200ms for the NPB Benchmarks and mean file access period values are in the range 100 - 10000 ms (based on [Rao99] and our measurements, assuming one file access every iteration). Table 5.1 shows the results.

The time overheads were under 1% for all applications. Further, this is true for both in-memory tracing and asynchronous file tracing.

Mean comm. period (ms)	Mean file access period (ms)	Time overhead		Space (MB/hour)
		In-memory tracing	Asynch. file tracing	
30	183	0.21%	0.55%	0.72
3	159	0.15%	0.69%	2.51
87	67	0.03%	0.05%	1.50

Table 5.1: The Tracing Cost of Controlled Re-execution

The space cost is small considering that typical scientific applications can require in the order of 1GB of disk space for storing application data [AHKM96]. Further, existing restoration techniques such as checkpointing and message logging would typically have much larger disk space requirements. The space requirements can be further reduced by garbage collection making in-memory tracing feasible. Note that space required in any given period of time is exactly $12r + (8n + 4)f$ bytes where r is the number of receive events, n is the number of processes, and f is the number of file accesses.

We conclude that the tracing costs for controlled re-execution are very small.

5.5.2 The Benefits of Controlled Re-execution

The benefits of controlled re-execution lie in improved likelihood of recovery. In these experiments we aim at measuring how much of a gain this is. We use synthetic applications with randomly generated communication and file access patterns. Therefore, races are automatically generated periodically. In such a simulated faulty scenario, we test how far an execution can proceed under each of the three methods before a failure occurs. Note that the meaning of a failure is different for the three methods:

- *simple re-execution:* A failure occurs when a race causes data to be corrupted on the file. We measure the time of failure at the time of the concurrent writes to the file.
- *locked re-execution:* A failure is a deadlock. We measure the time of failure as the minimum time that a process involved in the deadlock starts to wait for a resource.
- *controlled re-execution:* A failure does not really occur. Instead, controlled re-execution can guarantee before-hand whether or not it can execute safely. In cases where it cannot give such a guarantee, there must be a cycle of critical sections in the interval graph. We say that controlled re-execution fails if it cannot guarantee a safe re-execution and we measure the time of failure as the minimum of the times at which a critical section in the cycle is entered. This is fair since controlled re-execution can safely execute upto this point in time.

The metric we use is:

- **mean time to failure (mttf):** The time from the beginning of the execution, averaged over all processes, upto the time of failure (as defined above).

The mttf for a re-execution method is an indicator of the ability of that method to tolerate faults in a faulty application. The greater the mttf, the greater is the likelihood of recovery.

Figure 5.3 shows the plot of mttf against mean communication period for simple, locked and controlled re-execution methods. Multiple synthetic applications of the corresponding parameter values were executed to obtain each mttf point. We varied the mean communication period over a range chosen based on applications in the NPB benchmarks (10 - 200 ms). The mean file access period was kept fixed at 200 ms. It is important to note that the scale is logarithmic since the corresponding

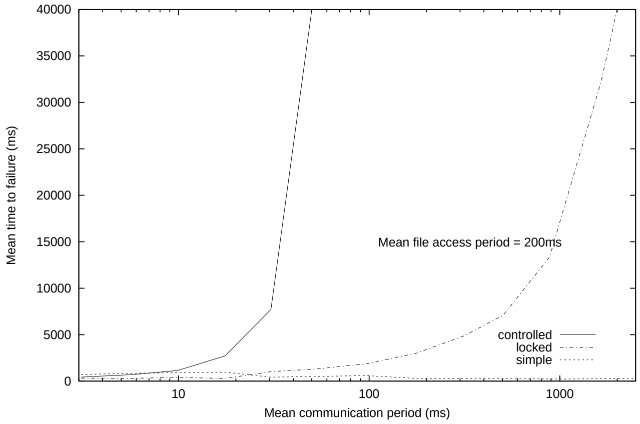

Figure 5.3: Plotting mttf against mean comm. period

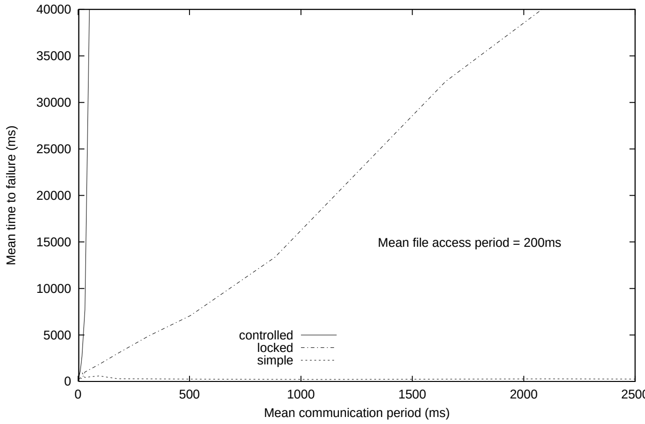

Figure 5.4: Plotting mttf against mean comm. period on a linear scale

linear-scaled graph shown in Figure 5.5.2 has too great a difference between the three mttf plots for the three methods to be clearly depicted.

For both locked and controlled re-execution, we expect that the likelihood of a failure (deadlock/cycle of critical sections) is expected to be lower as the mean communication period increases and the communications get less dense. Clearly, simple re-execution remains unaffected by the communication pattern changes since the file access density is fixed.

The reason that controlled re-execution has a greater mttf than locked re-execution is that for a given communication density, the likelihood of a deadlock is significantly greater than the likelihood of a cycle of critical sections. If there is a cycle of critical sections, there must necessarily be a deadlock. Further, many deadlocks do not involve cycles of critical sections since a deadlock may occur when file locks are acquired in an order that interferes with the messages, whereas a cycle of critical sections depends solely on the messages.

The reason that controlled re-execution has a greater mttf than simple re-execution is that if there is a cycle of critical sections then two of those critical sections must occur at the same time leading to a race. However, if a race occurs there need not be any synchronization cycles (and the likelihood reduces as the communication density decreases).

Next we plot mttf against mean file access period while keeping the mean communication period fixed. Figure 5.5 shows the results. The parameters for mean file access period were picked based on applications in the NPB benchmark.

The reason all three graphs rise with increasing mean file access period is as follows. As the file access density decreases, so does the likelihood of two file accesses occurring concurrently. Therefore, races are less likely. Further, deadlocks are also less likely since the file accesses are spaced further apart making it less likely for the locks to interfere with synchronizations. Further as critical sections get spaced

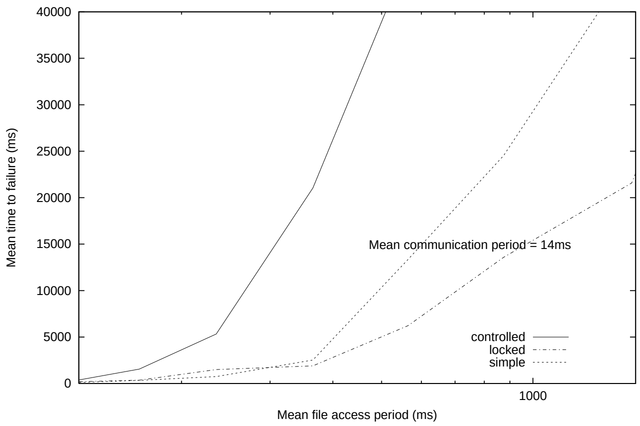

Figure 5.5: Plotting mttf against mean file access period

further apart, the chance of a cycle of synchronizations also diminishes.

The reason controlled re-execution has a higher mttf than the other two methods is as follows. If there is a cycle of critical sections, then clearly there must be a race since two critical sections in the cycle must occur together. Further, there must necessarily be a deadlock. However, the reverse is not true.

Thus, the overall conclusion of these experiments is that the mttf is significantly higher for controlled re-execution than for locked and simple re-execution.

5.6 Extensions

In this section, we discuss some extensions to the controlled re-execution method. Each of the extensions results from weakening one of the assumptions made in this study.

Extension to other blocking/non-blocking communication primitives

We assumed that for all communications, sends are non-blocking and receives are blocking. This assumption was made for simplicity and the other combinations can be handled as well. The case of both blocking sends can be modeled by an extra message (in addition to the one being sent) that is sent by the receiver after the message transfer completes and for which the sender blocks waiting. The case of non-blocking receives can be handled by simply treating them in the same way as blocking receives for the purpose of predicate control. Often, non-blocking receives are made in cases where the relative order of message delivery does not affect the future synchronizations. In such cases, the non-blocking receives need not be traced and the computation input to predicate control consists of only the blocking receives.

Extension to read-write files

We have assumed that files are write-only for the purpose of our study and so the only races are write-write races. If the processes were to read from the file as well, then the reads cause an extra communication channel between processes and also introduce another potential source of non-determinism based on the order of file accesses. There are two scenarios with different requirements for dealing with read-write files.

In the first scenario, we consider the general case with no assumptions. The reads must be traced for the dependencies they cause just as messages with vector clocks being maintained for the files in addition to the processes. Since controlled re-execution attempts to change the order of accesses at a file, in general, it is impossible to replay the same execution beyond the first recovered race. However, in cases where the change in order of file accesses does not affect the computation, the replay can progress. It is possible to continue execution using controlled re-execution upto the point where a divergence from the previous execution is detected. No guarantees can be made beyond this point.

The second scenario is applicable to applications whose synchronization patterns do not change with the values read from files. This is often true in the application domain under consideration since, although the values of communication messages often change with the values read from files, the order of synchronizations remains unchanged. Since controlled re-execution is only sensitive to the order and not the content of messages, in such cases it is not necessary to trace and replay read accesses.

Extension to other non-deterministic events

In our study, we make the commonly made assumption that receive events are the only sources of non-determinism. In general, there may be other sources of non-determinism. We have already discussed non-determinism induced by read accesses from files. Some other sources of non-determinism are thread scheduling events in multi-threaded programs, system calls, and asynchronous signals. Much work has been done in the context of rollback recovery [GGL$^+$90, SE96] and in distributed debugging [TCO91, MCL89, RC96] towards tracing and replaying multi-threaded programs. For a detailed discussion of system call and asynchronous signals, we refer the reader to [EAWJ99]. If the recovery system uses message logging, it would already require to trace and replay non-deterministic events. In such cases, the necessary mechanisms would be already available for controlled re-execution.

5.7 Summary

To summarize our evaluation of the controlled re-execution method with respect to simple and locked re-execution methods.

- **Advantages:**
  - **Ability to recover:** Our experiments tested the ability of the three re-execution methods to tolerate races in synthetic applications under different values of communication and file access density. The results showed that the mean time to failure was significantly higher in the case of controlled re-execution.

- **Guarantee:** Controlled re-execution does not depend on chance and can guarantee a race-free re-execution *a priori* while the other two methods cannot.

- **Disadvantages:**
  - **Piece-wise Determinism Assumption:** The controlled re-execution method makes the piece-wise determinism assumption while the other two methods do not.
  - **Tracing Overhead:** Unlike the other two methods, controlled re-execution involves a tracing cost. However, our experiments on synthetic applications with high communication and file access densities show that the time overhead is less than 1% and the space cost is in the range of 0.5-2.5 MB/hour.

Chapter 6

Related Work

We discuss the related research in three sections corresponding to the topics of each of the previous three chapters: the predicate control problem, the predicate detection problem, and controlled re-execution.

6.1 The Predicate Control Problem

The predicate control results in this dissertation were first published in [TG98c] and [TG99]. We are aware of two previous studies of controlling distributed systems to maintain classes of global predicates. One study [MSWW81] allows global properties within the class of conditional elementary restrictions [MSWW81]. Unlike our model of a distributed system, their model uses an off-line specification of pair-wise mutually exclusive states and does not use causality. [Ray88] and [TG94] study the on-line maintenance of a class of global predicates based on ensuring that a sum or sum-of-product expression on local variables does not exceed a threshold. In contrast to these approaches, our focus is on general global boolean predicates, disjunctive predicates, and mutual exclusion predicates.

Since our results in predicate control were published, some advances in solving predicate control have been made in [MG00]. The authors present an equivalent formulation of the predicate control problem in terms of "admissible sequences". Based on this, they provide an algorithm for disjunctive predicates that has the same time complexity as the algorithm presented in this dissertation, but has the added advantage of generating the minimum synchronizations. They also solve the predicate control problem for the class of "region predicates" that can express such conditions as deadlock and termination.

Relation to Predicate Detection

Since there is a duality between observation and control, it is not surprising that there is a relation between predicate detection, a formalization of observation, and predicate control, a formalization of control. To understand the relationship, consider a computation in which *definitely* : $\neg\phi$ is detected, where the modality *definitely* for a predicate [CM91] implies that the predicate becomes true in *any* possible execution (run) corresponding to the computation. Clearly, there can be no controlling computation of ϕ in the computation since any execution corresponding to the controlling computation is also an execution corresponding to the original computation. Therefore, a controlling computation of a predicate ϕ in a given computation exists if and only if *definitely* : $\neg\phi$ is not detected in the computation. As evidence of this, the necessary and sufficient conditions for detecting *definitely* : ϕ for conjunctive predicates were demonstrated in [GW96], and are similar to our results in predicate control for disjunctive predicates. An important difference between the two problems is that, though the predicate detection problem for *definitely* : $\neg\phi$ can be used to determine whether a controlling computation of ϕ exists, it is not concerned with actually finding the controlling computation. Another interesting conclusion is that all of our predicate control algorithms can be used to solve a corresponding predicate detection problem.

Relation to On-line Synchronization

The predicate control problem is an off-line synchronization problem since it assumes that the computation is known *a priori*. In contrast, existing synchronization problems [BA90] such as mutual exclusion, readers writers, and dining philosophers are on-line and require synchronization without such pre-knowledge. The predicate control problem for each of the predicates we have studied has a corresponding on-line synchronization problem, in which the computation is an on-line rather than an off-line input. Predicate control for mutual exclusion predicates corresponds to the widely-studied distributed mutual exclusion problem [Ray86, BA90, Vel93, Sin93]. Predicate control for readers writers predicates corresponds to the readers writers problem [CHP71, Lam77, BA90]. Predicate control for independent mutual exclusion predicates does not correspond to a new on-line problem. Instead, multiple distributed mutual exclusion protocols operate independently assuming that deadlocks are prevented using a programming discipline such as two-phase locking. Predicate control for disjunctive predicates corresponds to a special case of the k-mutual exclusion problem [Ray89, SR92, MBB$^+$92, HJK93, BV95] (that allows up to k processes inside a critical section at the same time) in which $k = (n - 1)$.

Each of the on-line problems above make assumptions on the programming discipline, such as non-blocking critical sections or two-phase locking. In a general context without such assumptions, it can be shown that the on-line synchronization problems stated above are impossible to solve [TG98c], and therefore, it is impossible to maintain the predicate while avoiding deadlocks. Our solutions to the predicate control problems do not make such assumptions. Instead, they make use of the pre-knowledge of the computation to maintain the predicate while avoiding deadlocks. Thus, off-line synchronization is, inherently, a more powerful technique than on-line synchronization.

Another distinction between the on-line and off-line problems is that the predicate control problems operate on a single computation of the distributed program. The on-line synchronization problems, on the other hand, operate at the level of the programming language and must, therefore, correctly synchronize for all possible computations of the program.

6.2 The Predicate Detection Problem

The results in predicate detection presented in this dissertation were first published in [TG98a]. Surveys of predicate detection may be found in [BM93, SM94, Gar96]. There have been three approaches to solving predicate detection. The first approach [CL85, SK86, HJPR87, Bou87] is based on periodically computing a consistent cut of the system through a global snapshot protocol and checking whether the cut satisfies the given predicate. This method works only for stable predicates, such as deadlock and termination, which remain true once they become true. However, in general, predicates need not be stable and may become true only between consecutive snapshots. The second approach [CM91, AV94, JMN95] constructs the entire lattice of cuts to detect a given predicate. This approach is able to detect unstable predicates but it has an exponential time complexity of $O(k^n)$ where k is the number of "relevant" local events and n is the number of processes. The third approach [MC88, MI92, AV93, HPR93, TG93, GW94, BR94, FRGT94, JJJR94, CG95, GC95, GTFR95, TG95, VD95, BFR96, GW96, HMRS96, HR96, GCKM97] is based on using the structure of the predicate. Instead of constructing the entire lattice of cuts, the approach uses the computation directly. Thus, efficient algorithms can be constructed , but they are restricted to certain classes of predicates. In this dissertation, our focus has been on the third approach.

Within the third approach, we focus on predicates that are defined on a single cut. There are two modalities defined for detecting a predicate ϕ defined on a single cut. The first modality, *possibly:*ϕ, is defined to be true if the lattice of cuts has a path from the initial cut to the final cut containing at least one cut that satisfies ϕ. The second modality, *definitely:*ϕ, is true if every path from the initial cut to the final cut in the lattice of cuts contains at least one cut that satisfies ϕ. Note that our definition of predicate detection corresponds to *possibly:*ϕ.

It has been shown that for the important class of conjunctive predicates, predicate detection can be solved efficiently for both *possibly:*ϕ [GW94] and *definitely:*ϕ [GW96]. Although these solutions are centralized, distributed algorithms have also been designed [GC95, HMRS96]. Further, conjunctive predicates have been extended to "channel predicates" [GCKM97] that consider both the states of the message channels and the states of the processes. The work most closely related to our study is [GW94]. The main difference is that while they use the happened before model of compoutation, our study uses the extended model of computation which allows improved predicate detection. We use the algorithm presented in [GW94] as part of our solution for detecting conjunctive predicates in the extended model.

The Extended Model of Computation

In our study of the predicate detection problem, we used an extended model of computation that partially orders the events both across processes and within a process so as to reduce false causality. Using partial orders to model concurrency avoids the combinatorial explosion involved in the interleaving model. This fact has led to a number of studies of such partial order models [Lam78, Pra86]. Problems in *false causality* have been observed in application domains other than predicate detection, such as: race detection [SBN$^+$97], causally ordered communication [CS93], and optimistic recovery [DTG99]. Models that extend the happened before model [Lam78] by allowing partially ordered events within a process have also been proposed in [ACG93, HW96, TG98b].

6.3 Controlled Re-execution

For a good introduction to the past work in software fault tolerance, the interested reader is referred to [Lyu95]. There are three approaches to tolerating software faults. The "design diversity" approach [AC77, Ran75] uses redundant modules to prevent a single point-of-failure. This method is effective provided the same fault does not appear in different modules. The "data diversity" approach [AK88] use techniques to re-express input data in an attempt to bypass faults. However, such re-expression is effective only for a limited range of applications. This dissertation is concerned with the "environment diversity" approach, first introduced in the progressive retry method [WHF+97] which aims at bypassing a fault by rolling back the application and restarting from a previous state. For certain "transient" faults, the progressive retry method was found to be effective.

The rollback recovery method used in [WHF+97] involves three distinct phases: detection, restoration, and re-execution. Our focus has been on the detection and re-execution phases. However, much work has been done in the restoration phase in the context of conventional (not software) fault tolerance. For a survey of general rollback recovery methods in distributed systems for global state restoration, the reader is referred to [EAWJ99]. Aspects of restoration peculiar to software fault tolerance have been studied in [WHF+97, CC98a].

We have already discussed the related work for the detection phase in the previous section on predicate detection. With respect to the re-execution phase, existing approaches [WHF+97] and [CC98a] have targetted general faults and have used a simple re-execution technique based on simply restarting the system from a restored state, with possible message reorderings. These re-execution techniques cannot guarantee a safe re-execution. Our controlled re-execution approach focuses on the class of synchronization faults and can give a guarantee of safe re-execution based on controlling the re-execution with added synchronization messages.

Relation to Speculative Computing

Speculative computing is another rollback approach for tolerating synchronization faults in both shared-memory multiprocessor environments [Smi81, SP98, APR99] and distributed shared memory environments [HW94, KCDZ95, KS97, BPA98, Mue99]. Speculative or optimistic computing is more aggressive than conventional fault tolerant computing in that faults are allowed to occur intentionally. This allows greater efficiency by avoiding expensive fault prevention techniques that use locks or barrier synchronizations. Speculative computing methods generally assume a weak consistency model such as release consistency [KCDZ95] or group write consistency [HW94]. Further, they assume that the level of atomicity is at the level of reads and writes to a single object. In contrast, the controlled re-execution method assumes a sequential consistency model [Lam79] and atomicity at the level of multi-object operations [MG98] including sends and receives of messages and reads and writes to files.

Another distinction is in the re-execution method employed. The forms of control used in speculative methods are distinct from that used by controlled re-execution – controlling the relative timing among processes through synchronization. For example, branch prediction schemes [Smi81, SP98] ensure that a different execution path is followed during re-execution based on the faulty branch choices made during the previous execution. In another example, [KCDZ95] describes a hybrid scheme as a compromise between lazy release consistency and eager release consistency. The protocol operates by optimistically sending only a fraction of the data that would be sent by an eager protocol. On encountering invalid data, the protocol switches to lazy mode, explicitly acquiring the data. Unlike these specialized forms of control, controlled re-execution limits its control to the relative timing among processes.

Chapter 7

Conclusions and Future Directions

Software fault tolerance has conventionally been perceived as a passive approach consisting of monitoring the system and attempting general purpose techniques such as restarting from a previous checkpoint or failing over to a backup process. In this dissertation we have aimed at demonstrating the benefits of a more active approach to software fault tolerance that involves taking *corrective* rather than *reactive* measures based on an understanding of the failure and its causes. This involves a more aggressive approach to the two functions of a run-time environment – *observation* and *control*. The observation function must involve monitoring the system not merely to detect failures but also to provide information of the causes that led to the failure. The complementary control function uses this information to decide on appropriate corrective actions.

We make a first step in this direction by showing how, in the case of synchronization faults, by tracing synchronization information during normal execution, one can take more effective corrective action during re-execution. This led us to the investigation of the little studied off-line synchronization problem – *the predicate control problem*. We designed algorithms for solving the predicate control problem for some important classes of predicates. This is encouraging, especially in the light of the fact that the corresponding and better studied *on-line* synchronization problems are impossible to solve.

In addition to control, we also worked towards improved observation methods by studying and designing algorithms for detecting conjunctive predicates in an extended model. The extended model is especially important since it allows improved detection, but we find, not surprisingly, that it makes the problem significantly harder to solve.

An interesting future direction would be to try to apply corrective approaches to tolerate other classes of software faults. This need not be in a transparent manner, but may involve a programming paradigm in which information that is useful for recovery is made explicit. For example, failures that arise from insufficient memory may be recovered using corrective strategies that invoke garbage collection. Failures arising from timeouts may be corrected by reducing the load on the machine through the migration of some processes.

Another application domain that would benefit greatly from the observation and control paradigm is that of distributed debugging. In fact, since it is more tolerable to incur overhead in debugging than in fault tolerance, more aggressive observation methods could be used to allow improved control. For example, tracing the values of process variables, could allow the control strategy to intelligently reset the variables to test the application under different conditions. It would also be useful to apply predicate control as was done in the controlled re-execution method to test the application under various synchronization conditions.

We have limited our study of the controlled re-execution method for tolerating synchronization faults to race faults. It is important to study the applicability of a similar technique to other types of synchronization faults, such as deadlocks, disjunctive predicate violations, and violations of load balancing. Another restriction we made was to focus on the application domain of long-running, non-interactive applications that communicate with messages and files. This is a natural first step owing to the nice synchronization and shared resource interfaces. An interesting question is whether the same approach could be used in other application domains, such as multi-threaded applications which communicate using wait-notify primitives and share objects in memory. Another interesting application domain is distributed shared memory systems which maintain consistency conditions over distributed shared objects.

In the study of the predicate control problem, there are some interesting open questions. We have solved predicate control for mutual exclusion predicates and for disjunctive predicates. These are two extreme subcases of the more general *k-mutual exclusion* predicates. The k-mutual exclusion predicates specify that at most k of the processes can enter critical sections together. Thus, we have solved the cases of 1-mutual exclusion and $(n-1)$-mutual exclusion. We do know of a necessary condition for solving the predicate control problem for k-mutual exclusion. This corresponds to a set of critical sections in which each critical section has at least k incoming $\mapsto$ edges from the other critical sections. Note that from this, we can derive the necessary and sufficient conditions for both 1-mutual exclusion and $(n-1)$-mutual exclusion. However, it still remains an open question whether the k-mutual exclusion problem can be efficiently solved.

It would also be useful to solve the predicate control problem in the context of *relational predicates* [Gar96]. An example of such a predicate is $(x_1+x_2+\cdots+x_n > k)$, where x_i's are integer variables in different processes and k is a constant. This allows us to express conditions such as the loss of a token and violations of a limited resource.

The extended model of computation allows events in a process to be locally ordered. This addresses the problem of *false causality* inherent in the use of the conventional happened before model. In this dissertation, we have studied the implications of the extended model in predicate detection and in another work we have studied the implications in optimistic message logging protocols [DTG99]. Another application of the extended model can be found in predictive performance models of distributed systems [HW96]. In general, the extended model is useful in applications that benefit from improved causality tracking. Another application domain that falls in this class is that of message ordering systems that ensure causal ordering between messages [CS93]. Applying the extended model in causal ordering may result in more efficient implementations by avoiding unnecessary delaying of messages owing to false causality. However, the exact cost-to-benefit trade-offs remain to be studied.

Bibliography

[AC77] A. Avizienis and L. Chen. On the implementation of N-version programming for software fault tolerance during execution. In *Proc. of the First IEEE-CS International Conference on Computer Software and Applications*, pages 149 – 155, November 1977.

[ACG93] M. Ahuja, T. Carlson, and A. Gahlot. Passive-space and time view: Vector clocks for achieving higher performance, program correction, and distributed computing. *IEEE Transactions on Software Engineering*, 19(9):845–855, September 1993.

[Ada84] E. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2 – 14, Jan 1984.

[AHKM96] R. Ahuja, A. Ho, S. Konstantinidou, and P. Messina. A quantitative study of parallel scientific applications with explicit communication. *The Journal of Supercomputing*, 10(1):5–24, 1996.

[AK88] P. E. Ammann and J. C. Knight. Data diversity: An approach to software fault-tolerance. *IEEE Transactions on Computers*, 37(4):418–425, April 1988.

[APR99] S. V. Adve, V. S. Pai, and P. Ranganathan. Recent advances in memory consistency models for hardware shared memory systems. *Proceedings of the IEEE*, 87(3):445–455, March 1999.

[ARV98] L. Alvisi, S. S. Rao, and H. M. Vin. Low-overhead protocols for fault-tolerant file-sharing. In *Proc. of the International Conference on Distributed Computing Systems*, Amsterdam, The Netherlands, May 1998. IEEE.

[AV93] S. Alagar and S. Venkatesan. Hierarchy in testing distributed programs. *Lecture Notes in Computer Science*, 749:101–116, 1993.

[AV94] S. Alagar and S. Venkatesan. Techniques to tackle state explosion in global predicate detection. In *Proc. of the International Conference on Parallel and Distributed Systems*, pages 412–417. IEEE, December 1994.

[BA90] M. Ben-Ari. *Principles of concurrent and distributed programming.* Prentice Hall, 1990.

[BFR96] O. Babaoglu, E. Fromentin, and M. Raynal. A unified framework for the specification and run-time detection of dynamic properties in distributed computations. *Journal of Systems and Software*, 33(3):287–298, June 1996.

[BHS+95] David Bailey, Tim Harris, William Saphir, Rob van der Wijngaart, Alex Woo, and Maurice Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, 1995.

[BM93] O. Babaoglu and K. Marzullo. Consistent global states of distributed systems: fundamental concepts and mechanisms. In S. Mullender, editor, *Distributed Systems*, chapter 4. Addison-Wesley, 1993.

[Bou87] L. Bouge. Repeated snapshots in distributed systems with synchronous communication and their implementation in CSP. *Theoretical Computer Science*, 49:145–169, 1987.

[BPA98] R. Biachini, R. Pinto, and C. Amorim. Data prefetching for software DSMs. In *Proceedings of the International Conference on Supercomputing*, pages 385–392, Melbourne, Australia, July 1998. ACM.

[BR94] O. Babaoglu and M. Raynal. Specification and detection of behavioral patterns in distributed computations. In *Proc. of 4th IFIP WG 10.4 International Conference on Dependable Computing for Critical Applications*, San Diego, USA, January 1994.

[BV95] S. Bulgannawar and N. H. Vaidya. A distributed K-mutual exclusion algorithm. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 153–160. IEEE, 1995.

[CC98a] S. Chandra and P. M. Chen. Checkpointing and the fail-stop model. http://www.eecs.umich.edu/˜pmchen/Rio/papers.html, December 1998.

[CC98b] S. Chandra and P. M. Chen. How fail-stop are faulty programs? In *Proc. of the Symposium on Fault-tolerant Computing (FTCS)*, Munich, Germany, June 1998. IEEE.

[CG95] C. M. Chase and V. K. Garg. Efficient detection of restricted classes of global predicates. In *Proc. of the Workshop on Distributed Algorithms*, pages 303–317, Le Mont-Saint-Michel, France, 1995.

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM*, 14(10):667–668, October 1971.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63 – 75, February 1985.

[CL95] R. Cypher and E. Leu. Efficient race detection for message-passing programs with nonblocking sends and receives. In *Proceedings of the Symposium on Parallel and Distributed Processing*, pages 534 – 541. IEEE, 1995.

[CM91] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163 – 173, Santa Cruz, California, 1991.

[Cri91] F. Cristian. Understanding fault-tolerant distributed systems. *CACM*, 34(2):56 – 78, Feb 1991.

[CS93] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *Proc. of the 11th Symp. on Operating System Principles*, pages 44 – 57. ACM, 1993.

[CS98] J. D. Choi and H. Srinivasan. Deterministic replay of Java multi-threaded applications. In *2nd SIGMETRICS Symp. on Parallel and Distr. Tools*, pages 48 – 59, Aug. 1998.

[DTG99] O. P. Damani, A. Tarafdar, and V. K. Garg. Optimistic recovery in multi-threaded distributed systems. In *Proc. of the Symposium on Reliable Distributed Systems (SRDS)*, pages 234 – 243, Lausanne, Switzerland, October 1999. IEEE.

[EAWJ99] E. N. Elnozahy, Lorenzo Alvisi, Y. M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. Technical Report CMU-CS-99-148, Dept. of Computer Science, Carnegie Mellon University, 1999.

[Fid91] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28 – 33, August 1991.

[Fin97] S. A. Fineberg. NAS Application I/O (BTIO) Benchmark. http://parallel.nas.nasa.gov/MPI-IO/btio/index.html, 1997.

[FL97] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *Proc. of 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 22–25, Newport, USA, June 1997.

[FRGT94] E. Fromentin, M. Raynal, V. K. Garg, and A. I. Tomlinson. On the fly testing of regular patterns in distributed computations. In *Proc. of the 23rd International Conference on Parallel Processing*, pages 73–76, St. Charles, USA, August 1994.

[Gar96] V. K. Garg. *Principles of Distributed Systems*. Kluwer Academic Publishers, 1996.

[GC95] V. K. Garg and C. M. Chase. Distributed algorithms for detecting conjunctive predicates. In *Proc. of the 15th International Conference on Distributed Computing Systems*, pages 423–430, Vancouver, Canada, 1995.

[GCKM97] V. K. Garg, C. M. Chase, R. B. Kilgore, and J. R. Mitchell. Efficient detection of channel predicates in distributed systems. *Journal of Parallel and Distributed Computing*, 45(2):134–147, 1997.

[GGL+90] A. P. Goldberg, A. Gopal, K. Li, R. E. Strom, and D. F. Bacon. Transparent recovery of Mach applications. In *USENIX Mach Workshop*, pages 169–183, 1990.

[GL96] W. D. Gropp and E. Lusk. User's guide for mpich, a portable implementation of MPI. Technical Report ANL/MCS-TM-ANL-96/6, Argonne National Laboratory, 1996.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*, chapter 3. Morgan Kaufmann Publishers, 1993.

[GTFR95] V. K. Garg, A. I. Tomlinson, E. Fromentin, and M. Raynal. Expressing and detecting general control flow properties of distributed computations. In *Proc. of the 7th IEEE Symposium on Parallel and Distributed Processing*, pages 432–438, San Antonio, USA, October 1995.

[GW94] V. K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299 – 307, March 1994.

[GW96] V. K. Garg and B. Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1323 – 1333, December 1996.

[HJK93] S.-T. Huang, J.-R. Jiang, and Y.-C. Kuo. k-coteries for fault-tolerant k entries to a critical section. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 74 – 81. IEEE, 1993.

[HJPR87] J.-M. Helary, C. Jard, N. Plouzeau, and M. Raynal. Detection of stable properties in distributed applications. In *Proceedings of the 6th Symposium on Principles of Distributed Computing*, pages 125–136, Vancouver, Canada, 1987. ACM.

[HK93] Y. Huang and C. Kintala. Software implemented fault tolerance: technologies and experience. In *Proc. IEEE Fault-Tolerant Comp. Symp.*, pages 138 – 144, June 1993.

[HMRS96] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient distributed detection of conjunction of local predicates in asynchronous computations. In *Proc. of the 8th International IEEE Symposium on Parallel and Distributed Processing*, pages 589–593, New Orleans, USA, October 1996.

[HPR93] M. Hurfin, N. Plouzeau, and M. Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 32 – 42. ACM/ONR, 1993.

[HR96] M. Hurfin and M. Raynal. Detecting diamond decklaces in labelled directed acyclic graphs (a problem from distributed debugging). In Springer-Verlag, editor, *Proc. of the 22nd International Workshop on Graph Theoretic Concepts in Computer Science, LNCS*, pages 211–223, New Orleans, USA, June 1996.

[HW94] Gudjon Hermannsson and Larry D. Wittie. Optimistic synchronization in distributed shared memory. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 345–354. IEEE, 1994.

[HW96] C. E. Hrischuk and C. M. Woodside. Proper time: Causal and temporal relations of a distributed system. Technical Report SCE-96-04, Systems and Computer Engineering, Carleton University, 1996.

[IL95] R. K. Iyer and I. Lee. Software fault tolerance in computer operating systems. In M. R. Lyu, editor, *Software Fault Tolerance*, Trends in Software Series, chapter 11, pages 249 – 278. John Wiley & Sons, Inc., 1995.

[JJJR94] C. Jard, T. Jeron, G. V. Jourdan, and J. X. Rampon. A general approach to trace-checking in distributed computing systems. In *Proc. of* *the 14th International Conference on Distributed Computing Systems*, pages 396–403, Poznan, Poland, June 1994. IEEE.

[JMN95] R. Jegou, R. Medina, and L. Nourine. Linear space algorithm for on-line detection of global predicates. In Springer-Verlag, editor, *Proc. of the International Workshop on Structures in Concurrency Theory*, 1995.

[KCDZ95] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. An evaluation of software-based release consistent protocols. *Journal of Parallel and Distributed Computing*, 29:126–141, September 1995.

[KN94] D. Kotz and N. Nieuwejaar. Dynamic file-access characteristics of a production parallel scientific workload. In *Proc. Supercomputing '94*, pages 640–649, 1994.

[KS97] M. Karlsson and P. Stenstrom. Effectiveness of dynamic prefetching in multiple-writer distributed virtual shared memory systems. *Journal of Parallel and Distributed Computing*, 43(7):79–93, July 1997.

[Lam77] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558 – 565, July 1978.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computer*, 28(9):690–691, September 1979.

[LMC87] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471 – 482, April 1987.

[Lyu95] M. R. Lyu, editor. *Software Fault Tolerance*. Trends in Software Series. John Wiley & Sons, Inc., 1995.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215 – 226. Elsevier Science Publishers B. V. (North Holland), 1989.

[MBB+92] K. Makki, P. Banta, K. Been, N. Pissinou, and E. Park. A token based distributed k mutual exclusion algorithm. In *Proceedings of the Symposium on Parallel and Distributed Processing*, pages 408 – 411. IEEE, December 1992.

[MC88] B. P. Miller and J. Choi. Breakpoints and halting in distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 316 – 323. IEEE, 1988.

[MCL89] J. M. Mellor-Crummey and T. J. Leblanc. A software instruction counter. In *Proc. of the 3rd Symp. on Architectural Support for Programming Languages and Operating Systems*, pages 78 – 86, April 1989.

[MG98] N. Mittal and V. K. Garg. Consistency conditions for multi-object distributed operations. In *Proceedings of 18th International Conference on Distributed Computing Systems*, pages 582–589, Amsterdam, The Netherlands, May 1998. IEEE.

[MG00] N. Mittal and V. K. Garg. Debugging distributed programs using controlled re-execution. In *Proc. of the 19th ACM Symposium on Principles of Distributed Computing*, pages 239–248, Portland, USA, July 2000.

[MI92] Y. Manabe and M. Imase. Global conditions in debugging distributed programs. *Journal of Parallel and Distributed Computing*, 15:62–69, 1992.

[MSWW81] A. Maggiolo-Schettini, H. Wedde, and J. Winkowski. Modeling a solution for a control problem in distributed systems by restrictions. *Theoretical Computer Science*, 13(1):61 – 83, January 1981.

[Mue99] F. Mueller. Adaptive DSM-runtime behavior via speculative data distribution. In J. Rolim et al., editor, *Parallel and Distributed Computing, Proceedings of the IPPS/SPDP99 Workshops, Workshop on Run-Time Systems for Parallel Programming*, LNCS 1586, pages 553–567. Springer-Verlag, April 1999.

[Net91] R. H. B. Netzer. *Race condition detection for debugging shared-memory parallel programs*. PhD thesis, University of Wisconsin-Madison, 1991.

[Net93] R. H. B. Netzer. Optimal tracing and replay for debugging shared-memory parallel programs. In *Proc. of ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 1 – 11, San Diego, USA, May 1993. Also available as ACM SIGPLAN Notices Vol. 28, No. 12.

[NM92] R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Supercomputing '92*, pages 502 – 511, November 1992.

[Pan95] C. M. Pancake. The emperor has no clothes: What HPC users need to say and HPC vendors need to hear. In *Supercomputing*, San Diego, USA, December 1995. ACM/IEEE.

[Pra86] V. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33 – 71, 1986.

[Ran75] B. Randell. System structure for software fault-tolerance. *IEEE Transactions on Software Engineering*, 1(2):220 – 232, June 1975.

[Rao99] S. S. Rao. *Egida: A Toolkit for Low-overhead Fault-tolerance*. PhD thesis, Univ. of Texas at Austin, 1999.

[Ray86] M. Raynal. *Algorithms for Mutual Exclusion*. MIT Press, 1986.

[Ray88] M. Raynal. *Distributed Algorithms and Protocols*. John Wiley and Sons Ltd., 1988.

[Ray89] K. Raymond. A distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 30:189–193, February 1989.

[RC96] M. Russinovich and B. Cogswell. Replay for concurrent non-deterministic shared-memory applications. In *Proc. ACM SIGPLAN Conf. on Programming Languages and Implementation (PLDI)*, pages 258–266, 1996.

[RZ97] M. Ronnse and W. Zwaenepoel. Execution replay for TreadMarks. In *Proc. of the 5th EUROMICRO Workshop on Parallel and Distributed Processing (PDP'97)*, pages 343–350, January 1997.

[SBN+97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multi-threaded programs. In *Proc. of the 16th ACM Symposium on Operating System Principles*, pages 27 – 37, Saint-Malo, France, October 1997.

[SE96] J. H. Slye and E. N. Elnozahy. Supporting nondeterministic execution in fault-tolerant systems. In *Proc. 26th Fault Tolerant Computing Symposium*, pages 250–259, 1996.

[Sin93] M. Singhal. A taxonomy of distributed mutual exclusion. *Journal of Parallel and Distributed Computing*, 18:94 – 101, 1993.

[SK86] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *Proc. of the 6th International Conference on Distributed Computing Systems*. IEEE, 1986.

[SM94] R. Schwartz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[Smi81] J. E. Smith. A study of branch prediction strategies. In *Proc. of the 8th International Symposium on Computer Architecture*, pages 135–148, May 1981.

[SP98] J. Stark and Y. Patt. Variable length path branch prediction. In *Architectural Support for Programming Languages and Operating Systems*, pages 170–179, 1998.

[SR92] P. K. Srimani and R. L. Reddy. Another distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 41:51–57, January 1992.

[Tai97] K. Tai. Race analysis of traces of asynchronous message-passing programs. In *Proceedings of the 17th International Conference on Distributed Computing Systems*, pages 261 – 268. IEEE, 1997.

[TCO91] K. C. Tai, R. H. Carver, and E. E. Obaid. Debugging concurrent ADA programs by deterministic execution. *IEEE Transactions on Software Engineering*, 17(1):45–63, January 1991.

[TG93] A. I. Tomlinson and V. K. Garg. Detecting relational global predicates in distributed systems. In *Proc. of ACM/ONR Workshop on Parallel* *and Distributed Debugging*, pages 21–31, San Diego, USA, May 1993. Also available as ACM SIGPLAN Notices Vol. 28, No. 12.

[TG94] A. I. Tomlinson and V. K. Garg. Maintaining global assertions on distributed sytems. In *Computer Systems and Education*, pages 257 – 272. Tata McGraw-Hill Publishing Company Limited, 1994.

[TG95] A. I. Tomlinson and V. K. Garg. Observation of software for distributed systems with rcl. In Springer-Verlag, editor, *Proc. of the 15th Conference on the Foundations of Software Technology and Theoretical Computer Science, LNCS*, December 1995.

[TG98a] A. Tarafdar and V. K. Garg. Addressing false causality while detecting predicates in distributed programs. In *Proc. of the 18th International Conference on Distributed Computing Systems*, pages 94 – 101, Amsterdam, The Netherlands, May 1998.

[TG98b] A. Tarafdar and V. K. Garg. Happened before is the wrong model for potential causality. Technical Report ECE-PDS-1998-006, Parallel and Distributed Systems Laboratory, ECE Dept, University of Texas at Austin, 1998.

[TG98c] A. Tarafdar and V. K. Garg. Predicate control for active debugging of distributed programs. In *Proc. of the 9th Symposium on Parallel and Distributed Processing*, pages 763–769, Orlando, USA, April 1998. IEEE.

[TG99] A. Tarafdar and V. K. Garg. Software fault tolerance of concurrent programs using controlled re-execution. In *Proc. of the 13th International Symposium on Distributed Computing*, pages 210–224, Bratislava, Slovak Republic, September 1999.

[TRA96] F. J. Torres-Rojas and M. Ahamad. Plausible clocks: Constant size logical clocks for distributed systems. In *Proc. of the Workshop on Distributed Algorithms*, Bologna, Italy, October 1996.

[VD95] S. Venkatesan and B. Dathan. Testing and debugging distributed programs distributively. *IEEE Transactions on Software Engineering*, 21(2):163–177, 1995.

[Vel93] Martin G. Velazquez. A survey of distributed mutual exclusion algorithms. Technical Report 93-116, Computer Science Dept., Colorado State University, 1993.

[WHF+97] Y. M. Wang, Y. Huang, W. K. Fuchs, C. Kintala, and G. Suri. Progressive retry for software failure recovery in message-passing applications. *IEEE Trans. on Computers*, 46(10):1137–1141, October 1997.

Vita

Ashis Tarafdar was born on February 11, 1973 in Howrah, India, the son of Anima Tarafdar and Sankar Prosad Tarafdar. He completed his schooling in Bombay, India in June 1990. He received the Bachelor of Technology degree in Computer Science and Engineering from the Indian Institute of Technology at Bombay in July 1994. Thereafter, he joined graduate school at the University of Texas at Austin where he received the Master of Sciences degree in Computer Sciences in December 1996. He has been employed at Microsoft Corporation, IBM Almaden Research Center, and IBM T. J. Watson Research Center during the summers of 1995, 1996, and 1999, respectively. He was awarded the MCD graduate fellowship in January 1995 by the University of Texas at Austin.

Permanent Address: c/o Mrs. Anima Tarafdar
606 Vigyan, Scientists' CHS, Sector-17,
Vashi, New Mumbai - 400703, India

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.