The Dissertation Committee for Neeraj Mittal

Certifies that this is the approved version of the following dissertation:

Techniques for Analyzing Distributed Computations

Committee:

Vijay K. Garg, Supervisor

Anish Arora

Craig M. Chase

Mohamed G. Gouda

Aloysius K. Mok

Harrick Vin

Techniques for Analyzing Distributed Computations

by

Neeraj Mittal, 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

May 2002

To my parents

Acknowledgments

I consider myself to have been an extremely fortunate Ph.D. student in having Vijay K. 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 Om Damani and Ashis Tarafdar for discussions early in my Ph.D., which inspired me to follow up on my early ideas. Alper Sen has helped me at various times by reviewing my papers and providing me with valuable criticism that has greatly improved my work. My committee members, Mohamed G. Gouda, Anish Arora, Aloysius K. Mok, Craig M. Chase, and Harrick Vin, 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. Fun evenings, weekend trekking trips, late-night discussions, intense workouts are but some of the memorable experiences that I have variously shared with Subramanyam Gooty, Santanu Sinha, Vipin Gupta, Gokul Rajaram, Ravi P.

Bulusu, Praveen K. Jaini, Parminder S. Chhabra, Vineet Kahlon, and Subramanian Iyer. The biggest credit goes to my family for their love and support. My parents, Suresh C. Mittal and Shakuntla Mittal, have kept me going by their never-ending confidence in me. My sisters, Amita Gupta, Vanita Gupta, and Neelima Kumar, have helped me with practical advice at various stages. Without my family, this dissertation would never have been written.

NEERAJ MITTAL

The University of Texas at Austin
May 2002

Techniques for Analyzing Distributed Computations

Publication No. _______________

Neeraj Mittal, Ph.D.
The University of Texas at Austin, 2002

Supervisor: Vijay K. Garg

Inherent non-determinism in distributed programs and presence of multiple threads of control makes it difficult to write correct distributed software. Not surprisingly, distributed systems are particularly vulnerable to software faults. To build a distributed system capable of tolerating software faults, two important problems need to be addressed: *fault detection* and *fault recovery*.

The fault detection problem requires finding a (consistent) global state of the computation that satisfies certain predicate (*e.g.*, violation of mutual exclusion). To prevent a fault from causing any serious damage such as corrupting stable storage, it is essential that it be detected in a timely manner. However, we prove that detecting a predicate in 2-CNF, even when no two clauses contain variables from the same process, is an NP-complete problem. We develop a technique, based on *computation slicing*, to reduce the size of the computation and thus the number of global states to be examined for detecting a predicate. Slicing can be used to throw away the *extraneous* global states of the computation in an efficient manner, and focus on only those that are currently *relevant* for our purpose. To detect a fault, therefore, rather than searching the state-space of the computation, it is much more efficient

to search the state-space of the slice. We identify several useful classes of predicates for which the slice can be computed efficiently. Our experimental results indicate that slicing can lead to an exponential reduction over existing techniques both in terms of time as well as space for fault detection.

To recover from faults, we consider *rollback recovery* approach, which involves restoring the system to a previous state and then re-executing. We focus on rollback recovery using *controlled re-execution*, which is useful and effective for tolerating *synchronization faults*. Unlike other approaches which depend on chance and do not ensure that the re-execution is fault-free, the controlled re-execution method avoids synchronization faults during re-execution in a deterministic fashion. Specifically, it selectively adds synchronization dependencies during re-execution to ensure that the previously detected synchronization faults do not occur again. We provide efficient algorithms to solve the problem for two important classes of synchronization faults.

Contents

Chapter 1

Introduction

Recent advances in communication technology have led to a rapid proliferation of distributed systems. For example, a cluster of servers provided Web coverage of the Sydney Summer Olympics. As another example, mass-distributed computing was recently used to discover the largest known prime number. As distributed systems evolve from the special case to commonplace, ensuring their reliable operation has emerged as an important and challenging problem. With distributed systems being increasingly employed in safety-critical environments, a failure in one of these systems could have irreparable, if not tragic, consequences. There have been several examples of serious systems failures (*e.g.*, Ariane 5, Therac 25, Mars Observer) caused at least in part by critical defects in the software.

Inherent non-determinism in distributed programs and presence of multiple threads of control make it difficult to write correct distributed software. Not surprisingly, distributed systems are especially vulnerable to software faults. Dealing with software faults requires efforts at multiple levels [TP00]. Early in the software cycle, design methodologies, technologies and techniques that are aimed at preventing the introduction of faults into the design can be used (*fault prevention*). Later, the implementation can be verified using testing, and the faults thereby exposed can be removed using debugging (*fault removal*). In spite of extensive testing and debugging, software faults may persist even in production quality software. *Fault tolerance* can be used as an extra layer of protection to provide acceptable level of performance and safety at runtime after a fault becomes active. In this dissertation, we focus on fault removal and fault tolerance techniques to improve the reliability of distributed software.

Fault Removal

The correctness of a program is often expressed using a combination of *safety* and *liveness* properties. A safety property specifies what the program must not do (ensures "nothing *bad* will ever happen"). An example of a safety property is mutual exclusion which demands that at no time should there be more than one process in its critical section. A liveness property, on the other hand, specifies what the program must eventually do (guarantees "something *good* will eventually happen"). An example of a liveness property is that every process which is trying to acquire a resource will succeed eventually.

Testing and debugging has been widely used for developing traditional sequential programs. *Testing* involves executing the program for a specific input sequence and then validating the output obtained with respect to the given safety and liveness properties. Specifically, when testing for safety property, the objective is to verify that the system always stayed in a safe state throughout the execution, or, in other words, the system did not traverse through an unsafe state. Similarly, when testing for liveness property, the aim is to ascertain that some desired condition eventually became true in the execution. In case testing reveals that the program behaved erroneously (it violated either safety or liveness property), *debugging* is the process of tracking down the bug that caused the program to exhibit the faulty behaviour.

The state of a distributed system, commonly referred to as *global state*, is given by the set of events that have been executed so far (on all processes). In an asynchronous distributed system, however, it is not possible for an external observer to determine the exact order in which the events generated by the system were executed in real-time. The events can only be partially ordered; the partial order is referred to as the *Lamport's happened-before relation* [Lam78] and the corresponding partially ordered set (or poset) is called a *distributed computation*. Each interleaving of events that respects the happened-before relation corresponds to an order in which the events could have been executed. Testing a computation with respect to safety and liveness properties, therefore, translates into answering the following queries: "Does there exist an interleaving of events in which the system passes through an unsafe global state?" and "Does a liveness property eventually become true in all possible interleavings of events?" The two problems correspond to the *predicate detection* problem under *possibly* and *definitely* modalities [CM91, GW91], respectively.

On discovering a fault in the computation during testing phase, the next step is to analyze the computation to locate the source of the fault. While the skill and intuition of the programmer play an important role in debugging, tools that provide an effective environment for debugging are indispensable. For example, on detecting a violation of safety property, a programmer can gain considerable insight into the bug, that caused the violation, by learning whether all possible interleavings of events are unsafe in the sense that they all pass through a global state that is unsafe. In that case, the bug cannot be fixed by adding or removing synchronization alone. On the other hand, if it is possible to eliminate all unsafe interleavings by adding synchronization to the computation, without creating a deadlock, then *too* *little* synchronization is likely to be the problem. Furthermore, the knowledge of the exact synchronization needed to maintain a safety property can facilitate the localization of the bug in the program. The problem of finding a synchronization required to maintain a safety property in a computation is referred to as the *predicate control* problem [TG98b].

Analyzing an erroneous computation in order to track down the source of the fault is complicated by the fact that the computation in general contains exponential number of global states. Therefore it is helpful and desirable to focus on only those global states that are likely to be involved in the fault. For example, to locate the bug, it may suffice to examine only transitless global states, the ones in which all sent messages have been received. To that end, we define the notion of *computation slice*. Intuitively, slice is a concise representation of those global states of the computation that satisfy certain property. More precisely, the slice of a computation with respect to a predicate is the computation satisfying the following two conditions. First, it contains *all* global states for which the predicate evaluates to true. Second, among all computations that fulfill the first condition, it contains the *least* number of global states. A slice may contain exponentially fewer number of global states than the computation, thereby substantially reducing the size of the computation to be analyzed.

Fault Tolerance

A production quality software which has been extensively tested and debugged contains around 3 bugs per 1,000 lines of code [GR93]. Many systems, especially those employed in safety-critical environments, should be able to operate properly even in the presence of these bugs. An overwhelming majority of the bugs tend to be non-deterministic in nature and are often caused by transient conditions such as timing and synchronization. Therefore they do not manifest themselves in every

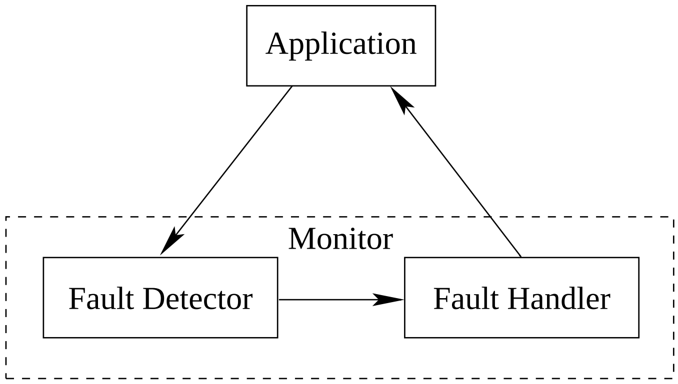


Figure 1.1: A software fault tolerance system.

program execution with the same input sequence and it is possible to tolerate them at runtime using *rollback recovery* [GR93]. A system capable of tolerating software faults can be built using a monitor that continuously observes the system execution to detect an occurrence of a fault. On detecting a fault, it rolls back the program to a state before the fault occurred and re-executes it hoping that the previously detected fault does not occur again. To prevent the fault from causing any serious damage such as corrupting stable storage, it is essential that the monitor be able to detect the fault in a timely manner. This requires the fault detection algorithm to be fast and efficient. Further, to minimize the disruption in service caused by the fault, it is desirable that during re-execution the fault be avoided in a deterministic fashion instead of relying on chance [WHF+97]. Tarafdar and Garg [TG99] proposed the *controlled re-execution* approach which assumes some knowledge about the fault (*e.g.*, fault occurred because of improper synchronization) but provides a guarantee that the previously detected fault will not recur during re-execution. Therefore to build a software fault tolerant system, two issues need to be addressed: (1) fault detection which gives rise to the problem of detecting a predicate under *possibly* modality, and (2) fault recovery which in the case of synchronization faults gives rise to the predicate control problem.

To summarize, our goals are:

- To investigate the problem of detecting a predicate in a computation.
- To investigate the problem of controlling a predicate in a computation.
- To formulate and investigate the notion of slice of a computation with respect to a predicate.

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

1.1 Detecting Global Predicates

Verifying the correctness of an observed behaviour of a program, for a specific input sequence, gives rise to the problem of detecting a predicate in a computation under *possibly* and *definitely* modalities. When detecting a predicate under *possibly* modality, the objective is to find a global state in the computation that violates the safety property. For example, consider the computation in Figure 1.2 with three processes p_1, p_2 and p_3. The safety property is mutual exclusion which demands that no two processes are in critical sections (labeled CS_1, CS_2, CS_3 and CS_4) at the same time. Clearly, the given computation does not maintain mutual exclusion at all times. Specifically, mutual exclusion is violated for global state C in which processes p_1 and p_3 are in their respective critical sections.

Detecting a predicate under *definitely* modality requires verifying that the liveness property eventually becomes true in all interleavings of events. For example, consider the computation in Figure 1.3 with two processes p_1 and p_2. The liveness property requires that the system always passes through a state in which both

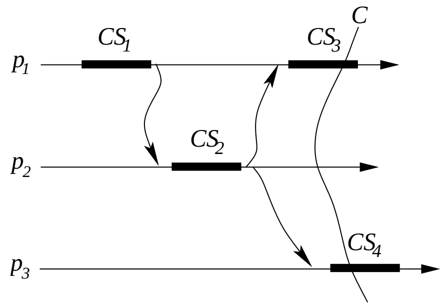


Figure 1.2: Detecting a predicate under *possibly* modality.

processes are in the second round. Clearly, if the events are interleaved in the order $a\ e\ b\ c\ f\ d\ g\ h$, the desired condition never becomes true.

It can be proved that detecting a predicate in a computation under *definitely* modality is the dual of controlling a predicate in a computation. Thus our results in solving the predicate control problem are applicable to the predicate detection problem under *definitely* modality as well. Hereafter, the default modality for predicate detection is *possibly*. Moreover, we do not specify *possibly* modality unless we need to distinguish it from *definitely* modality.

Contributions

It is always useful and desirable to know for what classes of predicates an efficient polynomial-time detection algorithm is unlikely to exist. To that end, Chase and Garg prove in [CG95] that detecting a predicate in 3-CNF is an NP-complete problem. Also, Stoller and Schneider [SS95] show that it is computationally hard to detect a 2-local conjunctive predicate (a predicate expressed as conjunction of clauses where each clause depends on variables of at most two processes). We demonstrate that detecting a predicate in 2-CNF even when no two clauses contain variables from the same process is an NP-complete problem as well. It may be noted that our intractability result subsumes the two aforementioned NP-completeness results. Nevertheless, computation slicing, discussed later, can be used to achieve

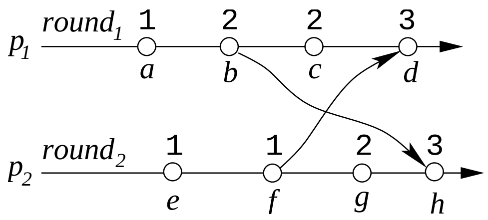


Figure 1.3: Detecting a predicate under *definitely* modality.

an exponential improvement in time as well as space for detecting a predicate that is otherwise computationally hard to detect.

Additionally, we establish that detecting a relational predicate of the form $x_1 + x_2 + \cdots + x_n = k$ for constant k, where x_i is an integer variable on process p_i, is an NP-complete problem. This is somewhat surprising because a relational predicate of the form $x_1+x_2+\cdots+x_n \leqslant k$, for constant k, can be detected efficiently. (This is true even when $\leqslant$ is replaced with $\geqslant$.) However, for certain restricted but useful class of general computations, it is indeed possible to provide an efficient polynomial-time algorithm to detect the former relational predicate. This class corresponds to computations in which each x_i is incremented or decremented by at most one at each step. Such computations are generated, for example, when each x_i is a binary variable and can assume values 0 or 1. As a corollary, any symmetric predicate—predicate composed from boolean variables that is invariable under any permutation of its variables—can be efficiently detected.

1.2 Controlling Global Predicates

The problem of controlling a predicate in a computation involves adding synchronization to the computation, without creating a cycle, such that the given predicate is never falsified in the resultant computation. As an example, consider the computation in Figure 1.4(a) with three processes p_1, p_2 and p_3. Suppose the stated predicate is the mutual exclusion predicate which requires that no two processes are

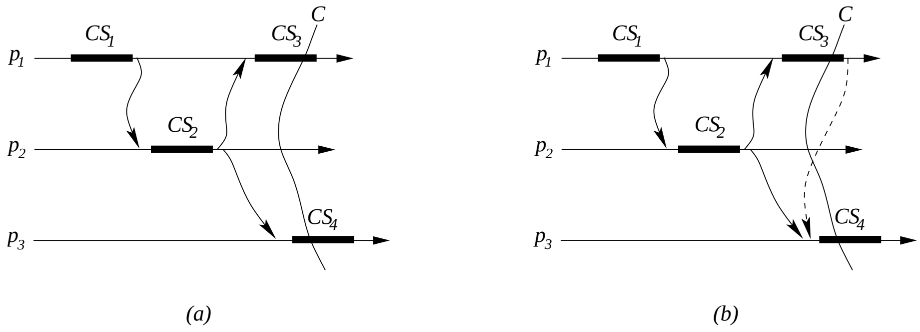


Figure 1.4: Controlling a predicate in a computation: (a) original computation, and (b) controlled computation.

in critical sections (labeled CS_1, CS_2, CS_3 and CS_4) at the same time. Clearly, the computation does not maintain mutual exclusion at all times. Figure 1.4(b) depicts the same computation with added synchronization that ensures that mutual exclusion is maintained at all times. We call such a computation as "controlled computation" and the added synchronization as "controlling synchronization". The main difficulty in determining such a controlling synchronization lies in adding the synchronization dependencies in such a manner as to maintain the given property without causing deadlock with existing synchronization dependencies.

Contributions

Tarafdar and Garg prove in [TG98b] that it is in general NP-complete to compute a controlling synchronization for a predicate. We therefore focus on two useful classes of predicates for which polynomial-time algorithms can be provided.

The first class of predicates we consider is the class of "region predicates". Informally, a region predicate partitions the set of global states of the computation that satisfy the predicate into bounded convex regions, one for each event. Some examples of region predicates include "the virtual clocks of all processes are approximately synchronized", and channel predicates such as "all request messages have been received". We give an $O(n|E|^2)$ algorithm for computing a controlling synchronization for a region predicate, where n is the number of processes and E is the set of events. We also show that the controlling synchronization generated by the algorithm is *optimal* in the sense that it permits the maximum possible concurrency in the controlled computation.

The other class of predicates we study is the class of "disjunctive predicates". A disjunctive predicate can be expressed as disjunction of local predicates. Some examples include "at least one server is not busy", "at least one philosopher does not have a fork", and $(n-1)$-mutual exclusion with n processes in the system. Intuitively, a disjunctive predicate ensures that a bad combination of local conditions does not occur. We provide an $O(n|E|)$ algorithm for computing a controlling synchronization for a disjunctive predicate, where n is the number of processes and E is the set of events. We further modify the algorithm to compute a controlling synchronization with the *least* number of synchronization dependencies. The modified algorithm has $O(|E|^2)$ time-complexity.

1.3 Slicing Distributed Computations

The slice of a computation with respect to a predicate is the computation with the least number of global states such that it contains all global states of the original computation satisfying the given predicate. As an illustration, consider the computation in Figure 1.5(a). In the figure, the first event on each process initializes the state of the process. The initial global state is therefore obtained by executing the events a, e and u. Suppose we wish to examine only those global states for which $(x_1 \geqslant 1) \wedge (x_3 \leqslant 3)$. A concise representation of such global states—referred to as slice—is shown in Figure 1.5(b). Informally, in the slice, the partial order is specified on subsets of events rather than events. Intuitively, all events in a subset are executed *atomically*, that is, either none of them is executed or all of them are

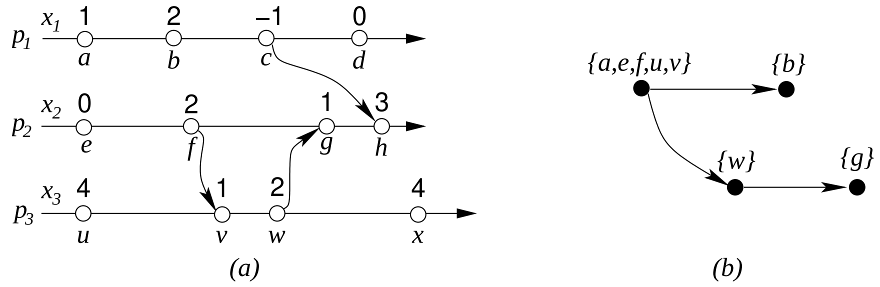


Figure 1.5: (a) A computation, and (b) its slice with respect to the predicate $(x_1 \geqslant 1) \wedge (x_3 \leqslant 3)$.

executed. For instance, the global state of the computation obtained by executing the events a, b, e and u is not a global state of the slice because only some of the events in the subset $\{a, e, f, u, v\}$ have been executed. The computation contains twenty eight global states whereas the slice contains only six global states.

Now, suppose we want to find a global state of the computation for which the predicate $(x_1 * x_2 + x_3 < 5) \wedge (x_1 \geqslant 1) \wedge (x_3 \leqslant 3)$ evaluates to true. Without computation slicing, we are forced to examine all global states of the computation to ascertain whether some global state satisfies the predicate. With computation slicing, however, we can restrict our search to the global states of the slice, thereby resulting in substantial savings.

Contributions

We first establish that slice exists and is uniquely defined for all predicates. The slice for a predicate may contain global states that do not satisfy the predicate. We identify the class of "regular predicates" for which the slice is "lean". In other words, the slice for a regular predicate contains precisely those global states for which the predicate evaluates to true. The set of global states that satisfy a regular predicate forms a sublattice, that is, it is closed under intersection and union. Some examples of regular predicates are: conjunction of local predicates like "no process has the token", and channel predicates such as "all request messages have been received". We prove that the class of regular predicates is closed under conjunction, that is, the conjunction of two regular predicates is also a regular predicate. We devise an efficient algorithm to compute the slice for a regular predicate. The time-complexity of the algorithm is $O(n^2|E|)$, where n is the number of processes and E is the set of events. Additionally, for special cases of regular predicates such as conjunction of local predicates, we develop *optimal* algorithms for computing the slice which have $O(|E|)$ time-complexity. In addition to regular predicates, we also provide efficient algorithms to compute the slice for many classes of non-regular predicates including "linear predicates" and "post-linear predicates".

We prove that it is intractable in general to compute the slice for a predicate. Nonetheless, it is still useful to be able to compute an approximate slice for such a predicate efficiently. An approximate slice may be bigger than the actual slice but will be much smaller than the computation itself. To that end, we develop efficient algorithms to compose two slices using "grafting". Specifically, given two slices, grafting involves computing either (1) the smallest slice that contains all global states common to both the slices, or (2) the smallest slice that contains all global states that belong to at least one of the slices. We apply grafting to efficiently compute the slice for the complement of a regular predicate—referred to as "co-regular predicate". The algorithm has $O(n^2|E|^2)$ time-complexity, where n is the number of processes and E is the set of events. We also employ grafting to compute the slice for a "k-local predicate" with constant k in polynomial-time. More importantly, we use grafting to compute an approximate slice—in polynomial-time—for a predicate composed using $\wedge$ and $\vee$ operators from predicates for which the slice can be computed efficiently (*e.g.*, regular predicates, linear predicates). Example of such predicate is: $(x_1 \vee \neg x_2) \wedge (x_3 \vee \neg x_1) \wedge (x_2 \vee x_3)$, where each x_i is a linear predicate. We conduct simulation tests to experimentally measure the effectiveness of computation slicing in pruning the search space when detecting a predicate. Our results indicate that computation slicing can lead to an exponential reduction over existing techniques both in terms of time as well as space.

1.4 Overview of the Dissertation

The remainder of this dissertation is organized as follows. In Chapter 2, we define our model. Next, we have three main chapters of the dissertation. Chapter 3 discusses our results in detecting global predicates, Chapter 4 investigates the problem of controlling global predicates, and Chapter 5 describes our study of the computation slicing technique. In Chapter 6, we give a summary of the related work. Finally, we draw conclusions and describe future directions in Chapter 7.

Chapter 2

System Model

In this chapter we formally describe the model and notation used in this dissertation. Our model is based on the Lamport's *happened-before* model [Lam78]. The model is further extended in Chapter 5 where we discuss computation slicing in detail.

2.1 Distributed Computations

We assume an asynchronous distributed system with the set of processes $P = \{p_1, p_2, \ldots, p_n\}$. Each process executes a predefined program. Processes do not share any clock or memory; they communicate and synchronize with each other by sending messages over a set of channels. We assume that channels are reliable, that is, messages are not lost, altered or spuriously introduced into a channel. We do not assume FIFO channels.

The *local computation* of a process is given by the sequence of events that transforms the *initial state* of the process into the *final state*. At each step, the *local state* is captured by the initial state together with the sequence of events that have been executed up to that step. Each event is either an *interval event* or an *external event*. An external event could be a *send event* or a *receive event* or both. An event causes the local state of a process to be updated. Additionally, a send event causes a message or a set of messages to be sent and a receive event causes a message or a set of messages to be received. We assume the presence of fictitious *initial events* on each process p_i, denoted by $\bot_i$. The initial event occurs before any other event on the process and initializes the state of that process. We denote the last event on process p_i, called the *final event*, by $\top_i$. Let $\bot$ and $\top$ denote the set of all initial events and final events, respectively.

Let $proc(e)$ denote the process on which event e occurs. The predecessor and successor events of e on $proc(e)$ are denoted by $pred(e)$ and $succ(e)$, respectively, if they exist. Observe that an initial event does not have a predecessor and a final event does not have a successor.

We model a *distributed computation* (or simply a *computation*) by an irreflexive partial order on a set of events. We use $\langle E, \rightarrow \rangle$ to denote a distributed computation with the set of events E and the partial order $\rightarrow$. The partial order $\rightarrow$ is given by the Lamport's *happened-before relation* (or *causality relation*) [Lam78] which is defined as the smallest transitive relation satisfying the following properties:

1. if events e and f occur on the same process, and e occurred before f in real time then e happened-before f, and

2. if events e and f correspond to the send and receive, respectively, of a message then e happened-before f.

Given a computation $\langle E, \rightarrow \rangle$, we denote the order of events on processes by $\xrightarrow{P}$ which is referred to as *process order*. Note that the projection of $\xrightarrow{P}$ onto the events of a single process is a total order. The reflexive closure of an irreflexive partial order $\leadsto$ is represented by $\underline{\leadsto}$ and its transitive closure is denoted by $\leadsto^{+}$. A

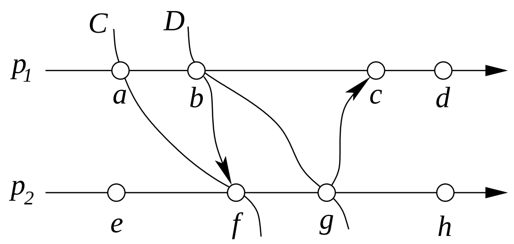


Figure 2.1: An example of a computation.

run or *interleaving* of a computation $\langle E, \rightarrow \rangle$ is some total order on events E that is consistent with the partial order $\rightarrow$.

Example 2.1 *Figure 2.1 depicts a computation involving two processes, namely p_1 and p_2. The local computation of each process advances from left to right as shown in the figure. The circles represent events and the arrows denote messages. The local computation of p_1 is given by the sequence abcd. The event b is a send event, the event f is a receive event and the event d is an internal event. Here, $\bot_1 = a$ and $\bot_2 = e$ whereas $\top_1 = c$ and $\top_2 = h$. Also, $proc(b) = p_1$, $pred(b) = a$ and $succ(e) = c$. The set of events $E = \{a, b, c, d, e, f, g, h\}$ and the happened-before order $\rightarrow = \{(a, b), (b, c), (c, d), (e, f), (f, g), (g, h), (b, f), (g, c)\}^+$. The process order $\xrightarrow{P}$ is given by $\{(a, b), (b, c), (c, d), (e, f), (f, g), (g, h)\}^+$. Finally, aebfghcd is a run of the computation.*

2.2 Cuts, Consistent Cuts and Frontiers

The state of a distributed system, called the *global state*, is given by the collective state of processes. The equivalent notion based on events is called *cut* and is defined as a subset of events that contains all initial events such that it contains an event only if its predecessor, if it exists, also belongs to the subset. Formally,

$$C \text{ is a cut } \triangleq (\bot \subseteq C) \wedge \langle \forall\, e : e \in C : e \notin \bot \Rightarrow pred(e) \in C \rangle$$

The *frontier* of a cut C is defined as the set of those events in C whose successors are not in C. Formally,

$$frontier(C) \;\triangleq\; \{\, e \in C \mid e \not\in \top \;\Rightarrow\; succ(e) \not\in C \,\}$$

We say that a cut *passes through* an event if the event is included in its frontier. Not every cut can occur during system execution. A cut is said to be *consistent* if it contains an event only if it also contains all events that happened-before it. Formally,

$$C \text{ is a consistent cut} \;\triangleq\; (C \text{ is a cut}) \wedge \langle \forall\, e, f : e \rightarrow f : f \in C \;\Rightarrow\; e \in C \rangle$$

In particular, only those cuts which are consistent can possibly occur during an execution. The equivalent notion based on state is called *consistent global state.* We denote the set of consistent cuts of a computation $\langle E, \rightarrow \rangle$ by $\mathcal{C}(\langle E, \rightarrow \rangle)$.

Two events are *consistent* if there exists a consistent cut that passes through both the events, otherwise they are *inconsistent.* It can be verified that events e and f are inconsistent if and only if either $succ(e)\underline{\rightarrow} f$ or $succ(f)\underline{\rightarrow} e$. Finally, two events e and f are *independent* if they are incomparable with respect to $\rightarrow$.

Example 2.2 *Consider the computation in Figure 2.1. Pictorially, we represent a cut by a line drawn from top to bottom passing through exactly one event on each process; an event belongs to the cut if and only if it either lies on the line or lies on the left of the line. The cut* $C = \{a, e, f\}$*. The cut* D *is consistent whereas* C *is not. Here,* $frontier(C) = \{a, f\}$ *and* $frontier(D) = \{b, g\}$*. The events* b *and* f *are consistent whereas events* a *and* f *are not. Finally, events* c *and* h *are independent but* b *and* f *are not.*

2.3 Global Predicates

A *global predicate* (or simply a *predicate*) is defined as a boolean-valued function on variables of processes. Given a consistent cut, a predicate is evaluated with respect to the values of variables resulting after executing all events in the cut. If a predicate b evaluates to true for a consistent cut C, we say that "C satisfies b" and denote it by $C \models b$.

A global predicate is *local* if it depends on variables of a single process. Note that it is possible to evaluate a local predicate with respect to an event on the appropriate process. In case the predicate evaluates to true, the event is called a *true event*; otherwise, it is called a *false event*. We use $e \models b$ to denote the fact that the event e satisfies the local predicate b.

A run is called *safe* with respect to a predicate if every consistent cut of the run satisfies the predicate; otherwise, the run is *unsafe*.

Remark 2.1 *We assume that the time-complexity of evaluating a predicate for a consistent cut is polynomial in input size. However, for convenience, throughout this dissertation, we specify the time-complexity of our algorithms assuming that the time-complexity of evaluating a predicate is linear in number of processes whose variables the predicate depends on. In case the time-complexity is actually higher, the time-complexity of the algorithms will increase correspondingly.*

The value of a predicate is defined with respect to a consistent cut. So, what does it mean to evaluate a predicate for a computation which may consist of several consistent cuts? Given a computation, it is possible to evaluate a predicate under various modalities, namely *possibly*, *definitely*, *invariant* and *controllable* [CM91, WG91, SUL00, TG99, MG00]. A predicate is said to be *possibly* true in a computation if there exists a consistent cut of the computation for which the predicate evaluates to true. On the other hand, a predicate *definitely* holds in a computation if it eventually becomes true in all possible runs of the computation. The modalities *invariant* and *controllable* are duals of the modalities *possibly* and *definitely*, respectively. That is, a predicate is *invariant* in a computation if every consistent cut of the computation satisfies the predicate, whereas it is *controllable*

Concept	Description	Notation
local computation	sequence of events on a process	
distributed computation (or simply computation)	irreflexive partial order on set of events	$\langle E, \rightarrow \rangle$
run/interleaving	total order on events consistent with the partial order of a distributed computation	
process order	order of events on processes	$\overset{P}{\rightarrow}$
cut	an event is in the cut only if its predecessor is also in the cut	C, D
frontier	subset of events in the cut whose successors do not belong to the cut	$\mathit{frontier}(C)$
passes through	event is contained in the frontier of the cut	
consistent cut	an event is in the cut only if all its preceding events (with respect to the partial order) are also in the cut	
consistent events	some consistent cut passes through both the events	
independent events	events are incomparable with respect to the given partial order	
global predicate (or simply predicate)	boolean-valued function on variables of processes	b
safe run	every consistent of the run satisfies the global predicate	
local predicate	global predicate that depends on variables of only a single process	
true event	event satisfies the local predicate	

Table 2.1: A summary of the various concepts.

Notation	Description
$proc(e)$	process on which event e occurs
$pred(e)$	predecessor of event e (on $proc(e)$)
$succ(e)$	successor of event e (on $proc(e)$)
$\bot_i$	initial event on process p_i
$\top_i$	final event on process p_i
$\bot$	set of initial events
$\top$	set of final events
$\rightarrow, \rightsquigarrow, \mapsto$	irreflexive partial orders on set of events
$\underline{\rightarrow}$	reflexive closure of $\rightarrow$
R^+	transitive closure of relation R
$C \models b$	consistent cut C satisfies global predicate b
$e \models b$	event e satisfies local predicate b
$\langle E, \rightarrow \rangle \models modal : b$	global predicate b holds in distributed computation $\langle E, \rightarrow \rangle$ under *modal* modality $modal \in \{possibly, definitely, controllable, invariant\}$

Table 2.2: A summary of the notation.

in a computation if there exists a safe run of the computation with respect to the predicate. The *predicate detection problem* [CM91, CG98, SUL00, MG01b] typically refers to monitoring a predicate under *possibly* (and sometimes under *definitely*) modality, whereas the *predicate control problem* [TG98b, TG99, MG00] involves monitoring a predicate under *controllable* modality.

Given a predicate b and a computation $\langle E, \rightarrow \rangle$, we use $\langle E, \rightarrow \rangle \models possibly : b$ to denote the fact that b possibly holds in $\langle E, \rightarrow \rangle$. The expressions $\langle E, \rightarrow \rangle \models definitely : b$, $\langle E, \rightarrow \rangle \models invariant : b$ and $\langle E, \rightarrow \rangle \models controllable : b$ can be similarly interpreted.

Table 2.1 and Table 2.2 summarize various notations and concepts defined in this chapter.

Chapter 3

Detecting Global Predicates

In this chapter, we describe in detail our results pertaining to the detection of global predicates in distributed computations primarily under *possibly* modality. In particular, we provide solutions to all the open problems proposed in [Gar97].

3.1 Overview

We start by defining the problem formally in Section 3.2. Informally, the problem of detecting a predicate typically refers to monitoring it under *possibly* or *definitely* modality.

Chase and Garg [CG95] prove that it is in general NP-complete to detect a 3-CNF predicate under *possibly* modality. Stoller and Schneider [SS95] show that detecting a 2-local conjunctive predicate under *possibly* modality is NP-complete in general as well. A 2-local conjunctive predicate is a conjunction of clauses such that each clause depends on variables of at most two processes. In Section 3.3, we introduce a new class of predicates called "singular k-CNF predicates". Informally, a k-CNF predicate is singular if no two clauses contain variables from the same process. We show that detecting even a singular 2-CNF predicate under *possibly* modality is NP-complete in general. Our NP-completeness result subsumes the two aforementioned NP-completeness results [CG95, SS95]. It also bridges the wide gap between the known tractability [GW94] and intractability [CG95, SS95] results that existed until now. Further, the NP-completeness result can be used to establish the intractability of detecting other "interesting" singular predicates under *possibly* modality.

It is, however, possible to devise an efficient polynomial-time algorithm for detecting a singular k-CNF predicate under *possibly* modality provided that the computation satisfies certain property, namely it is either receive-ordered or send-ordered [TG98a]. The algorithm is based on Tarafdar and Garg's algorithm for detecting a conjunctive predicate under *possibly* modality for the strong causality model which is an extension of the Lamport's happened-before model [Lam78] in the sense that it allows events on a process to be only partially ordered [TG98a]. The time-complexity of the algorithm is $O(|E|^2)$, where E is the set of events. We also discuss techniques that can be used to achieve an exponential reduction in time over existing techniques for the solving the general version. However, note that the time-complexity of the algorithm for the general version will be exponential in the worst case.

In Section 3.4, we extend the definition of "relational predicate" introduced in [TG97] to include the equality operator. A relational predicate is of the form $x_1 + x_2 + \cdots + x_n$ relop k, where each x_i is an integer variable on process p_i, k is some constant and $\mathsf{relop} \in \{=, <, \leqslant, >, \geqslant\}$. Chase and Garg [CG95] gave polynomial-time algorithm to detect a relational predicate under *possibly* modality when relop $\in \{<, \leqslant, >, \geqslant\}$ based on the notion of max-flow/min-cut. We prove that it is in general NP-complete to detect a relational predicate under *possibly* modality when $\mathsf{relop} = '=' $. However, an efficient polynomial-time algorithm can be developed for the case when each x_i is incremented or decremented by at most one at each step. The time-complexity of the algorithm is $O(|E|^2 \log(|E|))$, where E is the set of events. As a corollary, the above algorithm can be used to detect any "symmetric predicate" on boolean variables under *possibly* modality. A symmetric predicate is invariant under any permutation of its variables. Examples of symmetric predicates include "absence of two-third majority", "exclusive-or of local predicates" and "not all local predicates have the same value".

Although the computation that we construct to prove the NP-completeness result for singular 2-CNF predicates may contain events that send and/or receive multiple messages, it is relatively easy to modify the computation such that each event sends or receives at most one message while ensuring that the NP-completeness result still holds. The basic idea is to replace each event by a contiguous sequence of events such that each event in the sequence sends or receives at most one message (but not both) and the resultant computation satisfies the desired property.

Tarafdar and Garg [TG98b] proved that it is in general NP-complete to monitor a predicate under *controllable* modality. Since the problem of monitoring a predicate under *definitely* modality is dual of the problem of monitoring a predicate under *controllable* modality, it is in general coNP-complete to detect a predicate under *definitely* modality. For their NP-completeness proof, Tarafdar and Garg transformed an arbitrary instance of the problem of detecting a predicate b under *possibly* modality to an instance of monitoring the predicate $x \vee b$ under *controllable* modality [TG98b]. Using their construction and our NP-completeness result for singular 2-CNF predicates, it can be established that controlling a singular 3-CNF predicate in a computation is also intractable in general. This in turn implies that detecting a singular 3-DNF predicate (dual of singular 3-CNF predicate) under *definitely* modality is coNP-complete in general.

3.2 Problem Statement

The predicate detection problem typically refers to monitoring a predicate under *possibly* or *definitely* modality [CM91, WG91]. In this chapter, we mainly focus on detecting a predicate under *possibly* modality and make *possibly* modality explicit only when we need to distinguish it from *definitely* modality.

3.3 Singular k-CNF Predicates

A predicate of boolean variables in conjunctive normal form (CNF) is called *singular* if no two clauses contain variables from the same process. Roughly speaking, a predicate in CNF is singular if it is possible to rewrite the predicate such that each variable occurs in at most one clause and each process hosts at most one variable. For convenience, we write a singular predicate in k-CNF (exactly k literals per clause) as *singular k-CNF predicate*. A singular 1-CNF predicate is also called *conjunctive predicate* [GW94]. For example, let x_i be a boolean variable on process p_i. Then the predicate $(x_1 \vee x_2) \wedge (x_3 \vee x_4 \vee x_5)$ is a singular CNF predicate whereas the predicate $(x_1 \vee x_2) \wedge (x_2 \vee x_3)$ is not.

We first prove that the problem of detecting a singular k-CNF predicate is intractable in general even when k is two. Efficient algorithms for detecting the predicate, however, exist when k is one [CG98]. Our NP-completeness result subsumes the two earlier known NP-completeness results [CG98, SS95]. We next present a polynomial-time algorithm for solving the problem for two special cases, namely when the computation is either receive-ordered or send-ordered [TG98a]; the two notions are defined later in Section 3.3.2. We also discuss techniques that can be used to achieve an exponential reduction in time over existing techniques for solving the general version. The following observation comes in useful for achieving the aforementioned results.

Observation 3.1 *Consider a singular k-CNF predicate b with m clauses $c_i = x_i^1 \vee x_i^2 \vee \cdots \vee x_i^k$, $1 \leqslant i \leqslant m$, where x_i^j is a boolean variable on process p_i^j. Let grp_i denote the subset of processes that host the variables in c_i, that is, $grp_i = \{p_i^j \mid 1 \leqslant j \leqslant k\}$. A* necessary and sufficient *condition for the existence of a consistent cut that satisfies b is the existence of m* pairwise consistent true events *e_i, $1 \leqslant i \leqslant m$, such that each e_i is an event on some process in grp_i.*

The above observation follows from the fact that, given a set of pairwise consistent events—not necessarily from all processes, it is always possible to find a consistent cut that passes through all the events in the set. More precisely, given an event e, let $C_{least}.e$ denote the *least* consistent cut of the computation that passes through e. Now, given a subset of events F, consider the consistent cut $C(F)$ defined as follows:

$$C(F) \triangleq \bigcup_{e \in F} (C_{least}.e)$$

It can be verified that $C(F)$ is not only a consistent cut but also passes through every event in F.

3.3.1 NP-Completeness Result

The problem is in NP because the general problem of detecting an arbitrary boolean expression is in NP [CG98]. To establish its NP-hardness, we transform an arbitrary instance of a *variant* of the satisfiability problem [CLR91], which we call *non-monotone 3-SAT problem*, to an instance of detecting a singular 2-CNF predicate.

Definition 3.1 (non-monotone 3-SAT problem) *Given a formula in CNF such that (1) each clause has at most three literals, and (2) each clause with exactly three literals has at least one positive literal and one negative literal, does there exist a satisfying truth assignment for the formula?*

The NP-completeness of the non-monotone 3-SAT problem follows from the intractability of the 3-SAT problem. Specifically, given a formula in 3-CNF, it can be easily transformed into a formula that satisfies the above-mentioned conditions; we call such a formula *non-monotone 3-CNF formula*. Consider a clause in a 3-CNF formula containing only positive literals, say $c_i = y_i^1 \vee y_i^2 \vee y_i^3$. We replace the clause c_i with three clauses $y_i^1 \vee y_i^2 \vee \neg z_i^3$, $y_i^3 \vee z_i^3$ and $\neg y_i^3 \vee \neg z_i^3$. The last two clauses ensure that, in any satisfying truth assignment, y_i^3 and z_i^3 are logical negation of each other. A similar substitution can be made for clauses containing only negative literals. It is easy to verify that the resultant formula is a non-monotone 3-CNF formula. Furthermore, the new formula is satisfiable if and only if the original formula is satisfiable. Thus we have the following theorem.

Theorem 3.1 *The non-monotone 3-SAT problem is NP-complete in general.*

We now prove the NP-hardness of detecting a singular 2-CNF predicate. Observe that finding a satisfying truth assignment for a non-monotone 3-CNF formula is equivalent to finding a subset of literals, one from each clause, that are mutually non-conflicting. Consequently, it follows from Observation 3.1 that if the computation and the singular 2-CNF predicate satisfy the properties: (1) for each clause in the formula there is a clause in the predicate and vice versa, (2) there is a one-to-one correspondence between the literals in the formula and the true events in the computation, and (3) two literals conflict if and only if the corresponding true events are inconsistent, then the formula is satisfiable if and only if the predicate possibly holds in the computation.

Given a non-monotone 3-CNF formula with clauses c_i, $1 \leqslant i \leqslant m$, we construct a computation and a singular 2-CNF predicate as follows. Without loss of generality, assume that each clause has at least two literals—a lone literal in a clause has to be assigned value true in any satisfying assignment—and no clause contains conflicting literals. For each clause c_i in the formula, we add two processes p_i^1 and p_i^2

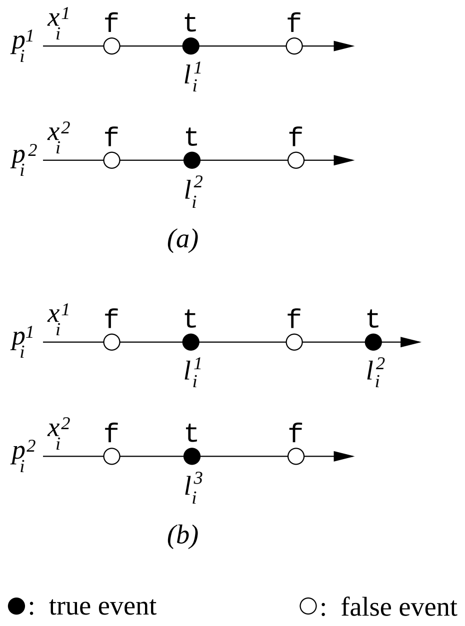


Figure 3.1: The local computation when the clause has (a) two literals and (b) three literals.

to the computation hosting boolean variables x_i^1 and x_i^2, respectively. Initially, all variables evaluate to false. We also add the clause $x_i^1 \vee x_i^2$ to the (singular 2-CNF) predicate. We next describe the local computations of the two processes. There is one true event for each literal in the formula. Depending on the number of literals in the clause, there are two possible cases to consider:

Case 1 ($|c_i| = 2$): Let $c_i = l_i^1 \vee l_i^2$. The local computations of processes p_i^1 and p_i^2 consist of a true event, corresponding to literals l_i^1 and l_i^2, respectively, followed by a false event. For an illustration refer to Figure 3.1(a).

Case 2 ($|c_i| = 3$): Let $c_i = l_i^1 \vee l_i^2 \vee l_i^3$. Without loss of generality, assume that l_i^1 is a positive literal and l_i^2 is a negative literal. The local computation of the process p_i^1 consists of a true event, corresponding to the literal l_i^1, followed by a false event, finally followed by a true event, corresponding to the literal l_i^2. The local

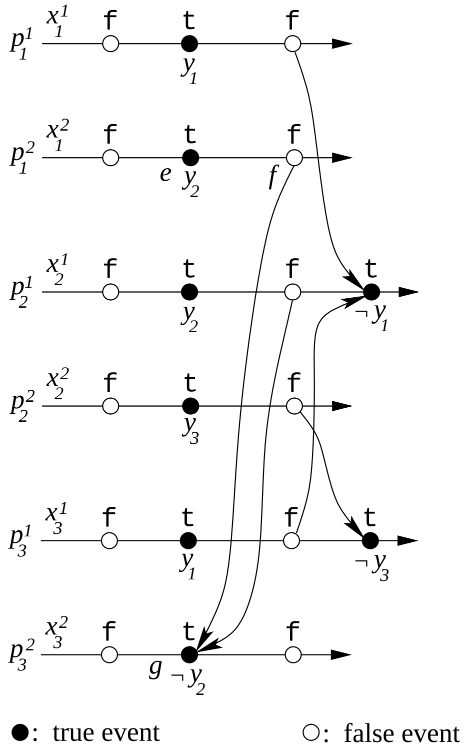


Figure 3.2: An illustration of the transformation (for the non-monotone 3-CNF formula $(y_1 \vee y_2) \wedge (y_2 \vee \neg y_1 \vee y_3) \wedge (y_1 \vee \neg y_3 \vee \neg y_2)$).

computation of the process p_i^2 consists of a true event, corresponding to the literal l_i^3, followed by a false event. For an example see Figure 3.1(b).

Now, given a satisfying truth assignment, the required subset of mutually consistent true events (see Observation 3.1) can be constructed by selecting, for each clause in the predicate, the true event corresponding to the literal with value true (each clause must contain at least one such literal because the truth assignment satisfies the formula). Conversely, given a consistent cut that satisfies the predicate, for each clause in the formula, we can assign the value true to that literal for which the corresponding true event in contained in the cut's frontier. However, in the computation constructed so far, it is possible for two true events to be consistent even if the corresponding literals are conflicting. Thus we may end up assigning true values to conflicting literals. To prevent this from happening, we make the true events corresponding to the conflicting literals inconsistent by adding an arrow (that is, a message) from the successor of the true event corresponding to the positive literal to the true event corresponding to the negative literal as shown in Figure 3.2. For example, e is a true event corresponding to the positive literal y_2 and g is the true event corresponding to the negative literal $\neg y_2$ which conflicts with y_2. Therefore we add an arrow from the successor of e, namely f, to g.

It remains to be shown that the arrows do not create any cycle and two true events are consistent if and only if the corresponding literals are non-conflicting. It suffices to show that there is no causal chain in the computation involving more than one message (or arrow) or, in other words, no dependency is created between true events due to *transitivity*. Observe that the true event corresponding to a negative literal is always at the receiving end of an arrow, if at all, and the successor of the true event corresponding to the positive literal, which is a false event, is always at the sending end of an arrow, if at all. Since there are no other arrows in the computation, each external event in the computation is either a send event or a receive event but not both. Furthermore, if a process contains more than one true event, the true event for the negative literal occurs after the true event for the positive literal. This ensures that if a process has both send and receive events then the receive event occurs after the send event. Thus any causal chain, on reaching a process via a message, cannot subsequently follow any more messages, thereby limiting the size of the causal chain to at most one message.

It is easy to see that the reduction takes polynomial-time and the non-monotone 3-CNF formula is satisfiable if and only if some consistent cut of the computation satisfies the singular 2-CNF predicate.

Theorem 3.2 *Detecting a singular 2-CNF predicate is NP-complete in general.*

Using the above theorem, it can be proved that even detecting predicates such as $(x_1 < x_2) \wedge (x_3 < x_4) \wedge \cdots \wedge (x_{n-1} < x_n)$, where each x_i is an integer variable on process p_i, is NP-complete in general. More precisely,

Corollary 3.3 *Detecting a conjunction of clauses of the form x_i relop x_j, where each x_i is an integer variable and relop $\in \{<, \leqslant, >, \geqslant, \neq\}$, such that no two clauses contain variables from the same process is NP-complete in general.*

Proof: The proof involves a simple reduction from a singular 2-CNF predicate. Consider a clause $y_i \vee y_j$ in a singular 2-CNF predicate. We define integer variables x_i and x_j such that x_i is 0 whenever y_i is false and is -1 otherwise. Similarly, x_j is 0 whenever y_j is false and is 1 otherwise. It can be easily verified that $y_i \vee y_j$ holds if and only if x_i is less than x_j. Similar reductions can be given for other relational operators. $\square$

Although the computation that we construct assumes that an event can send or receive multiple messages, it can be easily modified to ensure that an event sends or receives at most one message while maintaining the property that the formula is satisfiable if and only if the predicate holds in the computation.

3.3.2 Efficient Algorithm for Special Cases

Tarafdar and Garg [TG98a] consider extension of the Lamport's happened-before model [Lam78] for predicate detection that allows events on a process to be partially ordered. They call it the *strong causality model.* For this model, they present an algorithm for detecting a conjunctive predicate when either all receive events on every process are totally ordered or all send events on every process are totally ordered. We denote this algorithm by CPDSC—Conjunctive Predicate Detection in Strong Causality Model. Observation 3.1 enables us to view each group grp_i as a *meta-process* with events on it as partially ordered. Thus CPDSC algorithm can be applied to solve our problem in a straightforward fashion. However, as in their case, either all receive events on every meta-process are totally ordered, that is, the computation is *receive-ordered*, or all send events on every meta-process are totally ordered, that is, the computation is *send-ordered*. We only give an overview of the algorithm here assuming that the computation is receive-ordered. The proof of correctness and other details can be found elsewhere [TG98a].

For the happened-before model, Garg and Waldecker [GW94] give a polynomial-time algorithm for detecting a conjunctive predicate. We denote their algorithm by CPDHB—Conjunctive Predicate Detection in Happened-Before Model. Note that, given a set of true events, one from each process, either events in the set are pairwise consistent or there exist events e and f in the set such that $succ(e)$ happened-before f. Since events on a process are totally ordered in the happened-before model, e is also inconsistent with every event on the process that occurs after f. This allows us to eliminate e from consideration in a scan of the computation from left to right, thereby giving an efficient algorithm for detecting a conjunctive predicate.

Since events on a meta-process are, in general, not totally ordered, CPDHB algorithm cannot be applied directly. However, if the computation is receive-ordered then it satisfies Property 3.1 that enables a polynomial-time algorithm to be devised. Consider a computation $\langle E, \rightarrow \rangle$. We first extend the partial order $\rightarrow$ as follows: for two independent events e and f on a meta-process such that f is a receive event, add an arrow from e to f. It can be proved that the added arrows do not create any cycle [TG98a]. We then linearize the new partial order thus generated to obtain a total order on all events, say $\rightsquigarrow$. It can be verified that the computation satisfies the following property:

Property 3.1 *Given events e, f and g such that events f and g are on the same meta-process but events e and f are on different meta-processes, we have,*

$$(e \rightarrow f) \wedge (f \rightsquigarrow g) \Rightarrow e \rightarrow g$$

Thus, given events e and f on different meta-processes such that $succ(e) \rightarrow f$, by virtue of Property 3.1, e is also inconsistent (with respect to $\rightarrow$) with every event g that occurs after f (with respect to $\rightsquigarrow$) on the same meta-process (as f). Since events on a meta-process are totally ordered with respect to $\rightsquigarrow$, we can eliminate e from consideration in a scan of $\langle E, \rightsquigarrow \rangle$ from left to right. This gives us an efficient algorithm to detect a singular k-CNF predicate when the computation is receive-ordered. The time complexity of the above algorithm is $O(|E|^2)$.

3.3.3 Algorithms for the General Case

For the general case, when the computation is neither receive-ordered nor send-ordered, we can form subsets of processes with each subset containing exactly one process from each meta-process. The CPDHB algorithm can then be applied to each subset [SS95]. Alternatively, we can divide events on each meta-process into a set of chains of events that cover all true events in that meta-process—each true event belongs to at least one chain. We then construct subsets of chains with each subset containing exactly one chain from each meta-process. The CPDHB algorithm can then be applied to each subset. The minimum number of chains needed to cover all true events in a meta-process is upper-bounded by k.

3.4 Relational Predicates: $\mathbf{x_1 + x_2 + \cdots + x_n = k}$

A *relational predicate* [TG97] is of the form $x_1 + x_2 + \cdots + x_n$ relop k, where each x_i is an integer variable on process p_i and $\mathsf{relop} \in \{=, <, >, \leqslant, \geqslant\}$. Note that our definition of relational predicates includes equality which was excluded in the definition by Tomlinson and Garg [TG97]. For convenience, we abbreviate the predicate *possibly*: $(x_1 + x_2 + \cdots + x_n \ \mathsf{relop}\ k)$ by *possibly*: $(\mathsf{relop}\ k)$. For example, *possibly*: $(= k)$ is a shorthand for *possibly*: $(x_1 + x_2 + \cdots + x_n = k)$. Likewise, we obtain *definitely*: $(\mathsf{relop}\ k)$.

We first establish the NP-completeness of evaluating *possibly*: $(= k)$ in general. We next present a polynomial-time algorithm for the special case when each x_i is incremented or decremented by at most one at each step.

3.4.1 NP-Completeness Result

The problem is in NP because the general problem of detecting an arbitrary boolean expression is in NP [CG98]. To prove its NP-hardness, we reduce an arbitrary instance of the subset sum problem [GJ91, problem SP13] to an instance of detecting *possibly*: $(= k)$. The subset sum problem is defined as follows:

Definition 3.2 (subset sum problem [GJ91]) *Given a finite set A, size $s(a_i) \in Z^+$ for each $a_i \in A$ and a positive integer B, does there exist a subset $A' \subseteq A$ such that the sum of the sizes of the elements in A' is exactly B?*

The reduction is as follows. There is a process p_i that hosts variable x_i for each element a_i in the set A. The initial value of each x_i is set to zero. Each process has exactly one event e_i; the final value of x_i, after executing e_i, is $s(a_i)$. Finally, k is set to B. It is easy to see that the reduction takes polynomial-time and the required subset exists if and only if *possibly*: $(= k)$ holds.

Theorem 3.4 *Detecting possibly*: $(= k)$ *when each x_i can be modified (incremented or decremented) by an arbitrary amount at each step is NP-complete in general.*

3.4.2 Efficient Algorithm for the Special Case

It is possible to devise an efficient algorithm for detecting *possibly*: $(= k)$ in a computation provided that each x_i is incremented or decremented by at most one at each step. The algorithm is based on monitoring predicates *possibly*: $(\leqslant k)$ and *possibly*: $(\geqslant k)$. Efficient algorithms to observe these predicates can be found elsewhere [CG95, TG97].

A consistent cut C' is *reachable* from a consistent cut C if it is possible to attain C' from C by executing zero or more events. It can be verified that C' is reachable from C if and only if $C \subseteq C'$. If C' can be obtained from C by executing exactly one event then C' *immediately succeeds* C. Furthermore, C *immediately precedes* C'.

A sequence of consistent cuts $\{C_i\}_{i>0}$ forms a *path* in a computation if each C_{i+1} immediately succeeds C_i. Observe that if C' is reachable from C then there is a path from C to C' and vice versa. Moreover, every run corresponds to a path in the computation.

Observation 3.2 *Let C and C' be consistent cuts such that C' is obtained from C by executing at most one event. Then $|sum(C') - sum(C)| \leqslant 1$.*

Given a consistent cut C, let $sum(C)$ denote the value of the sum $x_1 + x_2 + \cdots + x_n$ evaluated at C. Given a pair of integers u and v, let $range(u, v)$ denote the set $[\min\{u, v\} \ldots \max\{u, v\}]$. For example, $range(3, 8) = [3 \ldots 8] = \{3, 4, 5, 6, 7, 8\}$ and $range(6, 2) = [2 \ldots 6] = \{2, 3, 4, 5, 6\}$.

Theorem 3.5 *Let C and C' be consistent cuts such that there is a path s from C to C' in the computation. Then, for each v,*

$$v \in range(sum(C), sum(C')) \Rightarrow \langle \exists\, D : D \in s : sum(D) = v \rangle$$

Proof: Without loss of generality, assume that $sum(C) \leqslant sum(C')$. The proof for the other case, when $sum(C) \geqslant sum(C')$, is similar and has been omitted. Assume that $v \in range(sum(C), sum(C'))$, that is, $sum(C) \leqslant v \leqslant sum(C')$. If $v = sum(C')$ then C' is the required consistent cut. Thus assume that $v < sum(C')$. Starting from C we follow the path s by executing, one-by-one, zero or more events in $C' \setminus C$ until we reach a consistent cut H such that $sum(H) \geqslant v$ for the first time. We claim that $sum(H) = v$. Assume, by the way of contradiction, that $sum(H) \neq v$, that is, $sum(H) > v$. Note that H exists since $sum(C') > v$. Let G be the consistent cut that immediately precedes H along the path. Note that G exists since $sum(C) \leqslant v$. Moreover, $sum(G) < v$ because H is the first consistent cut with sum at least v. Thus (1) $sum(H) > v$ implying that $sum(H) \geqslant v + 1$, and (2) $sum(G) < v$ implying that $sum(G) \leqslant v - 1$. Combining the two, we have $sum(H) - sum(G) \geqslant 2$, a contradiction. Therefore $sum(H) = v$ and H is the required consistent cut. $\square$

The central idea behind the algorithm for detecting $possibly\colon(= k)$ is to find a pair of consistent cuts C and C', if they exist, such that C' is reachable from C and k lies in $range(sum(C), sum(C'))$. Theorem 3.5 then guarantees the existence of a consistent cut that satisfies $x_1 + x_2 + \cdots + x_n = k$. The consistent cut C is always set to the initial consistent cut $\bot$. The advantage is that every consistent cut of the computation is reachable from the initial consistent cut. The next lemma furnishes sufficient conditions for $possibly\colon(= k)$ to hold in a computation.

Lemma 3.6 *We have,*

$$(sum(\bot) \leqslant k) \wedge (possibly\colon(\geqslant k)) \;\Rightarrow\; possibly\colon(= k), \text{ and}$$
$$(sum(\bot) \geqslant k) \wedge (possibly\colon(\leqslant k)) \;\Rightarrow\; possibly\colon(= k)$$

Proof: Assume that the conjunction $(sum(\bot) \leqslant k) \wedge (possibly\colon(\geqslant k))$ holds. Since $possibly\colon(\geqslant k)$ is true, there exists a consistent cut with C' with $sum(C') \geqslant k$. Thus, from Theorem 3.5, there exists a consistent cut D such that $sum(D) = k$ implying that $possibly\colon (= k)$ holds. Likewise, $(sum(\bot) \geqslant k) \wedge (possibly\colon (\leqslant k))$ implies $possibly\colon (= k)$. $\square$

The following lemma presents sufficient conditions for $definitely\colon (= k)$ to hold in a computation. The proof is similar to the proof of Lemma 3.6 and has been omitted.

Lemma 3.7 *We have,*

$$(sum(\bot) \leqslant k) \wedge (definitely\colon (\geqslant k)) \quad \Rightarrow \quad definitely\colon (= k), \text{ and}$$
$$(sum(\bot) \geqslant k) \wedge (definitely\colon (\leqslant k)) \quad \Rightarrow \quad definitely\colon (= k)$$

Finally, the following theorem gives the necessary and sufficient conditions for predicates $possibly\colon (= k)$ and $definitely\colon (= k)$ to hold in a computation.

Theorem 3.8 *We have,*

$$\text{(1) } possibly\colon (= k) \equiv (sum(\bot) \leqslant k) \wedge (possibly\colon (\geqslant k)) \bigvee (sum(\bot) \geqslant k) \wedge (possibly\colon (\leqslant k))$$

$$\text{(2) } definitely\colon (= k) \equiv (sum(\bot) \leqslant k) \wedge (definitely\colon (\geqslant k)) \bigvee (sum(\bot) \geqslant k) \wedge (definitely\colon (\leqslant k))$$

Proof: *(1)* Follows from the fact that $possibly\colon (= k)$ implies $possibly\colon (\leqslant k) \wedge possibly\colon (\geqslant k)$, the disjunction $(sum(\bot) \leqslant k) \vee (sum(\bot) \geqslant k)$ is a tautology and Lemma 3.6.

(2) Follows from the fact that $definitely\colon (= k)$ implies $definitely\colon (\leqslant k) \wedge definitely\colon (\geqslant k)$, the disjunction $(sum(\bot) \leqslant k) \vee (sum(\bot) \geqslant k)$ is a tautology and Lemma 3.7. $\square$

Observe that the final consistent cut is reachable from every consistent cut of a computation. Thus an alternate set of necessary and sufficient conditions for *possibly*:$(= k)$ and *definitely*:$(= k)$ based on final consistent cut can also be derived. The time-complexity of computing *possibly*: $(\leqslant k)$ or *possibly*: $(\geqslant k)$ [TG97, CG95] is $O(|E|^2 \log(|E|))$. Thus the time-complexity of computing *possibly*: $(= k)$ is also $O(|E|^2 \log(|E|))$.

Since *possibly* distributes over disjunction, the following predicates, expressed as disjunction of predicates of the form $x_1 + x_2 + \cdots + x_n$ exactly equals k, can be easily detected using Theorem 3.8.

- absence of simple majority: $v_1 + v_2 + \cdots + v_n = n/2$, n even
- absence of two-third majority:
  $(v_1 + v_2 + \cdots v_n > \lfloor \frac{n}{3} \rfloor) \wedge (v_1 + v_2 + \cdots v_n < \lceil \frac{2n}{3} \rceil) \equiv \bigvee_{k \in A} (v_1 + v_2 + \cdots + v_n = k)$,
  where $A = [\lfloor \frac{n}{3} \rfloor + 1 \ldots \lceil \frac{2n}{3} \rceil - 1]$
- exactly k tokens: $token_1 + token_2 + \cdots + token_n = k$

Additionally, the symmetric predicates, defined as follows, can now be efficiently monitored.

Definition 3.3 (symmetric predicate [Koh78]) *A predicate $b(x_1, x_2, \ldots, x_n)$ defined on n boolean variables is called* symmetric *if it is invariant under any permutation of its variables.*

Some examples of symmetric predicates are $x \wedge y$, $x \vee y$, $x \oplus y$ and $(x \wedge y) \vee (\neg x \wedge \neg y)$. The necessary and sufficient condition for a predicate $b(x_1, x_2, \ldots, x_n)$ to be symmetric is that it may be specified by a set of numbers $\{a_1, a_2, \ldots, a_m\}$, where $0 \leqslant a_i \leqslant n$ and $m \leqslant n + 1$, such that it assumes value true when and only when, for some i, exactly a_i of the variables are true. For example, the symmetric predicate $(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$ is logically equivalent to the predicate $(x + y + z = 1) \vee (x + y + z = 2)$, where false and true are represented by 0 and 1, respectively, for the purpose of evaluating $x + y + z$. The proof of this result can be found elsewhere [Koh78, page 174]. Since, *possibly* distributes over disjunction, $possibly\!:\!b$ when b is a symmetric predicate can be efficiently computed using Theorem 3.8. Some examples of symmetric predicates that arise in distributed systems are:

- exclusive-or of local predicates:

  $x_1 \oplus x_2 \oplus \cdots \oplus x_n \equiv \bigvee_{k \text{ is odd}} (x_1 + x_2 + \cdots + x_n = k)$

- not all local predicates have the same value:

  $(x_1 \vee x_2 \vee \cdots \vee x_n) \wedge (\neg x_1 \vee \neg x_2 \vee \cdots \vee \neg x_n) \equiv \bigvee_{k \in A} (x_1 + x_2 + \cdots + x_n = k)$, where $A = [1 \ldots (n-1)]$

Chapter 4

Controlling Global Predicates

In this chapter, we discuss in detail our results pertaining to controlling global predicates in distributed computations.

4.1 Overview

We first define the problem formally in Section 4.2. Informally, a predicate is said to be controllable in a computation if it is possible to add synchronization dependencies, without creating a deadlock (that is, a cycle), such that every consistent cut of the resulting computation satisfies the predicate. In case the predicate can indeed be controlled in the computation, the set of synchronization dependencies required to control the predicate is referred to as "controlling synchronization". The resultant computation is called "controlled computation". A synchronization dependency from an event e to an event f means that f cannot be executed until e has been executed and can be implemented using a control message.

Tarafdar and Garg [TG98b] establish that it is in general NP-complete to control a predicate in a computation. However, efficient polynomial-time algorithms can be developed for many useful classes of predicates [TG98b, TG99].

In Section 4.3, we introduce a new class of predicates called "region predicates". A region predicate is a conjunction of p-region predicates, where p is a process, with possibly different p's. Roughly speaking, a p-region predicate partitions the set of consistent cuts that satisfy the predicate into a set of "convex regions", one for each event on process p, such that the set of consistent cuts that lie in a region forms a lattice under set containment. Intuitively, on reaching an event on process p, once the p-region predicate is falsified (that is, becomes false from true), it does not become true again until the computation advances beyond the event. The class of p-region predicates is closed under conjunction and hence the class of region predicates is closed under conjunction. Some examples of region predicates are termination, conjunctive predicates and monotonic channel predicates such as "at most (or at least) k messages in transit in any channel".

We present an efficient polynomial-time algorithm to control a region predicate in a computation. The time-complexity of our algorithm is $O(n|E|^2)$, where n is the number of processes and E is the set of events. We also prove that the controlling synchronization generated by our algorithm is *optimal* in the sense that it not only eliminates all unsafe runs but also retains all safe runs.

In Section 4.4, we introduce the notion of an "admissible sequence" of events with respect to a predicate. Specifically, we identify four properties that characterize an admissible sequence. Roughly speaking, an admissible sequence imposes a total order on "certain" events in the computation such that executing those events in that order ensures that the predicate is never falsified. We show that the existence of an admissible sequence of events with respect to a predicate is a necessary and sufficient condition for a predicate to be controllable in a computation. Further, given an admissible sequence, the controlling synchronization can be easily obtained and vice versa.

Based on the notion of admissible sequence, we devise a polynomial-time algorithm for controlling a "disjunctive predicate" in a computation. A disjunctive predicate is a disjunction of local predicates. Intuitively, a disjunctive predicate states that at least one local condition must be met at all times, or, in other words, a bad combination of local conditions does not occur. Examples of disjunctive predicates include "at least one server is available" and "at least one philosopher does not have any fork".

To control a disjunctive predicate in a computation, we construct a directed graph on "true-intervals" (maximal contiguous sequence of true events on a process) of the computation such that the problem of determining an admissible sequence reduces to finding an appropriate shortest path in the graph. The time-complexity of the algorithm is $O(n|T|)$, where n is the number of processes and T is the set of true-intervals, which is same as that of Tarafdar and Garg's algorithm [TG98b]. We further modify the algorithm to compute a *minimum* controlling synchronization—with the least number of synchronization dependencies—for a disjunctive predicate. Clearly, a minimum controlling synchronization minimizes the number of control messages required to maintain a disjunctive predicate in a computation. The time-complexity of the modified algorithm is $O(|E|^2)$, where E is the set of events.

4.2 Problem Statement

The predicate control problem refers to monitoring a predicate under *controllable* modality [TG98b]. Intuitively, a predicate is *controllable* in a computation if it is possible to make the computation "stricter" such that every consistent cut of the resulting computation satisfies the predicate. More precisely, a predicate b is controllable in a computation $\langle E, \rightarrow \rangle$ if there exists a set of synchronization dependencies $\xrightarrow{S}$ such that (1) $\xrightarrow{S}$ does not interfere with $\rightarrow$, that is, $(\rightarrow \cup \xrightarrow{S})$ is acyclic, and (2) every consistent cut of $\langle E, \leadsto \rangle$, where $\leadsto = (\rightarrow \cup \xrightarrow{S})^+$, satisfies b.

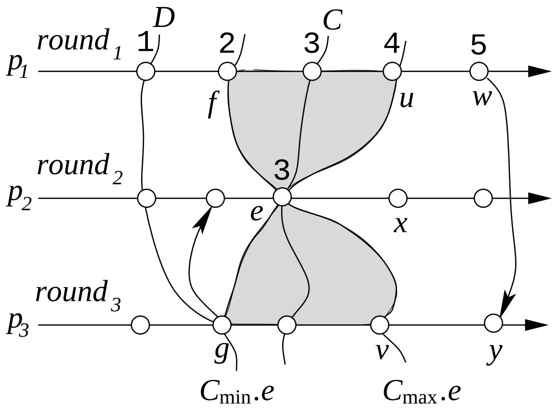


Figure 4.1: An example of a p-region predicate.

We call the synchronization $\xrightarrow{S}$ as a *controlling synchronization* and the computation $\langle E, \rightsquigarrow \rangle$ as the *controlled computation*. This definition of *controllable*: b is slightly different from the definition provided in Chapter 2. It can be verified that both definitions are actually equivalent.

Note that a synchronization dependency from an event e to an event f means that f cannot be executed until e has been executed and can be implemented using a control message.

4.3 Region Predicates

We first define a region predicate with respect to a process, called *p-region predicate*. Informally, a p-region predicate partitions the set of consistent cuts satisfying the predicate into a set of regions, one for each event on process p, satisfying certain properties. Firstly, the set of consistent cuts that lie in a region (that is, all events in the frontier of the cut belong to the region) forms a lattice. Secondly, each region is convex or, equivalently, a consistent cut that lies between two consistent cuts contained in the region also belongs to the region.

Example 4.1 *Consider the computation shown in Figure 4.1 and the predicate "processes p_1 and p_2 are approximately synchronized" expressed mathematically as $|round_1 - round_2| \leqslant \Delta_{12}$ with Δ_{12} set to 1. Consider the event e on p_2 depicted in the figure. Immediately after executing e, the value of $round_2$ is 3. Since $round_1$ is monotonically non-decreasing, there exist earliest and latest events on p_1, in this case f and u, respectively, such that the predicate holds. Furthermore, the predicate holds for every event on p_1 that lies between f and u. The region corresponding to e (the shaded area resembling the cross-section of an hourglass in the figure) is bounded on the left by the least consistent cut passing through e and f and on the right by the greatest consistent cut passing through e and u. The consistent cut C lies in the region whereas the consistent cut D does not. It can be verified that the region is actually convex and the set of consistent cuts that belong to the region forms a lattice.*

A p-region predicate is formally defined as follows:

Definition 4.1 (p-region predicate) *A predicate b is a p-*region predicate *if it satisfies the following properties. For each event e on process p,*

- **(weak lattice)** *If two consistent cuts that pass through e satisfy the predicate then so do the consistent cuts given by their set intersection and set union. Formally,*

$$(e \in \mathit{frontier}(C_1) \cap \mathit{frontier}(C_2)) \wedge (C_1 \models b) \wedge (C_2 \models b)$$
$$\Rightarrow$$
$$(C_1 \cap C_2 \models b) \wedge (C_1 \cup C_2 \models b)$$

- **(weak convexity)** *If two consistent cuts that pass through e satisfy the predicate then so does the consistent cut that lies between the two. Formally,*

$$(e \in \mathit{frontier}(C_1) \cap \mathit{frontier}(C_2)) \wedge (C_1 \models b) \wedge (C_2 \models b) \wedge$$
$$(C_1 \subseteq C \subseteq C_2)$$
$$\Rightarrow$$
$$C \models b$$

We call the two properties "weak" because they are only satisfied by those consistent cuts that satisfy the predicate *and pass through a given event*, and not by *all* consistent cuts that satisfy the predicate. Some examples of p_i-region predicates encountered in distributed systems are as follows:

- any local predicate on p_i
- "bounded" number of messages in transit from p_i to p_j: $send_{ij} - recv_{ij} \leqslant \Delta_{ij}$
- "almost" fair resource allocation between p_i and p_j, when the system is heavily loaded: $|alloc_i - alloc_j| \leqslant \Delta_{ij}$
- "bounded" drift between the clocks of p_i and p_j: $|clock_i - clock_j| \leqslant \Delta_{ij}$
- p_i and p_j are "approximately" synchronized: $|round_i - round_j| \leqslant \Delta_{ij}$
- $x_i < \min\{y_j, y_k\}$, where x_i, y_j and z_k are variables on p_i, p_j and p_k, respectively, with y_j and y_k monotonically non-decreasing

Given two p-region predicates, their conjunction is also a p-region predicate as established by the next theorem.

Theorem 4.1 *The class of p-region predicates is closed under conjunction.*

Proof: We have to prove that if b_1 and b_2 are p-region predicates then so is $b_1 \wedge b_2$. We first prove that $b_1 \wedge b_2$ satisfies the weak lattice property. Consider consistent cuts C_1 and C_2 passing through an event e on process p that satisfy $b_1 \wedge b_2$. By semantics of conjunction, both C_1 and C_2 satisfy b_1 as well as b_2. Applying the weak lattice property twice, we obtain $C_1 \cap C_2$ satisfies b_1 and b_2. Again, by semantics of conjunction, $C_1 \cap C_2$ satisfies $b_1 \wedge b_2$. Likewise, $C_1 \cup C_2$ satisfies $b_1 \wedge b_2$. Thus $b_1 \wedge b_2$ satisfies the weak lattice property.

We now prove that $b_1 \wedge b_2$ satisfies the weak convexity property. Consider consistent cuts C_1 and C_2 passing through e that satisfy $b_1 \wedge b_2$ and let C be any consistent cut that lies between the two. By semantics of conjunction, both C_1 and C_2 satisfy b_1 as well as b_2. Applying the weak convexity property twice, we obtain C satisfies b_1 and b_2. This implies that C satisfies $b_1 \wedge b_2$. Therefore $b_1 \wedge b_2$ satisfies the weak convexity property. $\square$

A *region predicate* is a conjunction of p-region predicates with possibly different p's. It can be verified that the predicate representing termination is actually a region predicate. Note that, for each p, true is a p-region predicate. Thus a region predicate b can be written as conjunction of n predicates such that the i^{th} conjunct, denoted by $b^{(i)}$, is a p_i-region predicate.

Given an event e on process p_i, we denote the *least* consistent cut passing through e that satisfies $b^{(i)}$ by $C_{\min}.e$. Similarly, we denote the *greatest* consistent cut passing through e that satisfies $b^{(i)}$ by $C_{\max}.e$. If there does not exist a consistent cut that passes through e and satisfies $b^{(i)}$ then neither $C_{\min}.e$ nor $C_{\max}.e$ exists. Additionally, trivially, $b^{(i)}$ (and hence b) cannot be controlled in the computation. However, if there exists at least one consistent cut passing through e that satisfies $b^{(i)}$ then both $C_{\min}.e$ and $C_{\max}.e$ exist and are well-defined. This is because, from the weak lattice property, the set of such consistent cuts forms a lattice under set containment ($\subseteq$) implying that the set has a minimum (corresponds to $C_{\min}.e$) and a maximum (corresponds to $C_{\max}.e$).

4.3.1 Finding a Controlling Synchronization

In order to find the synchronization necessary to control a region predicate in a computation, we first compute the synchronizations sufficient to control each of its conjunct (recall that the i^{th} conjunct corresponds to a p_i-region predicate). If it turns out that one or more of these conjuncts is not controllable then, trivially, the region predicate itself cannot be controlled. Further, in case the various synchronizations (corresponding to different conjuncts) do not interfere with each other and, in addition, the *collective* synchronization does not interfere with the happened-before relation of the computation then, clearly, the collective synchronization constitutes a controlling synchronization for the given region predicate. Unfortunately, the converse does not hold in general.

Example 4.2 *Suppose we wish to control the predicate* $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$ *in the computation shown in Figure 4.2(a), where each* x_i *is a boolean variable on process* p_i. *It can be verified that the arrow from event* h *to event* e *constitutes a controlling synchronization for the first conjunct* $x_1 \vee x_2$. *Similarly, the arrow from event* v *to event* u *constitutes a controlling synchronization for the second conjunct* $x_3 \vee x_4$. *However, the collective synchronization given by* $\{(h, e), (v, u)\}$ *interferes with the happened-before relation of the computation. In other words, it creates a cycle as shown in Figure 4.2(b). The first conjunct has another controlling synchronization, namely the arrow from event* f *to event* g. *In this case, the collective synchronization given by* $\{(f, g), (v, u)\}$ *neither interferes with itself nor with the happened-before relation of the computation, thereby constituting a controlling synchronization for the predicate* $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$.

However, if the computed synchronization for each conjunct is *smallest* in the sense that it is contained in every possible controlling synchronization for the respective conjunct then the converse also holds. That is, if the region predicate is controllable in a computation then the various synchronizations not only do not interfere with each other but, additionally, the collective synchronization does not interfere with the happened-before relation of the computation. Intuitively, this is because a controlling synchronization for a region predicate also acts as a controlling

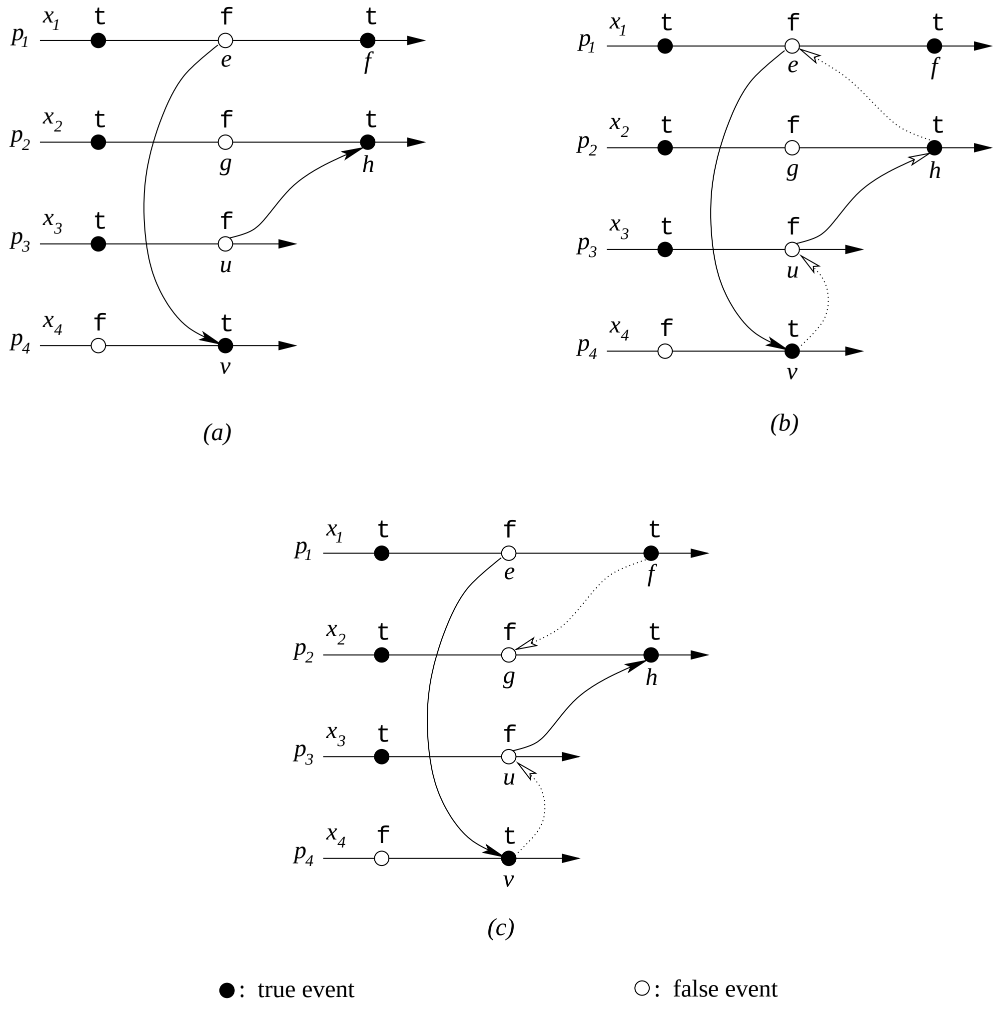


Figure 4.2: An example to illustrate that the interference of some collective synchronization with the happened-before relation does not imply that the predicate cannot be controlled.

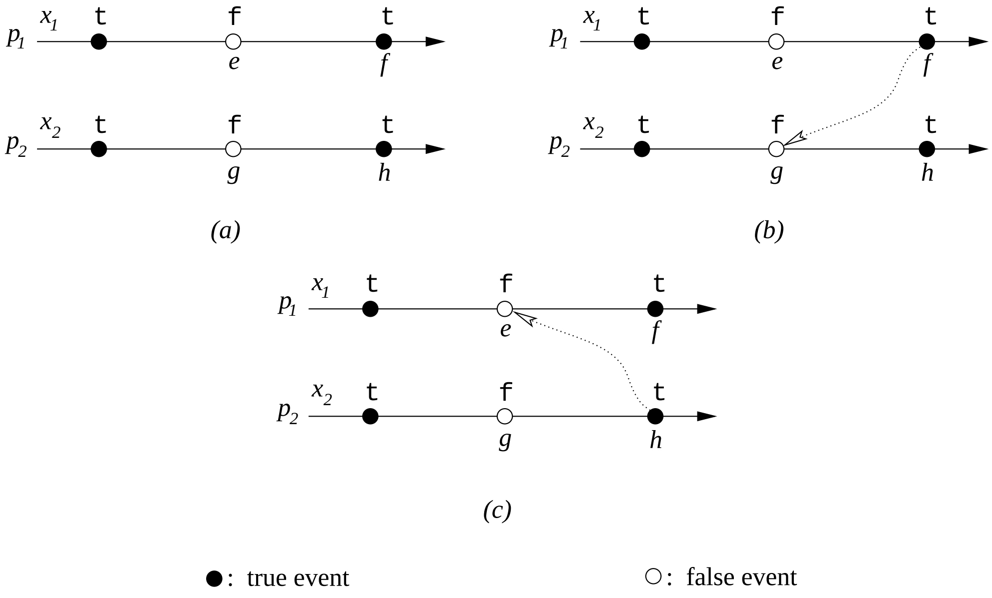


Figure 4.3: An example to illustrate that the smallest controlling synchronization may not always exist.

synchronization for each of its conjunct.

Definition 4.2 (smallest controlling synchronization) *We call a controlling synchronization* smallest *if it is contained in every possible controlling synchronization for the predicate. Formally, given a controlling synchronization* $\xrightarrow{S}$ *for a predicate* b *in a computation* $\langle E, \rightarrow \rangle$,

$$\xrightarrow{S} \textit{ is smallest} \quad \triangleq \quad \langle \forall \leadsto : \leadsto \textit{ extends } \rightarrow : \\ \langle E, \leadsto \rangle \models \textit{invariant} : b \;\equiv\; \leadsto \textit{ contains } \xrightarrow{S} \rangle$$

A smallest controlling synchronization may not always exist as illustrated by the following example.

Example 4.3 *Consider the computation in Figure 4.3(a). Suppose we desire to control the predicate* $x_1 \vee x_2$ *in the computation, where each* x_i *is a boolean variable* *on process p_i. Since the predicate $x_1 \vee x_2$ is not invariant in the computation to begin with, the smallest controlling synchronization, if it exists, must be non-empty. It can be verified that the arrow from event f to event g constitutes a controlling synchronization for the predicate $x_1 \vee x_2$, as shown in Figure 4.3(b), as does the arrow from event h to event e, as depicted in Figure 4.3(c). Moreover, the two synchronizations are mutually disjoint implying that the predicate $x_1 \vee x_2$ does not have a smallest controlling synchronization.*

As it happens, the smallest controlling synchronization in fact exists for a p-region predicate (and therefore also exists for a region predicate). Thus in order to find a controlling synchronization for a region predicate, from the above discussion, it suffices to devise an algorithm to compute the smallest controlling synchronization for a p-region predicate.

Consider a computation $\langle E, \rightarrow \rangle$ and a region predicate b. What does it entail to control the p_i-region predicate $b^{(i)}$, $1 \leqslant i \leqslant n$, in $\langle E, \rightarrow \rangle$? Consider an event e on process p_i. As we know, the computation progresses from the initial consistent cut $\perp$ to the final consistent cut E by executing, one-by-one, the events in E. For $b^{(i)}$ to hold when it first reaches e, it must be the case that no event in the frontier of the computation lies on the left of the frontier of $C_{\min}.e$. That is, when e is executed, all other events in the frontier of $C_{\min}.e$ must have already been executed. This entails adding synchronization dependencies from each event in the frontier of $C_{\min}.e$ that is different from e to e. We denote this synchronization by $\stackrel{e^{(1)}}{\rightarrow}$ and formally define it as follows:

$$\stackrel{e^{(1)}}{\rightarrow} \;\triangleq\; \{\, (f, e) \mid f \in \mathit{frontier}(C_{\min}.e) \setminus \{e\} \;\text{ and }\; e \not\in \perp \,\}$$

For an example refer to Figure 4.4. Furthermore, for $b^{(i)}$ to hold as long as the computation stays at e (equivalently, until the successor of e, if it exists, is executed), the frontier of the computation cannot advance beyond $C_{\max}.e$. That

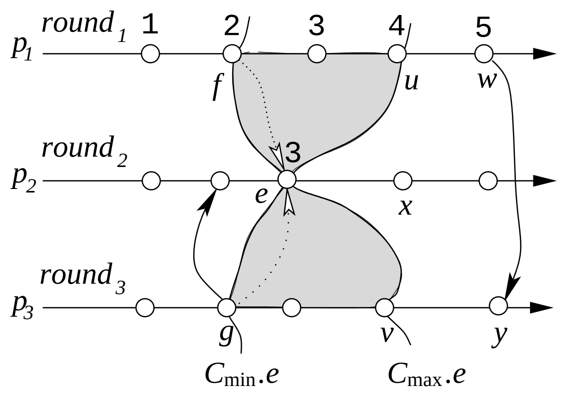


Figure 4.4: An illustration of the synchronization $\stackrel{e^{(1)}}{\rightarrow}$ (denoted by dotted arrows).

is, the successor of any event in the frontier of $C_{\max}.e$ that is different from e, if it exists, cannot be executed until the computation advances beyond e. This involves adding synchronization dependencies from the successor of e, if it exists, to the successor of every other event in the frontier of $C_{\max}.e$, if it exists. We denote this synchronization by $\stackrel{e^{(2)}}{\rightarrow}$ and formally define it as follows:

$$\stackrel{e^{(2)}}{\rightarrow} \;\triangleq\; \{\, (succ(e), succ(f)) \mid f \in \mathit{frontier}(C_{\max}.e) \setminus \{e\} \;\text{ and }\; \{e, f\} \cap \top = \emptyset \,\}$$

For an illustration see Figure 4.5. The overall synchronization needed for controlling $b^{(i)}$ in $\langle E, \rightarrow\rangle$ is given by the union of $(\stackrel{e^{(1)}}{\rightarrow} \cup \stackrel{e^{(2)}}{\rightarrow})$, where e ranges over the events on process p_i. Finally, the synchronization required to control b in $\langle E, \rightarrow\rangle$, denoted by $\stackrel{S}{\rightarrow}$, is given by:

$$\stackrel{S}{\rightarrow} \;\triangleq\; \bigcup_{e \in E} (\stackrel{e^{(1)}}{\rightarrow} \cup \stackrel{e^{(2)}}{\rightarrow}) \tag{4.1}$$

For convenience, we use $\stackrel{C}{\rightarrow}$ to denote the transitive closure of the relation obtained by adding $\stackrel{S}{\rightarrow}$ to $\rightarrow$. Formally,

$$\stackrel{C}{\rightarrow} \;\triangleq\; (\rightarrow \cup \stackrel{S}{\rightarrow})^{+}$$

The next lemma describes the sufficient condition under which a region predicate is controllable in a computation. Informally, this happens when each

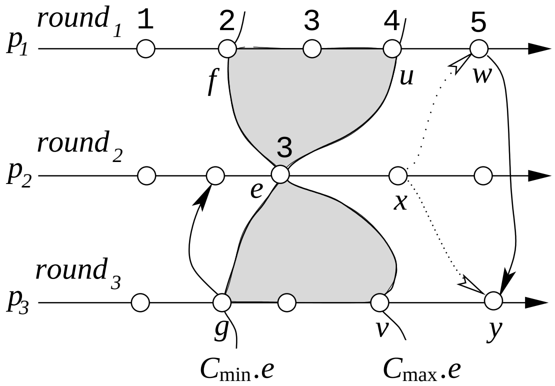


Figure 4.5: An illustration of the synchronization $\stackrel{e^{(2)}}{\rightarrow}$ (denoted by dotted arrows).

of its conjunct is controllable and the collective synchronization neither interferes with itself nor with the happened-before relation of the computation—which can be succinctly represented as: $(\rightarrow \cup \stackrel{S}{\rightarrow})$ is acyclic.

Lemma 4.2 (sufficient condition) *If (1) the initial and final consistent cuts of a computation* $\langle E, \rightarrow \rangle$ *satisfy a region predicate* b*, and (2)* $\stackrel{C}{\rightarrow}$ *is an irreflexive partial order then* b *is invariant in* $\langle E, \stackrel{C}{\rightarrow} \rangle$*.*

Proof: Consider a consistent cut C of $\langle E, \stackrel{C}{\rightarrow} \rangle$ and an event e contained in its frontier. We show that C lies between $C_{\min}.e$ and $C_{\max}.e$. We first prove that $C_{\min}.e \subseteq C$. If $e \in \bot$ then $C_{\min}.e = \bot$ because, trivially, $\bot$ is the least consistent cut of $\langle E, \rightarrow \rangle$ that passes through e and $\bot \models b$. Furthermore, by definition of consistent cut, $C \supseteq \bot$. Thus $C_{\min}.e \subseteq C$. The more interesting case is when $e \notin \bot$. We want to prove that,

$$
\begin{array}{cl}
& C_{\min}.e \subseteq C \\
\equiv & \{ \text{ definition of consistent cut and its frontier } \} \\
& \langle \forall f : f \in \mathit{frontier}(C_{\min}.e) : f \in C \rangle \\
\equiv & \{ \text{ by definition, } C_{\min}.e \text{ passes through } e \text{ } \}
\end{array}
$$

$$
\begin{array}{ll}
 & (e \in C) \ \wedge \ \langle \forall\, f : f \in \mathit{frontier}(C_{\min}.e) \setminus \{e\} : f \in C \rangle \\
\Leftarrow & \{\ C \text{ is a consistent cut of } \langle E, \stackrel{C}{\rightarrow} \rangle\ \} \\
 & (e \in C) \ \wedge \ \langle \forall\, f : f \in \mathit{frontier}(C_{\min}.e) \setminus \{e\} : f \stackrel{C}{\rightarrow} e \rangle \\
\Leftarrow & \{\ C \text{ passes through } e\ \} \\
 & \langle \forall\, f : f \in \mathit{frontier}(C_{\min}.e) \setminus \{e\} : (f \stackrel{C}{\rightarrow} e) \rangle \\
\Leftarrow & \{\ \stackrel{S}{\rightarrow} \subseteq \stackrel{C}{\rightarrow}\ \} \\
 & \langle \forall\, f : f \in \mathit{frontier}(C_{\min}.e) \setminus \{e\} : (f \stackrel{S}{\rightarrow} e) \rangle \\
\Leftarrow & \{\ \stackrel{e^{(1)}}{\rightarrow} \subseteq \stackrel{S}{\rightarrow}\ \} \\
 & \langle \forall\, f : f \in \mathit{frontier}(C_{\min}.e) \setminus \{e\} : (f \stackrel{e^{(1)}}{\rightarrow} e) \rangle \\
 & \{\ e \notin \bot \ \text{ and } \ \text{definition of } \stackrel{e^{(1)}}{\rightarrow}\ \}
\end{array}
$$

Likewise, $C \subseteq C_{\max}.e$. Let $\mathit{proc}(e) = p_i$. By definition, both $C_{\min}.e$ and $C_{\max}.e$ satisfy $b^{(i)}$. Thus, from the weak convexity property, C satisfies $b^{(i)}$. Since e was chosen arbitrarily, for each i, we can infer that C satisfies $b^{(i)}$. This implies that C satisfies b. $\square$

The next lemma proves that the synchronization given by $\stackrel{S}{\rightarrow}$ is indeed the smallest controlling synchronization for b in $\langle E, \rightarrow \rangle$. In other words, any other controlling synchronization for b in $\langle E, \rightarrow \rangle$, if it exists, must contain $\stackrel{S}{\rightarrow}$.

Theorem 4.3 *If a region predicate b is controllable in a computation $\langle E, \rightarrow \rangle$ then the synchronization $\stackrel{S}{\rightarrow}$ defined in (4.1) is the smallest controlling synchronization.*

Proof: Since b is controllable in $\langle E, \rightarrow \rangle$, there exists an irreflexive partial order $\leadsto$ that extends $\rightarrow$ such that b is invariant in $\langle E, \leadsto \rangle$. We need to prove that $\stackrel{S}{\rightarrow}$ is contained in $\leadsto$. It is sufficient to prove that, for each event e, both $\stackrel{e^{(1)}}{\rightarrow}$ and $\stackrel{e^{(2)}}{\rightarrow}$ are contained in $\leadsto$.

We first show that, for each event e, $\leadsto$ includes $\stackrel{e^{(1)}}{\rightarrow}$. Consider an event e, $e \notin \bot$, on process p_i. Note that if $e \in \bot$ then $\stackrel{e^{(1)}}{\rightarrow}$ is an empty set. In the proof we use the notion of the *least* consistent cut of $\langle E, \rightsquigarrow \rangle$ that contains e, denoted by $C_{least}.e$. By definition, $C_{least}.e$ passes through e and an event other than e belongs to $C_{least}.e$ if and only if it happened-before e in $\langle E, \rightsquigarrow \rangle$. Formally,

$$(e \in \mathit{frontier}(C_{least}.e)) \ \wedge \ \langle \forall f : f \neq e : f \in C_{least}.e \ \equiv \ f \rightsquigarrow e \rangle \tag{4.2}$$

We want to prove that,

$$
\begin{array}{ll}
 & \stackrel{e^{(1)}}{\rightarrow} \subseteq \rightsquigarrow \\
\equiv & \{ \text{ definition of } \stackrel{e^{(1)}}{\rightarrow} \} \\
 & \langle \forall f : f \in \mathit{frontier}(C_{\min}.e) \setminus \{e\} : f \rightsquigarrow e \rangle \\
\equiv & \{ \text{ using (4.2) } \} \\
 & \langle \forall f : f \in \mathit{frontier}(C_{\min}.e) \setminus \{e\} : f \in C_{least}.e \rangle \\
\Leftarrow & \{ \text{ definition of consistent cut and its frontier } \} \\
 & C_{\min}.e \ \subseteq \ C_{least}.e \\
\Leftarrow & \left\{ \begin{array}{l} C_{least}.e \text{ is a consistent cut of } \langle E, \rightarrow \rangle \text{ that passes through } e \text{ and} \\ \text{satisfies } b^{(i)} \text{ and } C_{\min}.e \text{ is the } \textit{least} \text{ such cut} \end{array} \right\} \\
 & (C_{least}.e \text{ is a consistent cut of } \langle E, \rightarrow \rangle) \ \wedge \ (e \in \mathit{frontier}(C_{least}.e)) \wedge \\
 & (C_{least}.e \ \models \ b^{(i)}) \\
\Leftarrow & \{ \ C_{least}.e \text{ is a consistent cut of } \langle E, \rightsquigarrow \rangle \text{ and } \rightarrow \subseteq \rightsquigarrow \ \} \\
 & (e \in \mathit{frontier}(C_{least}.e)) \ \wedge \ (C_{least}.e \ \models \ b^{(i)}) \\
\Leftarrow & \{ \text{ using (4.2) } \} \\
 & C_{least}.e \ \models \ b^{(i)} \\
\Leftarrow & \{ \ b^{(i)} \text{ is a conjunct of } b \ \} \\
 & C_{least}.e \ \models \ b \\
 & \{ \text{ since } b \text{ is invariant in } \langle E, \rightsquigarrow \rangle, \ C_{least}.e \text{ satisfies } b \ \}
\end{array}
$$

Similarly, it can be proved that, for each event e, $\rightsquigarrow$ includes $\stackrel{e^{(2)}}{\rightarrow}$. □

The necessary condition for a region predicate to be controllable in a computation can now be easily derived.

Lemma 4.4 (necessary condition) *If a region predicate b is controllable in a computation $\langle E, \rightarrow \rangle$ then (1) the initial and final consistent cuts of $\langle E, \rightarrow \rangle$ satisfy b, and (2) $\stackrel{C}{\rightarrow}$ is an irreflexive partial order.*

Proof: Since b is controllable in $\langle E, \rightarrow \rangle$, there exists an irreflexive partial order $\leadsto$ that extends $\rightarrow$ such that b is invariant in $\langle E, \leadsto \rangle$. Since $\perp$ and E are also the consistent cuts of $\langle E, \leadsto \rangle$, they satisfy b. Furthermore, from Theorem 4.3, $\stackrel{S}{\rightarrow}$ is the smallest controlling synchronization implying that $\leadsto$ contains $\stackrel{S}{\rightarrow}$. Thus $\leadsto$ contains $(\rightarrow \cup \stackrel{S}{\rightarrow})$. Since $\leadsto$ is an irreflexive partial order, $(\rightarrow \cup \stackrel{S}{\rightarrow})^+$ $(= \stackrel{C}{\rightarrow})$ is also an irreflexive partial order. $\square$

Finally, the next theorem combines the previous two lemmas and furnishes the necessary and sufficient condition for a region predicate to be controllable in a computation.

Theorem 4.5 (necessary and sufficient condition) *A region predicate b is controllable in a computation $\langle E, \rightarrow \rangle$ if and only if (1) the initial and final consistent cuts of $\langle E, \rightarrow \rangle$ satisfy b, and (2) $\stackrel{C}{\rightarrow}$ is an irreflexive partial order.*

It turns out that the controlling synchronization $\stackrel{S}{\rightarrow}$ defined in (4.1) is minimal in another sense. It not only eliminates all unsafe runs of the computation but also does not suppress any safe run. We call such a synchronization *optimal*.

Definition 4.3 (optimal controlling synchronization) *We call a controlling synchronization* optimal *if it does not suppress any safe run of the computation. Formally, given a controlling synchronization $\stackrel{S}{\rightarrow}$ for a predicate b in a computation $\langle E, \rightarrow \rangle$, where $\stackrel{C}{\rightarrow} = (\rightarrow \cup \stackrel{S}{\rightarrow})^+$,*

$$\stackrel{S}{\rightarrow} \textit{ is optimal} \quad \triangleq \quad \langle \forall \leadsto : \leadsto \textit{ is a total order on } E \textit{ that extends } \rightarrow : \\ \langle E, \leadsto \rangle \models \textit{invariant}: b \;\equiv\; \leadsto \textit{ extends } \stackrel{C}{\rightarrow} \rangle$$

In fact, the two aforementioned notions of minimality, namely the smallest and the optimal controlling synchronization, turn out to be identical. We establish their equivalence in the next theorem.

Theorem 4.6 (smallest versus optimal) *A smallest controlling synchronization is also optimal and vice versa.*

Proof: Consider a controlling synchronization $\xrightarrow{S}$ for a predicate b in a computation $\langle E, \rightarrow \rangle$ and let $\xrightarrow{C}$ be $(\rightarrow \cup \xrightarrow{S})^{+}$.

(*optimal* $\Rightarrow$ *smallest*) Assume that $\xrightarrow{S}$ is the optimal controlling synchronization. Consider an irreflexive partial order $\rightsquigarrow$ that extends $\rightarrow$ such that b is invariant in $\langle E, \rightsquigarrow \rangle$. Our obligation is to establish that $\rightsquigarrow$ contains $\xrightarrow{S}$. Let $\mapsto$ be a total order on E that extends $\rightsquigarrow$. Since $\rightsquigarrow$ extends $\rightarrow$, $\mapsto$ also extends $\rightarrow$ implying that $\langle E, \mapsto \rangle$ is a run of $\langle E, \rightarrow \rangle$. Moreover, $\langle E, \mapsto \rangle$ is a safe run of $\langle E, \rightarrow \rangle$ because b is invariant in $\langle E, \rightsquigarrow \rangle$ and therefore also invariant in $\langle E, \mapsto \rangle$. Since $\xrightarrow{S}$ is the optimal controlling synchronization, by definition, $\mapsto$ extends $\xrightarrow{C}$ or, in other words, $\mapsto$ includes $\xrightarrow{S}$. Since $\mapsto$ was chosen arbitrarily, we can infer that every total order on E that extends $\rightsquigarrow$ contains $\xrightarrow{S}$ implying that $\rightsquigarrow$ also contains $\xrightarrow{S}$.

(*smallest* $\Rightarrow$ *optimal*) Assume that $\xrightarrow{S}$ is the smallest controlling synchronization. Consider a safe run $\langle E, \rightsquigarrow \rangle$ of $\langle E, \rightarrow \rangle$. Our obligation is to establish that $\langle E, \rightsquigarrow \rangle$ is also a run of $\langle E, \xrightarrow{C} \rangle$, that is, $\rightsquigarrow$ contains $\xrightarrow{C}$. Note that b is invariant in $\langle E, \rightsquigarrow \rangle$. Since $\xrightarrow{S}$ is the smallest controlling synchronization, by definition, $\rightsquigarrow$ contains $\xrightarrow{S}$. This implies that $\rightsquigarrow$ extends $\xrightarrow{C}$ or $\langle E, \rightsquigarrow \rangle$ is a run of $\langle E, \xrightarrow{C} \rangle$. $\square$

From Theorem 4.3 and Theorem 4.6, we obtain,

Theorem 4.7 *If a region predicate b is controllable in a computation $\langle E, \rightarrow \rangle$ then the synchronization $\stackrel{S}{\rightarrow}$ defined in (4.1) is the optimal controlling synchronization.*

Theorem 4.7 implies that the controlling synchronization $\stackrel{S}{\rightarrow}$ defined in (4.1) is not too restrictive and, in fact, admits the maximum possible concurrency in the controlled computation.

From the earlier discussion, it follows that a controlling synchronization for a region predicate can be easily computed provided, for each event e, we can efficiently compute $C_{\min}.e$ and $C_{\max}.e$, if they exist. To that end, given a p-region predicate b and an event e on process p, we define a predicate b_e to be true for a consistent cut if it passes through e and satisfies b. Formally,

$$C \models b_e \ \triangleq \ (e \in \mathit{frontier}(C)) \wedge (C \models b)$$

It can be verified easily, using the weak lattice property, that if two consistent cuts satisfy b_e then so does the consistent cut given by their set intersection. Chase and Garg [CG98] call such predicates *linear*. Likewise, if two consistent cuts satisfy b_e then the consistent cut given by their set union also satisfies b_e. Such predicates are called post-linear [CG98].

Observation 4.1 *The predicate b_e is linear and post-linear.*

The consistent cuts $C_{\min}.e$ and $C_{\max}.e$ can be reinterpreted as the least and greatest consistent cut, respectively, that satisfy b_e. Chase and Garg [CG98] also provide algorithms to find the least consistent cut that satisfies a linear predicate and the greatest consistent cut that satisfies a post-linear predicate. Here, we focus on the former and give the basic idea behind the algorithm. The correctness proof and other details can be found elsewhere [CG98]. The algorithm is based on the *linearity property* which is defined as follows:

Algorithm Algo$_{4.1}$:

<u>Input</u>: (1) a computation $\langle E, \rightarrow \rangle$, (2) a p-region predicate b, and (3) an event e on process p

<u>Output</u>: $C_{\min}.e$, if it exists

```
C := least consistent cut of ⟨E, →⟩ that passes through e;
done := false;
while not(done) do
    if there exists an event f in frontier(C)
      such that succ(e) → f then
        exit("C_min.e does not exist");
    endif;
    if there exist events f and g, f ≠ e, in frontier(C)
      such that succ(f) → g then          // C is not a consistent cut
        C := C ∪ succ(f);                  // advance beyond f
    else                                   // C is a consistent cut
        if C |= b then done := true;
        else
            f := forbidden_{b_e}(C);       // invoke the linearity property
            if f = e or f ∈ ⊤ then         // cannot advance beyond f
                exit("C_min.e does not exist");
            else C := C ∪ succ(f);         // advance beyond f
            endif;
        endif;
    endif;
endwhile;
C_min.e := C;
```

Figure 4.6: The algorithm Algo$_{4.1}$ to compute $C_{\min}.e$ for an event e.

Algorithm $\mathsf{Algo}_{4.2}$:

Input: a computation $\langle E, \rightarrow \rangle$ and a region predicate b

Output: synchronization necessary to control b in $\langle E, \rightarrow \rangle$, if possible

if either $\perp$ or E does not satisfy b then
    exit("b cannot be controlled in $\langle E, \rightarrow \rangle$");
endif;
for each event e do
    compute $C_{\min}.e$ and $C_{\max}.e$;
    if either $C_{\min}.e$ or $C_{\max}.e$ does not exist then
        exit("b cannot be controlled in $\langle E, \rightarrow \rangle$");
    endfor;
endfor;
compute the synchronization $\xrightarrow{S}$ defined in (4.1);
if $(\rightarrow \cup \xrightarrow{S})$ is acyclic then
    exit($\xrightarrow{S}$);
else
    exit("b cannot be controlled in $\langle E, \rightarrow \rangle$");
endif;

Figure 4.7: The algorithm $\mathsf{Algo}_{4.2}$ to compute the synchronization necessary to control a region predicate in a computation.

Definition 4.4 (linearity property [CG98]) *A predicate satisfies the* linearity property *if, given a consistent cut that does not satisfy the predicate, there exists an event in its frontier, called the* forbidden *event, such that there does not exist a consistent cut containing the given consistent cut that satisfies the predicate and also passes through the forbidden event. Formally, given a computation $\langle E, \rightarrow \rangle$, a linear predicate b and a consistent cut C,*

$$C \not\models b \;\Rightarrow\; \langle \exists\, f : f \in \mathit{frontier}(C) : \langle \forall\, D : D \supseteq C : D \models b \Rightarrow \mathit{succ}(f) \in D \rangle \rangle$$

It is assumed that, given a linear predicate b, there is an efficient partial function $\mathit{forbidden}_b : \mathcal{C}(\langle E, \rightarrow \rangle) \longrightarrow E$ that can be used to compute the event f mentioned in the definition of the linearity property. It is hard to provide a general algorithm to compute the function that works for any linear predicate. Nevertheless, for the linear predicates encountered in practice, an efficient algorithm can indeed be given. For example, for a conjunctive predicate—a conjunction of local predicates—the forbidden event corresponds to that event in the cut's frontier for which the local predicate evaluates to false. Throughout this dissertation, we assume that a linear predicate also satisfies the *advancing property* which guarantees the existence of an efficient function to compute the forbidden event.

Figure 4.6 describes the algorithm $\mathsf{Algo}_{4.1}$ to compute $C_{\min}.e$ using the linearity property. Informally, starting from the least consistent cut that passes through e—which basically corresponds to the Fidge/Mattern's vector timestamp for e [Mat89, Fid91], the algorithm scans the computation from left to right adding events to the cut constructed so far one-by-one, using the linearity property, until the desired consistent cut is reached.

The time-complexity analysis of the algorithm $\mathsf{Algo}_{4.1}$ is as follows. Each iteration of the while loop at line 3 has $O(n)$ time complexity assuming that the time-complexity of invoking $\mathit{forbidden}_{b_e}$ at line 9 once is $O(n)$. Thus the time-complexity of the algorithm $\mathsf{Algo}_{4.1}$ for computing $C_{\min}.e$ is $O(n|E|)$. The algorithm to compute $C_{\max}.e$, based on the *post-linearity property* [CG98], is similar and has been omitted.

Figure 4.7 depicts the algorithm $\mathsf{Algo}_{4.2}$ that computes a synchronization for controlling a region predicate in a computation. The correctness of the algorithm follows from Theorem 4.5. Its time-complexity analysis is as follows. The time-complexity of executing the if statement at line 1 is $O(n)$. Each iteration of the for loop at line 3 has $O(n|E|)$ time-complexity giving the for loop an overall time-complexity of $O(n|E|^2)$. The synchronization at line 7 can be computed in $O(n|E|)$

time. Finally, the if statement at line 8 can be executed in $O(|E|^2)$ time. Thus the overall time-complexity of the algorithm $\mathsf{Algo}_{4.2}$ is $O(n|E|^2)$.

4.4 Disjunctive Predicates

A predicate is said to be *disjunctive* if it can be expressed as disjunction of local predicates. Some examples of disjunctive predicates are:

- at least one server is available: $avail_1 \vee avail_2 \vee \cdots \vee avail_n$
- at least one philosopher has no fork: $\neg fork_1 \vee \neg fork_2 \vee \cdots \vee \neg fork_n$

Intuitively, a disjunctive predicate states that at least one local condition must be met at all times, or, in other words, a bad combination of local conditions does not occur. Our algorithm for computing a controlling synchronization for a disjunctive predicate utilizes the notion of admissible sequence defined next.

4.4.1 Admissible Sequences

In this section, we establish that the notion of controllability is actually identical to the notion of admissible sequence whose motivation in turn lies in the control algorithm for a disjunctive predicate. We make the following observation:

Observation 4.2 *A consistent cut satisfies a disjunctive predicate if and only if it contains at least one true event in its frontier.*

Suppose we wish to control a disjunctive predicate in a computation. As the computation proceeds from the initial consistent cut to the final consistent cut, from the above observation it follows that it is both necessary and sufficient to ensure that throughout there exists at least one true event in the frontier of the computation. Thus at least one initial event must be a true event. To start with, one such initial

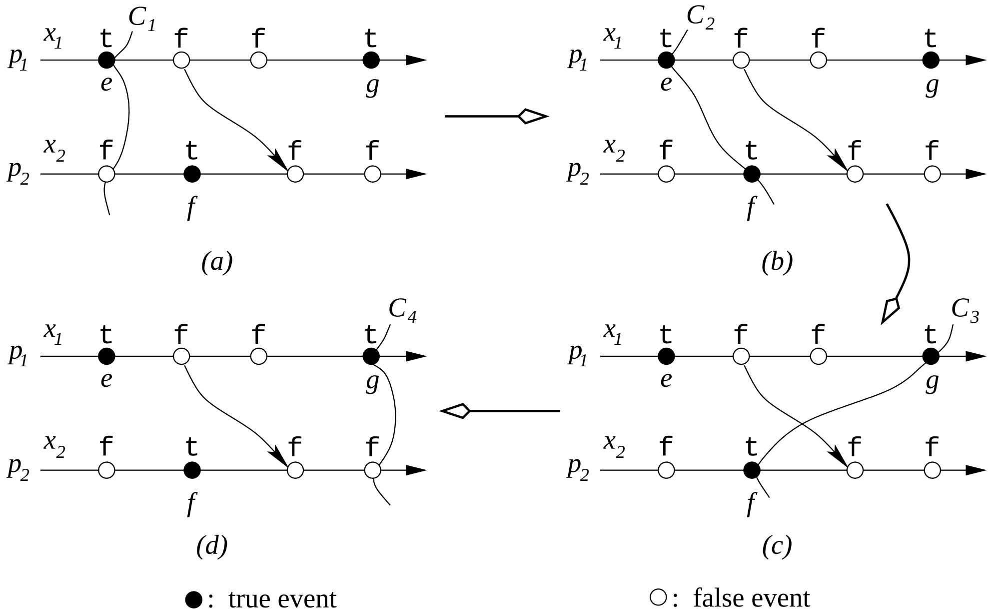


Figure 4.8: A strategy for controlling a disjunctive predicate.

event bears the responsibility for ensuring that the predicate stays true—by acting as an *anchor*—until the burden can be passed on to some other true event. This transference of burden continues until the computation reaches the final consistent cut.

Example 4.4 *We want to control the disjunctive predicate* $x_1 \vee x_2$ *in the computation depicted in Figure 4.8. The initial event* e *is a true event. Hence, using* e *as an anchor, the computation advances from the initial consistent cut* C_1*, shown in Figure 4.8(a), to the consistent cut* C_2*, portrayed in Figure 4.8(b). Next, using the true event* f *as an anchor, it advances to the consistent cut* C_3 *as shown in Figure 4.8(c). Finally, using the true event* g *as an anchor—which is also a final event, it reaches the final consistent cut* C_4 *as depicted in Figure 4.8(d). Since throughout the frontier of the computation passes through at least one true event, the predicate is never falsified.*

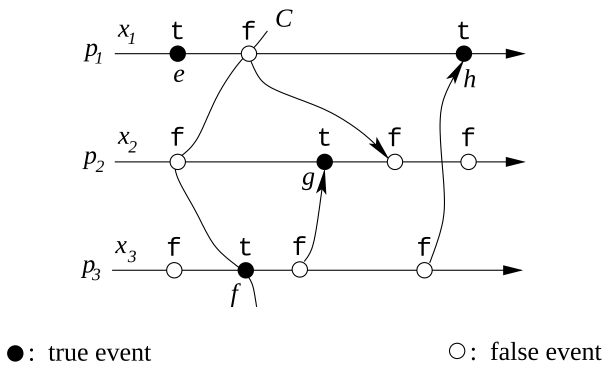


Figure 4.9: An example to illustrate the difficulty in choosing the next anchor event.

A natural question to ask is: "If there are more than one possible candidates for the next anchor event, which one should we choose?". The answer is non-trivial as illustrated by the following example.

Example 4.5 *Consider the computation shown in Figure 4.9. It has four true events, namely e, f, g and h. After using e as an anchor, the computation has two possible choices of events for the next anchor. They are the events f and g. The event h is unavailable because the computation has to advance beyond e before it can execute h. Clearly, f is a bad choice for anchor because once the computation reaches the consistent cut C, using f as an anchor, neither g nor h can be used as the next anchor without falsifying the predicate.*

The notion of admissible sequence attempts to answer the above question in a more general setting. In the next section, we formalize the aforementioned algorithm for controlling a disjunctive predicate using the notion of admissible sequence. We first define a legal cut as follows:

Definition 4.5 (legal cut) *A consistent cut is* legal *with respect to a sequence of events if it contains an event from the sequence only if it contains all its preceding* *events from the sequence too. Formally, given a consistent cut C and an event s_i from a sequence of events s,*

$$s_i \in C \;\Rightarrow\; \langle \forall\, j : j \leqslant i : s_j \in C \rangle$$

Roughly speaking, the notion of legal cut helps to capture those runs of a computation that respect the order of the events in a sequence. More precisely, given a sequence of events, if every consistent cut of a run is legal then the run and the sequence do not disagree on relative order of any pair of events and vice versa. We next define the notion of admissible sequence. Informally, every event in an admissible sequence acts as an anchor in the order given by the sequence. To be able to do so, the sequence must respect the happened-before order between events. This constraint is captured by the *agreement property*. The *continuity property* ensures that the transfer of burden from one event in the sequence to the next occurs "smoothly" in a single step. In other words, the computation does not advance beyond the current anchor event until it reaches the next anchor event. The *weak safety property* ascertains that, on reaching an anchor event, at least as long as the computation does not advance beyond the event the predicate is not falsified. Finally, the *boundary condition* captures the fact that the initial and final consistent cuts satisfy the predicate. Formally,

Definition 4.6 (admissible sequence) *A sequence of events $s = s_1 s_2 \cdots s_{l-1} s_l$ is* admissible *with respect to a predicate b and a computation $\langle E, \rightarrow \rangle$ if it satisfies the following properties:*

- **(boundary condition)** *The sequence starts with an initial event ends with a final event of the computation. Formally,*

$$(s_1 \in \bot) \;\wedge\; (s_l \in \top)$$

- **(agreement)** *The sequence respects the partial order (that is, happened-before relation) of the computation. Formally,*

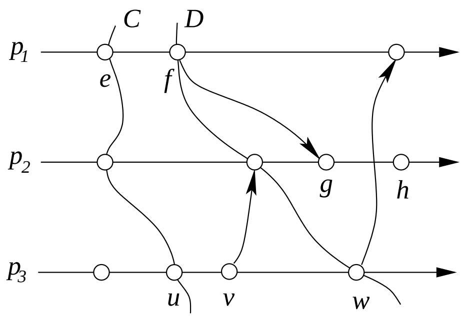


Figure 4.10: An example to illustrate the notion of legal cut and admissible sequence.

$$\langle \forall\, i, j : i < j : s_j \not\rightarrow s_i \rangle$$

- **(continuity)** *The successor of each event in the sequence, if it exists, did not happen-before the next event in the sequence. Formally,*

$$\langle \forall\, i : s_i \notin \top : succ(s_i) \not\rightarrow s_{i+1} \rangle$$

- **(weak safety)** *Any consistent cut of the computation that is legal with respect to the sequence and contains at least one event from the sequence in its frontier satisfies the predicate. Formally,*

$$\langle \forall\, C : C \text{ is legal with respect to } s : (s \cap frontier(C)) \neq \emptyset \;\Rightarrow\; C \models b \rangle$$

Example 4.6 *Consider the computation depicted in Figure 4.10. The consistent cut C is not legal with respect to the sequence of events $efuvh$ because it contains u but does not contain f which occurs before u in the sequence. On the other hand, the consistent cut D is legal with respect to the same sequence. The sequence $fuvh$ does not satisfy the boundary condition because the first event in the sequence, in this case f, is not an initial event. The sequence $egfh$ does not satisfy the agreement property because although f happened-before g in the computation, it occurs after g in the sequence. Finally, the sequence egh does not satisfy the continuity property as the successor of e, namely f, happened-before g, the next event in the sequence after e.*

The following theorem proves that existence of an admissible sequence is necessary for a predicate to be controllable in a computation. Specifically, we prove that any safe run of a computation constitutes an admissible sequence.

Theorem 4.8 (necessary condition) *If a predicate b can be controlled in a computation $\langle E, \rightarrow \rangle$ then there exists an admissible sequence with respect to b and $\langle E, \rightarrow \rangle$.*

Proof: Since b is controllable in $\langle E, \rightarrow \rangle$, there exists a total order $\leadsto$ that extends $\rightarrow$ such that b is invariant in $\langle E, \leadsto \rangle$. Let s be the sequence of events corresponding to $\langle E, \leadsto \rangle$. We prove that s is admissible with respect to b and $\langle E, \rightarrow \rangle$. Clearly, s satisfies the boundary condition and the agreement property. We next prove that s satisfies the continuity property. Assume the contrary. Then,

$$
\begin{array}{ll}
 & \langle \exists\, i :: succ(s_i) \rightarrow s_{i+1} \rangle \\
\equiv & \{\ s_i \rightarrow succ(s_i)\ \} \\
 & \langle \exists\, i :: s_i \rightarrow succ(s_i) \rightarrow s_{i+1} \rangle \\
\Rightarrow & \{\ succ(s_i) \in s \text{ because } s \text{ corresponds to } \langle E, \leadsto \rangle\text{—a run of } \langle E, \rightarrow \rangle\ \} \\
 & \langle \exists\, i, j :: s_i \rightarrow s_j \rightarrow s_{i+1} \rangle \\
\Rightarrow & \{\ s \text{ satisfies the agreement property}\ \} \\
 & \langle \exists\, i, j :: i < j < i+1 \rangle \\
\Rightarrow & \{\ i \text{ and } j \text{ are integers}\ \} \\
 & \text{a contradiction}
\end{array}
$$

Finally, we show that s satisfies the weak safety property. Consider a consistent cut C of $\langle E, \rightarrow \rangle$ that is legal with respect to s. We prove that C is also a consistent cut of $\langle E, \leadsto \rangle$. Consider events e and f. We have,

$$
\begin{array}{l}
\{\ \text{assumption}\ \} \\
(e \leadsto f)\ \wedge\ (f \in C)
\end{array}
$$

$$
\begin{array}{cl}
\equiv & \{ \text{ let } e = s_i \text{ and } f = s_j \ \} \\
 & (s_i \rightsquigarrow s_j) \ \wedge \ (s_j \in C) \\
\Rightarrow & \{ \text{ definition of } s \ \} \\
 & (i < j) \ \wedge \ (s_j \in C) \\
\Rightarrow & \{ \ C \text{ is legal with respect to } s \ \} \\
 & s_i \in C \\
\equiv & \{ \ s_i = e \ \} \\
 & e \in C
\end{array}
$$

Thus C is a consistent cut of $\langle E, \rightsquigarrow \rangle$. Since b is invariant in $\langle E, \rightsquigarrow \rangle$, C satisfies b. This establishes that s satisfies the weak safety property. □

Our next step is to prove that the existence of an admissible sequence is also a sufficient condition for a predicate to be controllable in a computation. To achieve that it suffices to give the synchronization necessary to control the predicate. Of course the synchronization will depend on the particular sequence. Observe that not all events in the sequence may be ordered by the happened-before relation. Thus, to ensure that they are executed in the order they occur in the sequence, we need to add synchronization dependencies from an event in the sequence to all other events that occur later in the sequence. This synchronization is denoted by $\stackrel{S^{(1)}}{\rightarrow}$ and is formally defined as follows:

$$
\stackrel{S^{(1)}}{\rightarrow} \ \triangleq \ \{ \, (s_i, s_j) \mid 1 \leqslant i < j \leqslant n \, \} \tag{4.3}
$$

For an example please refer to Figure 4.11. In the following lemma we show that if the sequence is admissible, in particular if it satisfies the agreement property, the above synchronization does not interfere with the happened-before relation of the computation. For convenience, we define $\stackrel{C^{(1)}}{\rightarrow}$ as the transitive closure of $\rightarrow \cup \stackrel{S^{(1)}}{\rightarrow}$. Formally,

$$
\stackrel{C^{(1)}}{\rightarrow} \ \triangleq \ \left(\rightarrow \cup \stackrel{S^{(1)}}{\rightarrow} \right)^{+}
$$

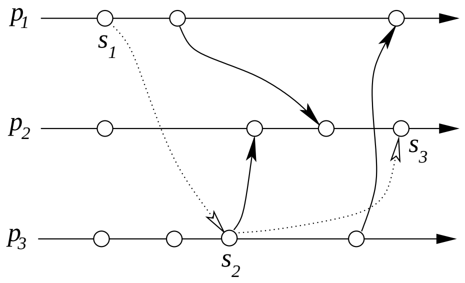


Figure 4.11: An illustration of the synchronization $\stackrel{S^{(1)}}{\rightarrow}$ (denoted by dotted arrows).

Lemma 4.9 $\stackrel{C^{(1)}}{\rightarrow}$ *is an irreflexive partial order.*

Proof: It suffices to prove that $\rightarrow \cup \stackrel{S^{(1)}}{\rightarrow}$ does not contain any cycle. Since $\rightarrow$ is an irreflexive partial order, a cycle, if it exists, must contain at least one pair of events ordered by $\stackrel{S^{(1)}}{\rightarrow}$. Moreover, since both $\rightarrow$ and $\stackrel{S^{(1)}}{\rightarrow}$ are transitive, the pairs of events in the cycle must be alternately ordered by $\rightarrow$ and $\stackrel{S^{(1)}}{\rightarrow}$. We first prove that there is no cycle containing exactly one pair of events ordered by $\stackrel{S^{(1)}}{\rightarrow}$. Assume the contrary. Then,

$$
\begin{array}{ll}
 & \langle \exists\, i, j :: s_i \stackrel{S^{(1)}}{\rightarrow} s_j \underline{\rightarrow} s_i \rangle \\
\Rightarrow & \{ \text{ definition of } \stackrel{S^{(1)}}{\rightarrow} \} \\
 & \langle \exists\, i, j :: (i < j) \ \wedge \ (s_j \underline{\rightarrow} s_i) \rangle \\
\Rightarrow & \{ \ s \text{ satisfies the agreement property } \} \\
 & \langle \exists\, i, j :: (s_j \not\underline{\rightarrow} s_i) \ \wedge \ (s_j \underline{\rightarrow} s_i) \rangle \\
\Rightarrow & \{ \text{ predicate calculus } \} \\
 & \text{a contradiction}
\end{array}
$$

We now prove that if there is a cycle that contains m, $m \geqslant 2$, pairs of events ordered by $\stackrel{S^{(1)}}{\rightarrow}$ then there is a cycle that contains strictly fewer than m pairs of events ordered by $\stackrel{S^{(1)}}{\rightarrow}$. Let the cycle be $s_i \stackrel{S^{(1)}}{\rightarrow} s_j \rightarrow s_u \stackrel{S^{(1)}}{\rightarrow} s_v \stackrel{C^{(1)}}{\underline{\rightarrow}} s_i$, where the path from s_v to s_i contains exactly $m - 2$ pair(s) of events ordered by $\stackrel{S^{(1)}}{\rightarrow}$. Since $\stackrel{S^{(1)}}{\rightarrow}$ is a total order, either $s_i \overset{S^{(1)}}{\underline{\rightarrow}} s_v$ or $s_v \overset{S^{(1)}}{\underline{\rightarrow}} s_i$. We have,

Case 1: $s_i \overset{S^{(1)}}{\underline{\rightarrow}} s_v$

$$
\begin{aligned}
& (s_i \overset{S^{(1)}}{\rightarrow} s_j \rightarrow s_u \overset{S^{(1)}}{\rightarrow} s_v \overset{C^{(1)}}{\underline{\rightarrow}} s_i) \wedge (s_i \overset{S^{(1)}}{\underline{\rightarrow}} s_v) \\
\Rightarrow\ & \{ \text{ simplifying } \} \\
& s_i \overset{S^{(1)}}{\underline{\rightarrow}} s_v \overset{C^{(1)}}{\underline{\rightarrow}} s_i \\
\Rightarrow\ & \{ \text{ simplifying } \} \\
& \text{a cycle with at most } m-1 \text{ pair(s) of events ordered by } \overset{S^{(1)}}{\rightarrow}
\end{aligned}
$$

Case 2: $s_v \overset{S^{(1)}}{\underline{\rightarrow}} s_i$

$$
\begin{aligned}
& (s_i \overset{S^{(1)}}{\rightarrow} s_j \rightarrow s_u \overset{S^{(1)}}{\rightarrow} s_v \overset{C^{(1)}}{\underline{\rightarrow}} s_i) \wedge (s_v \overset{S^{(1)}}{\underline{\rightarrow}} s_i) \\
\Rightarrow\ & \{ \text{ simplifying } \} \\
& s_i \overset{S^{(1)}}{\rightarrow} s_j \rightarrow s_u \overset{S^{(1)}}{\rightarrow} s_v \overset{S^{(1)}}{\underline{\rightarrow}} s_i \\
\equiv\ & \{ \text{ rewriting } \} \\
& s_j \rightarrow s_u \overset{S^{(1)}}{\rightarrow} s_v \overset{S^{(1)}}{\underline{\rightarrow}} s_i \overset{S^{(1)}}{\rightarrow} s_j \\
\Rightarrow\ & \{ \overset{S^{(1)}}{\rightarrow} \text{ is transitive } \} \\
& s_j \rightarrow s_u \overset{S^{(1)}}{\rightarrow} s_j \\
\Rightarrow\ & \{ \text{ simplifying } \} \\
& \text{a cycle with at most one pair of events ordered by } \overset{S^{(1)}}{\rightarrow}
\end{aligned}
$$

This establishes that there is no cycle in $\rightarrow \cup \overset{S^{(1)}}{\rightarrow}$ and thus $\overset{C^{(1)}}{\rightarrow}$ is an irreflexive partial order. $\square$

After adding the synchronization $\overset{S^{(1)}}{\rightarrow}$ to the computation $\langle E, \rightarrow \rangle$, the resulting computation $\langle E, \overset{C^{(1)}}{\rightarrow} \rangle$ retains only those consistent cuts—not necessarily all—that are legal. From the weak safety property, a sufficient condition for a legal cut to satisfy the predicate is that it should contain at least one event from the sequence in its frontier. To ensure this, given an event in the sequence, we add a

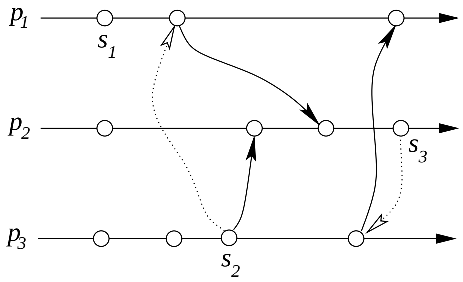


Figure 4.12: An illustration of the synchronization $\overset{S^{(2)}}{\rightarrow}$ (denoted by dotted arrows).

synchronization arrow from the event next to it in the sequence, if it exists and is on a different process, to its succeeding event on the process, if it exists. This synchronization, denoted by $\overset{S^{(2)}}{\rightarrow}$, ascertains that the computation does not advance beyond an event in the sequence until it reaches the next event in the sequence.

$$\overset{S^{(2)}}{\rightarrow} \;\triangleq\; \{\, (s_{i+1}, succ(s_i)) \mid 1 \leqslant i < n,\ s_i \notin \top \ \text{ and }\ proc(s_{i+1}) \neq proc(s_i) \,\} \tag{4.4}$$

For an illustration please see Figure 4.12. In the next lemma we establish that if the sequence is admissible, in particular if it satisfies the agreement and continuity properties, the above synchronization $\overset{S^{(2)}}{\rightarrow}$ does not interfere with $\overset{C^{(1)}}{\rightarrow}$. For convenience, we define $\overset{C^{(2)}}{\rightarrow}$ as the transitive closure of $\overset{C^{(1)}}{\rightarrow} \cup \overset{S^{(3)}}{\rightarrow}$. Formally,

$$\overset{C^{(2)}}{\rightarrow} \;\triangleq\; \left(\overset{C^{(1)}}{\rightarrow} \cup \overset{S^{(2)}}{\rightarrow}\right)^{+}$$

Lemma 4.10 $\overset{C^{(2)}}{\rightarrow}$ *is an irreflexive partial order.*

Proof: It suffices to prove that $\overset{C^{(1)}}{\rightarrow} \cup \overset{S^{(2)}}{\rightarrow}$ does not contain any cycle. Since, from Lemma 4.9, $\overset{C^{(1)}}{\rightarrow}$ is an irreflexive partial order, a cycle, if it exists, must contain at least one pair of events ordered by $\overset{S^{(2)}}{\rightarrow}$. We first prove that there is no cycle containing exactly one pair of events ordered by $\overset{S^{(2)}}{\rightarrow}$. Assume the contrary. We have,

$$\langle \exists\, i :: s_{i+1} \overset{S^{(2)}}{\rightarrow} succ(s_i) \overset{C^{(1)}}{\underline{\rightarrow}} s_{i+1} \rangle$$

$\Rightarrow$ { by definition of $\overset{S^{(2)}}{\to}$, $proc(s_{i+1}) \neq proc(s_i)$ implying $s_{i+1} \neq succ(s_i)$ }

$\langle \exists\, i :: s_{i+1} \overset{S^{(2)}}{\to} succ(s_i) \overset{C^{(1)}}{\to} s_{i+1} \rangle$

$\Rightarrow$ { since s satisfies the continuity property, $succ(s_i) \not\to s_{i+1}$ }

$\langle \exists\, i, j, k :: s_{i+1} \overset{S^{(2)}}{\to} succ(s_i) \underline{\to}\, s_j \overset{S^{(1)}}{\to} s_k \overset{C^{(1)}}{\underline{\to}} s_{i+1} \rangle$

$\Rightarrow$ { $\overset{S^{(1)}}{\to}$ is a total order on s }

$\langle \exists\, i, j ::(s_{i+1} \overset{S^{(2)}}{\to} succ(s_i) \underline{\to}\, s_j \overset{C^{(1)}}{\to} s_{i+1}) \bigwedge$
$((s_{i+1} \overset{S^{(1)}}{\underline{\to}} s_j) \vee (s_j \overset{S^{(1)}}{\to} s_{i+1}))\rangle$

$\Rightarrow$ { $s_{i+1} \overset{S^{(1)}}{\underline{\to}} s_j$ implies $s_{i+1} \overset{S^{(1)}}{\underline{\to}} s_j \overset{C^{(1)}}{\to} s_{i+1}$—contradicting Lemma 4.9 }

$\langle \exists\, i, j :: (s_{i+1} \overset{S^{(2)}}{\to} succ(s_i) \underline{\to}\, s_j \overset{C^{(1)}}{\to} s_{i+1}) \wedge (s_j \overset{S^{(1)}}{\to} s_{i+1})\rangle$

$\Rightarrow$ { $s_i \overset{P}{\to} succ(s_i)$ and $\overset{P}{\to} \subseteq \to$ }

$\langle \exists\, i, j :: (s_i \to s_j) \wedge (s_j \overset{S^{(1)}}{\to} s_{i+1})\rangle$

$\Rightarrow$ { $\overset{S^{(1)}}{\to}$ is a total order on s and s satisfies the agreement property }

$\langle \exists\, i, j :: (s_i \overset{S^{(1)}}{\to} s_j) \wedge (s_j \overset{S^{(1)}}{\to} s_{i+1})\rangle$

$\Rightarrow$ { s satisfies the agreement property }

$\langle \exists\, i, j :: i < j < i+1 \rangle$

$\Rightarrow$ { i and j are integers }

a contradiction

We now prove that if there is a cycle that contains m, $m \geqslant 2$, pairs of events ordered by $\overset{S^{(2)}}{\to}$ then there is a cycle that contains strictly fewer than m pairs of events ordered by $\overset{S^{(2)}}{\to}$. Let the cycle be $s_{i+1} \overset{S^{(2)}}{\to} succ(s_i) \overset{C^{(1)}}{\underline{\to}} s_{j+1} \overset{S^{(2)}}{\to} succ(s_j) \overset{C^{(2)}}{\underline{\to}} s_{i+1}$, where the path from $succ(s_j)$ to s_{i+1} contains exactly $m-2$ pair(s) of events ordered by $\overset{S^{(2)}}{\to}$. Since $\overset{S^{(1)}}{\to}$ is a total order, either $s_{i+1} \overset{S^{(1)}}{\underline{\to}} s_{j+1}$ or $s_{j+1} \overset{S^{(1)}}{\underline{\to}} s_{i+1}$. We have,

Case 1: $s_{i+1} \overset{S^{(1)}}{\underline{\to}} s_{j+1}$

$(s_{i+1} \overset{S^{(2)}}{\to} succ(s_i) \overset{C^{(1)}}{\underline{\to}} s_{j+1} \overset{S^{(2)}}{\to} succ(s_j) \overset{C^{(2)}}{\underline{\to}} s_{i+1}) \wedge (s_{i+1} \overset{S^{(1)}}{\underline{\to}} s_{j+1})$

$\Rightarrow$ { simplifying }

$s_{i+1} \overset{S^{(1)}}{\underline{\to}} s_{j+1} \overset{S^{(2)}}{\to} succ(s_j) \overset{C^{(2)}}{\underline{\to}} s_{i+1}$

$\Rightarrow$ { simplifying }

a cycle with at most $m - 1$ pair(s) of events ordered by $\stackrel{S^{(2)}}{\rightarrow}$

Case 2: $s_{j+1} \stackrel{S^{(1)}}{\rightrightarrows} s_{i+1}$

$$(s_{i+1} \stackrel{S^{(2)}}{\rightarrow} succ(s_i) \stackrel{C^{(1)}}{\rightrightarrows} s_{j+1} \stackrel{S^{(2)}}{\rightarrow} succ(s_j) \stackrel{C^{(2)}}{\rightrightarrows} s_{i+1}) \wedge (s_{j+1} \stackrel{S^{(1)}}{\rightrightarrows} s_{i+1})$$

$\Rightarrow$ { simplifying }

$s_{i+1} \stackrel{S^{(2)}}{\rightarrow} succ(s_i) \stackrel{C^{(1)}}{\rightrightarrows} s_{j+1} \stackrel{S^{(1)}}{\rightrightarrows} s_{i+1}$

$\Rightarrow$ { simplifying }

a cycle with at most one pair of events ordered by $\stackrel{S^{(2)}}{\rightarrow}$

This establishes that there is no cycle in $\stackrel{C^{(1)}}{\rightarrow} \cup \stackrel{S^{(2)}}{\rightarrow}$ and thus $\stackrel{C^{(2)}}{\rightarrow}$ is an irreflexive partial order. $\square$

The final step is to prove that the combined synchronization, given by $\stackrel{S^{(1)}}{\rightarrow} \cup \stackrel{S^{(2)}}{\rightarrow}$, indeed ensures that the predicate is invariant in the resulting computation. Specifically, we show that if the sequence is admissible then every consistent of the resultant computation satisfies the antecedent of the weak safety property. We denote the controlled computation by $\langle E, \stackrel{C}{\rightarrow} \rangle$, where $\stackrel{C}{\rightarrow}$ is same as $\stackrel{C^{(2)}}{\rightarrow}$.

Lemma 4.11 *Every consistent cut of* $\langle E, \stackrel{C}{\rightarrow} \rangle$ *satisfies* b.

Proof: Consider a consistent cut C of $\langle E, \stackrel{C}{\rightarrow} \rangle$. We first prove that C is legal with respect to s. Consider events s_i and s_j. We have,

{ assumption }

$(s_j \in C) \wedge (i < j)$

$\equiv$ { definition of $\stackrel{S^{(1)}}{\rightarrow}$ }

$(s_j \in C) \wedge (s_i \stackrel{S^{(1)}}{\rightarrow} s_j)$

$\Rightarrow$ { $\stackrel{S^{(1)}}{\rightarrow} \subseteq \stackrel{C}{\rightarrow}$ }

$(s_j \in C) \wedge (s_i \stackrel{C}{\rightarrow} s_j)$

$$\Rightarrow \quad \{\ C \text{ is a consistent cut of } \langle E, \xrightarrow{C} \rangle\ \}$$
$$s_i \in C$$

This establishes that C is legal with respect to s. We now prove that the frontier of C contains at least one event from s. To that end, we first prove that, for each i, $s_i \notin \top$ implies $s_{i+1} \xrightarrow{C} succ(s_i)$. Clearly, if $proc(s_{i+1}) \neq proc(s_i)$ then, by definition of $\xrightarrow{S^{(2)}}$, $s_{i+1} \xrightarrow{S^{(2)}} succ(s_i)$. Since $\xrightarrow{S^{(2)}} \subseteq \xrightarrow{C}$, $s_{i+1} \xrightarrow{C} succ(s_i)$. The more interesting case is when $proc(s_{i+1}) = proc(s_i)$. Since $proc(s_i) = proc(succ(s_i))$, $proc(s_{i+1}) = proc(succ(s_i))$. Then,

$$\{\ \text{events on a process are totally ordered by } \xrightarrow{P}\ \}$$
$$(s_{i+1} \xrightarrow{P} succ(s_i)) \ \vee\ (succ(s_i) \xrightarrow{P} s_{i+1})$$
$$\Rightarrow \quad \{\ \xrightarrow{P} \subseteq \rightarrow\ \}$$
$$(s_{i+1} \rightarrow succ(s_i)) \ \vee\ (succ(s_i) \rightarrow s_{i+1})$$
$$\Rightarrow \quad \{\ \text{since } s \text{ satisfies the continuity property, } succ(s_i) \not\rightarrow s_{i+1}\ \}$$
$$s_{i+1} \rightarrow succ(s_i)$$
$$\Rightarrow \quad \{\ \rightarrow \subseteq \xrightarrow{C}\ \}$$
$$s_{i+1} \xrightarrow{C} succ(s_i)$$

Assume, on the contrary, that the frontier of C does not contain any event from s. We prove by induction on i that, for each i, $s_i \in C$. Clearly, since s satisfies the boundary condition and $\bot \subseteq C$, $s_1 \in C$. We have,

$$\{\ \text{induction hypothesis}\ \}$$
$$s_i \in C$$
$$\equiv \quad \{\ \text{since } s_i \notin \mathit{frontier}(C),\ succ(s_i) \text{ exists and it belongs to } C\ \}$$
$$succ(s_i) \in C$$
$$\Rightarrow \quad \{\ s_{i+1} \xrightarrow{C} succ(s_i)\ \}$$
$$(s_{i+1} \xrightarrow{C} succ(s_i)) \ \wedge\ (succ(s_i) \in C)$$

$\Rightarrow$ { C is a consistent cut of $\langle E, \xrightarrow{C} \rangle$ }

$s_{i+1} \in C$

This establishes that $s_l \in C$. Since, since s satisfies the boundary condition, $s_l \in \top$. Thus, trivially, $s_l \in \mathit{frontier}(C)$—a contradiction. This implies that the frontier of C contains at least one event from s. Finally, since s satisfies the weak safety property, C satisfies b. $\Box$

Combining Lemma 4.9, Lemma 4.10 and Lemma 4.11, we obtain,

Theorem 4.12 (sufficient condition) *If there exists an admissible sequence with respect to a predicate b and a computation $\langle E, \rightarrow \rangle$ then b is controllable in $\langle E, \rightarrow \rangle$.*

Finally, from Theorem 4.8 and Theorem 4.12, it follows that,

Theorem 4.13 (necessary and sufficient condition) *It is possible to control a predicate b in a computation $\langle E, \rightarrow \rangle$ if and only if there exists an admissible sequence with respect to b and $\langle E, \rightarrow \rangle$.*

Although the motivation for defining the notion of admissible sequence was to devise a control algorithm for a disjunctive predicate, nonetheless the preceding theorem holds for *any* global predicate.

4.4.2 Finding a Controlling Synchronization

In this section, we derive an efficient algorithm for controlling a disjunctive predicate in a computation by using the notion of admissible sequence defined before. Since false is a local predicate of any process, a disjunctive predicate b can be written as disjunction of n predicates such that the i^{th} disjunct, denoted by $b^{(i)}$, is a local predicate of process p_i. The algorithm involves constructing a directed graph G, called the *true event graph*, as follows:

$$\begin{aligned} \mathsf{V}(G) &\triangleq \{\, e \mid e \models b^{(i)}, \text{ where } p_i = proc(e) \,\} \\ \mathsf{E}(G) &\triangleq \{(e,f) \mid e, f \in \mathsf{V}(G),\ e \neq f \ \text{ and } \ e \not\in \top \ \Rightarrow \ succ(e) \not\rightarrow f \,\} \end{aligned}$$

Here, $\mathsf{V}(G)$ and $\mathsf{E}(G)$ refer to the set of vertices and edges, respectively, of the graph G. We now define the notion of permissible path which is almost identical to the notion of admissible sequence except that a permissible path consists of true events only and may not satisfy the agreement property.

Definition 4.7 (permissible path) *A path in a true event graph (TEG) is* permissible *if it starts with an initial event and ends with a final event of the computation.*

Clearly, a permissible path satisfies the boundary condition as well as the continuity property. Furthermore, any consistent cut that contains a true event in its frontier, due to the semantics of disjunction, satisfies the predicate. Thus, a permissible path satisfies the weak safety property also. However, in general, a permissible may not satisfy the agreement property. But if a path besides being permissible is also the shortest one then it satisfies the agreement property too.

Example 4.7 *The true event graph for the computation shown in Figure 4.13(a) and the disjunctive predicate $x_1 \vee x_2$ is depicted in Figure 4.13(b). The path $eghfu$ is permissible but does not satisfy the agreement property because although f happened-before g in the computation, it occurs after g in the path. The path egu is the shortest permissible path. It can be verified that it indeed satisfies the agreement property.*

Lemma 4.14 *The shortest permissible path in a true event graph, if it exists, satisfies the agreement property.*

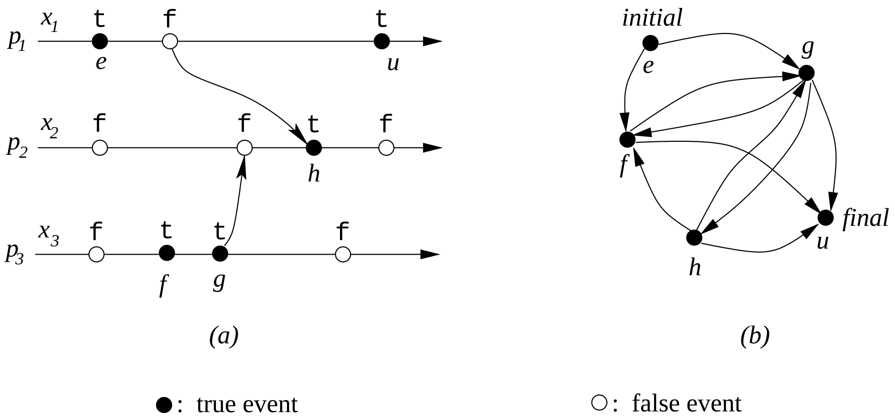


Figure 4.13: An algorithm to compute a controlling synchronization for a disjunctive predicate (edges to initial events and from final events have been omitted).

Proof: Assume that the true event graph does contain a permissible path. Consider the shortest permissible path $s = s_1 s_2 \cdots s_l$. Assume, on the contrary, that s does not satisfy the agreement property. Then,

$$
\begin{aligned}
& \langle \exists\, i, j : i < j : s_j \rightarrow s_i \rangle \\
\Rightarrow\ & \{\ s_j \notin \bot, \text{ otherwise } s_j s_{j+1} \cdots s_l \text{ is a shorter permissible path than } s\ \} \\
& \langle \exists\, i, j : i < j : (s_j \rightarrow s_i)\ \wedge\ (s_j \notin \bot) \rangle \\
\Rightarrow\ & \{\ i \geqslant 2, \text{ otherwise } s_i \in \bot \text{ implying } s_i \rightarrow s_j\text{—creating a cycle in } \rightarrow\ \} \\
& \langle \exists\, i, j : 2 \leqslant i < j : (s_j \rightarrow s_i)\ \wedge\ (s_j \notin \bot) \rangle \\
\Rightarrow\ & \{\ \text{since } s \text{ is the shortest permissible path, } (s_{i-1}, s_j) \notin \mathsf{E}(G)\ \} \\
& \langle \exists\, i, j : 2 \leqslant i < j : (\mathit{succ}(s_{i-1}) \rightarrow s_j)\ \wedge\ (s_j \rightarrow s_i) \rangle \\
\Rightarrow\ & \{\ \rightarrow \text{ is transitive }\} \\
& \langle \exists\, i : i \geqslant 2 : \mathit{succ}(s_{i-1}) \rightarrow s_i \rangle \\
\equiv\ & \{\ \text{definition of an edge }\} \\
& \langle \exists\, i : i \geqslant 2 : (s_{i-1}, s_i) \notin \mathsf{E}(G) \rangle \\
\Rightarrow\ & \{\ s \text{ is a path implying } \langle \forall\, i : i \geqslant 2 : (s_{i-1}, s_i) \in \mathsf{E}(G) \rangle\ \} \\
& \text{a contradiction}
\end{aligned}
$$

This establishes that s satisfies the agreement property. $\square$

The sufficient condition for a disjunctive predicate to be controllable in a computation can now be given as follows.

Theorem 4.15 (sufficient condition) *Given a disjunctive predicate b and a computation $\langle E, \rightarrow \rangle$, if there exists a permissible path in the corresponding true event graph G then b is controllable in $\langle E, \rightarrow \rangle$.*

Proof: Assume that G contains a permissible path. Clearly, each permissible path satisfies the boundary condition, the continuity property and the weak safety property. From Lemma 4.14, the shortest path among all permissible paths—not necessarily unique—also satisfies the agreement property. Thus the shortest permissible path in G constitutes an admissible sequence with respect to b and $\langle E, \rightarrow \rangle$. Using Theorem 4.13, b is controllable in $\langle E, \rightarrow \rangle$. $\square$

We next prove that the existence of a permissible path in the true event graph is also a necessary condition for a disjunctive predicate to be controllable in a computation.

Theorem 4.16 (necessary condition) *If a disjunctive predicate b is controllable in a computation $\langle E, \rightarrow \rangle$ then there exists a permissible path in the corresponding true event graph G.*

Proof: Assume that b is controllable in $\langle E, \rightarrow \rangle$. We inductively construct a path in the graph G that is permissible. Since b is controllable in $\langle E, \rightarrow \rangle$, there exists a total order $\rightsquigarrow$ that extends the partial order $\rightarrow$ such that b is invariant in $\langle E, \rightsquigarrow \rangle$. The initial consistent cut of the computation $\langle E, \rightsquigarrow \rangle$, given by $\perp$, satisfies b. Thus there exists a true initial event. We call it s_1. Starting from s_1, we construct a path s by adding events to the path constructed as yet until we reach a final event.

Let s_i denote the last event added to the path so far. If s_i is a final event then the path we have assembled so far is permissible. The more interesting case is when s_i is not a final event. Consider the *least* consistent cut of $\langle E, \rightsquigarrow \rangle$ that contains $succ(s_i)$, say C_i. Note that C_i is well-defined because the set of consistent cuts of a computation that contain a given event forms a lattice [JZ88, Mat89]. Since b is invariant in $\langle E, \rightsquigarrow \rangle$, C_i satisfies b. Thus the frontier of C_i contains a true event. We call it s_{i+1}. We still have to show that there is an edge from s_i to s_{i+1} in the graph G, that is, $succ(s_i) \not\rightarrow s_{i+1}$. By definition of C_i, for each $e \in C_i$, $e \underline{\rightsquigarrow} succ(s_i)$. Since $s_{i+1} \in C_i$, $s_{i+1} \underline{\rightsquigarrow} succ(s_i)$. Since $\rightsquigarrow$ is an irreflexive partial order, $succ(s_i) \not\rightsquigarrow s_{i+1}$. Thus $succ(s_i) \not\rightarrow s_{i+1}$ because $\rightarrow \subseteq \rightsquigarrow$.

Finally, we prove that a final event is eventually added to the path. Assume that $s_{i+1} \notin \top$. Since $s_{i+1} \in \mathit{frontier}(C)$, $succ(s_{i+1}) \notin C_i$. By definition of C_i, $succ(s_{i+1}) \not\underline{\rightsquigarrow} succ(s_i)$. Since $\rightsquigarrow$ is a total order, $succ(s_i) \rightsquigarrow succ(s_{i+1})$. This implies that $C_i \subsetneq C_{i+1}$, that is, s_{i+1} is different from every event already in the path. Thus no event is added to the path being built more than once, thereby establishing that a final event is eventually added to the path. $\square$

From Theorem 4.15 and Theorem 4.16, it follows that,

Theorem 4.17 (necessary and sufficient condition) *A disjunctive predicate b is controllable in a computation $\langle E, \rightarrow \rangle$ if and only if there exists a permissible path in the corresponding true event graph G.*

The true event graph has $O(|E|)$ vertices and $O(|E|^2)$ edges. The shortest permissible path in the graph can be determined using breadth first search in $O(|E|^2)$ time. Thus the algorithm has the overall time-complexity of $O(|E|^2)$. To improve the time-complexity, we attempt to reduce the number of edges in the graph. To that end, the following observation proves to be helpful.

Observation 4.3 *If there is an edge from a true event e to a true event f then* *there is an edge from every true event that occurs after* e *on* $proc(e)$ *to every true event that occurs before* f *on* $proc(f)$*. Formally,*

$$(e, f) \in \mathsf{E}(G) \;\Rightarrow\; \langle \forall\, g, h \in \mathsf{V}(G) : (e \xrightarrow{P} g) \wedge (h \xrightarrow{P} f) : (g, h) \in \mathsf{E}(G) \rangle$$

It can be verified that, given a true event e and a process p, if we only put an edge from e to the *last* true event f on p such that $succ(e) \not\rightarrow f$, in case $succ(e)$ exists, then Theorem 4.17 still holds. In particular, it can be proved that existence of a permissible path of length l in the true event graph implies existence of a permissible path in the "reduced" true event graph (RTEG) of length at most l. The reduced true event graph has at most $O(n|E|)$ edges, thereby reducing the time-complexity to $O(n|E|)$.

To reduce the time-complexity further, we define the notion of *true-interval*—a maximal contiguous sequence of true event on a process. Rather than find a sequence of true event that satisfy certain properties, we can find a sequence of true-intervals satisfying "similar" properties. The details are left to the reader. This algorithm for computing a controlling synchronization for a disjunctive predicate—based on true-intervals—has the time-complexity of $O(n|T|+|E|)$, where T is the set of true-intervals of the computation, which is same as that of Tarafdar and Garg's algorithm [TG98b].

4.4.3 Finding a Minimum Controlling Synchronization

We modify our algorithm for computing a controlling synchronization for a disjunctive predicate to compute a *minimum controlling synchronization*, that is, a synchronization with least number of dependencies that are not subsumed by the happened-before relation. We take advantage of the fact that the predicate to be controlled is disjunctive. As a result, a sequence of true events satisfies a stronger property than the weak safety property: "a consistent cut that contains at least one event from the sequence in its frontier satisfies the predicate". In particular, the cut is not required to be legal. Therefore the following holds:

Observation 4.4 *Let s be an admissible sequence with respect to b and $\langle E, \rightarrow \rangle$. If b is a disjunctive predicate then the synchronization given by $\stackrel{S^{(2)}}{\rightarrow}$ defined in (4.4) in Section 4.4.1 is sufficient to control b in $\langle E, \rightarrow \rangle$.*

Although the synchronization dependencies given by $\stackrel{S^{(1)}}{\rightarrow}$ can be omitted, the sequence is still required to satisfy the agreement property. This is to ensure that the synchronization $\stackrel{S^{(2)}}{\rightarrow}$ does not interfere with the happened-before relation of the computation. To count the number of synchronization dependencies in $\stackrel{S^{(2)}}{\rightarrow}$ that are not covered by $\rightarrow$, we assign weight to each edge as follows:

$$w(e, f) \quad \triangleq \quad \begin{cases} (0, 1) & : \text{ if } f \underline{\rightarrow} succ(e) \\ (1, 1) & : \text{ otherwise} \end{cases}$$

Two weights are added by adding their respective components and are compared lexicographically. As before in the case of true event graph, the shortest permissible path in a weighted true event graph not only satisfies the boundary condition, the continuity property and the weak safety property but also satisfies the agreement property.

Lemma 4.18 *The shortest permissible path in a weighted true event graph, if it exists, satisfies the agreement property.*

Proof: Assume that the weighted true event graph does contain a permissible path. Consider the shortest permissible path $s = s_1 s_2 \cdots s_l$. Assume, on the contrary, that s does not satisfy the agreement property. Then there exist integers i and j, where $i < j$, such that $s_j \rightarrow s_i$. Since s is the shortest permissible path, $s_j \notin \perp$; if otherwise, the path $s_j s_{j+1} \cdots s_l$ is a shorter permissible path than s—a contradiction. Furthermore, $i \geqslant 2$; if otherwise, $s_i \in \perp$ which implies that $s_i \rightarrow s_j$, thereby creating a cycle in $\rightarrow$. Two possible cases arise depending on whether there is an edge from s_{i-1} to s_j.

Case 1: $(s_{i-1}, s_j) \notin \mathsf{E}(G)$

$$
\begin{array}{ll}
 & \{ \text{ definition of an edge } \} \\
 & (succ(s_{i-1}) \rightarrow s_j) \ \wedge \ (s_j \rightarrow s_i) \\
\Rightarrow & \{ \rightarrow \text{ is transitive } \} \\
 & succ(s_{i-1}) \rightarrow s_i \\
\equiv & \{ \text{ definition of an edge } \} \\
 & (s_{i-1}, s_i) \notin \mathsf{E}(G) \\
\Rightarrow & \{ \ s \text{ is a path implying } (s_{i-1}, s_i) \in \mathsf{E}(G) \ \} \\
 & \text{a contradiction}
\end{array}
$$

In the second case, two possible sub-cases arise depending on the weight of the edge from s_{i-1} to s_j. If $w(s_{i-1}, s_j) = (0,1)$ then the path $s_1 s_2 \cdots s_{i-1} s_j \cdots s_l$ is permissible and has lesser weight than s—a contradiction. The more interesting case is when $w(s_{i-1}, s_j) = (1,1)$. Then,

Case 2.2: $w(s_{i-1}, s_j) = (1,1)$

$$
\begin{array}{ll}
 & \{ \text{ definition of the weight function } \} \\
 & s_j \not\underline{\rightarrow} succ(s_{i-1}) \\
\Rightarrow & \{ \ s_j \rightarrow s_i \text{ implying } s_i \underline{\rightarrow} succ(s_{i-1}) \ \Rightarrow \ s_j \underline{\rightarrow} succ(s_{i-1}) \ \} \\
 & s_i \not\underline{\rightarrow} succ(s_{i-1}) \\
\equiv & \{ \ (s_{i-1}, s_i) \in \mathsf{E}(G) \text{ and definition of the weight function } \} \\
 & w(s_{i-1}, s_i) = (1,1)
\end{array}
$$

Thus the path $s_1 s_2 \cdots s_{i-1} s_j \cdots s_l$ is permissible and has lesser weight than s—a contradiction. This establishes that s satisfies the agreement property. $\square$

For a path s with weight $w(s)$, let $w_f(s)$ and $w_s(s)$ denote the first and second entries, respectively, of the tuple $w(s)$. The rank of a weighted true event graph G, denoted by $rank(G)$, is given by,

$$rank(G) \quad \triangleq \quad \begin{cases} \perp & : \text{if there is no permissible path in G} \\ w_f(s) & : s \text{ is the shortest permissible path in } G \end{cases}$$

Intuitively, the rank gives the cardinality of a minimum controlling synchronization. We show that rank behaves in a continuous fashion by proving that adding a single synchronization dependency to a computation cannot reduce the rank of its weighted true event graph substantially. Consider a computation $\langle E, \rightsquigarrow \rangle$ such that (1) $\rightsquigarrow$ extends $\rightarrow$, and (2) the two computations $\langle E, \rightarrow \rangle$ and $\langle E, \rightsquigarrow \rangle$ differ by at most one message. Formally,

$$\langle \exists\, e, f :: \rightsquigarrow = (\rightarrow \cup (e, f)^+) \rangle$$

Let H be the weighted true event graph corresponding to b and $\langle E, \rightsquigarrow \rangle$.

Lemma 4.19 (bounded reduction) *If b is controllable in $\langle E, \rightsquigarrow \rangle$ then $rank(G)$ is at most one more than $rank(H)$.*

Proof: Since $\langle E, \rightsquigarrow \rangle \models$ *controllable*: b, by virtue of Theorem 4.16, there exists a permissible path in H. Consider the shortest permissible path in H, say $s = s_1 s_2 \cdots s_l$. For convenience, let w^G and w^H be the weight functions for the graphs G and H, respectively. Since $\rightarrow \subseteq \rightsquigarrow$, $succ(e) \not\rightsquigarrow f$ implies $succ(e) \not\rightarrow f$. Thus each edge of H is also an edge of G which implies that s is a path in G. The following can be easily verified.

$$rank(G) \leqslant w_f^G(s) \tag{4.5}$$

$$rank(H) = w_f^H(s) \tag{4.6}$$

$$\langle \forall\, e, f : (e,f) \in \mathsf{E}(H) : w^G(e,f) = (0,1) \;\Rightarrow\; w^H(e,f) = (0,1)\rangle \tag{4.7}$$

We first prove that $w_f^G(s) - w_f^H(s) \leqslant 1$. Assume the contrary. Thus, from (4.7), there exist at least two distinct edges in the path s such that their weight in G is $(1,1)$ but in H is $(0,1)$. Let the edges be (s_i, s_{i+1}) and (s_j, s_{j+1}), where $i \neq j$. Equivalently,

$$s_{i+1} \not\underline{\rightarrow}\, succ(s_i) \;\text{ and }\; s_{j+1} \not\underline{\rightarrow}\, succ(s_j) \tag{4.8}$$

$$s_{i+1} \underline{\rightsquigarrow}\, succ(s_i) \;\text{ and }\; s_{j+1} \underline{\rightsquigarrow}\, succ(s_j) \tag{4.9}$$

Let the additional message in $\langle E, \rightsquigarrow\rangle$ be from e to f. From (4.8) and (4.9), we can deduce that there exists a path from s_{i+1} to $succ(s_i)$ in $\langle E, \rightsquigarrow\rangle$ that involves the message from e to f. Likewise, there exists a path from s_{j+1} to $succ(s_j)$ in $\langle E, \rightsquigarrow\rangle$ that involves the message from e to f. Then,

$$s_{i+1} \underline{\rightsquigarrow}\, e \;\text{ and }\; f \underline{\rightsquigarrow}\, succ(s_i) \tag{4.10}$$

$$s_{j+1} \underline{\rightsquigarrow}\, e \;\text{ and }\; f \underline{\rightsquigarrow}\, succ(s_j) \tag{4.11}$$

Without loss of generality, assume that $i < j$. Two possible cases arise depending on whether there is an edge from s_i to s_{j+1} in H. We have,

Case 1: $(s_i, s_{j+1}) \notin \mathsf{E}(H)$

$\{$ definition of an edge $\}$

$succ(s_i) \rightsquigarrow s_{j+1}$

$\Rightarrow$ $\{$ using (4.11) $\}$

$succ(s_i) \rightsquigarrow e$

$\Rightarrow$ $\{$ using (4.10) $\}$

$f \rightsquigarrow e$

$\Rightarrow$ $\{$ definition of $\rightsquigarrow$ implies $e \rightsquigarrow f$ $\}$

a contradiction

In the second case, when there is an edge from s_i to s_{j+1}, from (4.10) and (4.11), $s_{j+1} \rightsquigarrow succ(s_i)$. Thus $w^H(s_i, s_{j+1}) = (0,1)$ implying that the path $s_1 s_2 \cdots s_i s_{j+1} \cdots s_l$ is permissible in H and has smaller weight than s—a contradiction. Thus,

$$w_f^G(s) - w_f^H(s) \leqslant 1 \tag{4.12}$$

Finally,

$$\begin{array}{ll} & \{ \text{ using (4.5) } \} \\ & rank(G) \leqslant w_f^G(s) \\ \equiv & \{ \text{ using (4.12) } \} \\ & rank(G) \leqslant w_f^H(s) + 1 \\ \equiv & \{ \text{ using (4.6) } \} \\ & rank(G) \leqslant rank(H) + 1 \end{array}$$

This establishes the lemma. $\square$

Now, assume that $rank(G) \neq 0$. Let $\mathcal{RCH}$ denote the subset of true events that are reachable from some initial true event in the weighted true event graph G via edges with weight $(0,1)$ only. Since $rank(G) \neq 0$, $\mathcal{RCH}$ does not contain any final event; if otherwise, there is a path from an initial event to a final event via edges with weight $(0,1)$ only, thereby forcing $rank(G)$ to be zero. For each process p_i, we identify an interval of contiguous events on p_i that we denote by $\mathcal{I}_i$. The first event of $\mathcal{I}_i$, denoted by $\mathcal{I}_i.lo$, is given by the successor of the last event on p_i that belongs to $\mathcal{RCH}$. In case there is no such event, $\mathcal{I}_i.lo$ is set to $\perp_i$, the initial event on p_i. The last event of $\mathcal{I}_i$, denoted by $\mathcal{I}_i.hi$, is given by the earliest event on p_i that did not occur before $\mathcal{I}_i.lo$ such that its successor, if it exists, is a true event. Clearly, $\mathcal{I}_i$ is non-empty and all events in $\mathcal{I}_i$ are false events. For convenience,

$$\mathcal{I} \triangleq \bigcup_{1 \leqslant i \leqslant n} \mathcal{I}_i$$

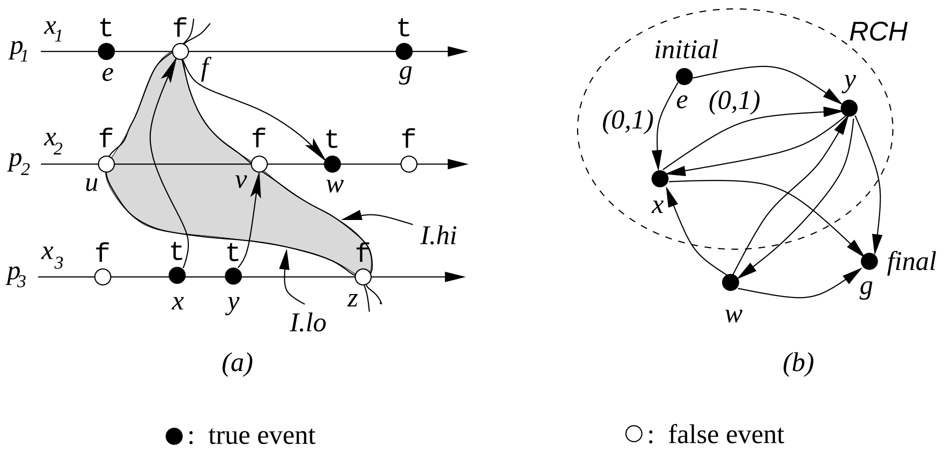


Figure 4.14: An example to illustrate $\mathcal{I}$.

$$\begin{aligned} \mathcal{I}.lo &\triangleq \{\, \mathcal{I}_i.lo \mid 1 \leqslant i \leqslant n \,\} \\ \mathcal{I}.hi &\triangleq \{\, \mathcal{I}_i.hi \mid 1 \leqslant i \leqslant n \,\} \\ succ(\mathcal{I}.hi) &\triangleq \{\, succ(e) \mid e \in \mathcal{I}.hi \;\text{ and }\; e \not\in \top \,\} \end{aligned}$$

Example 4.8 *Consider the computation portrayed in Figure 4.14(a) and the disjunctive predicate $x_1 \vee x_2 \vee x_3$. The corresponding weighted true event graph is depicted in Figure 4.14(b). The incoming edges to the initial event e and the outgoing edges from the final event g have been omitted for obvious reasons. All edges except the edges (e, x) and (x, y) have weight $(1, 1)$. For clarity, we have only labeled those edges that have weight $(0, 1)$ because they are fewer in number. Thus the set $\mathcal{RCH}$ is given by $\{e, x, y\}$. Further, $\mathcal{I}_1.lo = succ(e) = f$, $\mathcal{I}_2.lo = \bot_2 = u$ and $\mathcal{I}_3.lo = succ(y) = z$. Also, $\mathcal{I}_1.hi = f$, $\mathcal{I}_2.hi = v$ and $\mathcal{I}_3.hi = \top_3 = z$. Finally, $succ(\mathcal{I}) = \{succ(f), succ(v)\} = \{g, w\}$. The shaded region in Figure 4.14(a) corresponds to the space spanned by the events of $\mathcal{I}$.*

Observe that if all events in the frontier of a consistent cut belong to $\mathcal{I}$ then the cut will not satisfy the given disjunctive predicate. We make two observations about the set $succ(\mathcal{I}.hi)$. First, all events in the set are true events. Second, no event in the set belongs to $\mathcal{RCH}$. The following lemma proves that the computation must contain a consistent cut that does not satisfy the disjunctive predicate.

Lemma 4.20 *If the rank of a weighted true event graph is not zero then there exists a consistent cut of the computation that does not satisfy the disjunctive predicate.*

Proof: Our approach is to add enough synchronization dependencies to the computation $\langle E, \rightarrow \rangle$, without creating any deadlock (or cycle), to obtain another computation, say $\langle E, \leadsto \rangle$, that satisfies the required property. Specifically, we show that the computation $\langle E, \leadsto \rangle$ contains a consistent cut whose frontier is completely contained in $\mathcal{I}$. Since all events in $\mathcal{I}$ are false events, we obtain the desired result. The required set of dependencies, denoted by $\xrightarrow{\mathcal{I}}$, is given by,

$$\xrightarrow{\mathcal{I}} \;\triangleq\; \{\, (e, f) \mid e \in \mathcal{I}.lo \;\text{ and }\; f \in succ(\mathcal{I}.hi) \,\}$$

We first prove that adding dependencies from $\xrightarrow{\mathcal{I}}$ to $\rightarrow$ does not create any cycle. Consider a path $e \xrightarrow{\mathcal{I}} f \underline{\rightarrow} g \xrightarrow{\mathcal{I}} h$ (events e, f, g and h need not all be distinct, that is, an event or a sequence of events may be repeated in the path). By definition of $\xrightarrow{\mathcal{I}}$, $f \in succ(\mathcal{I}.hi)$ and $g \in \mathcal{I}.lo$. Clearly, $f \notin \perp$. This implies that $g \notin \perp$; if otherwise, $g \rightarrow f$, thereby creating a cycle in $\rightarrow$. Thus $pred(g)$ exists. Furthermore, both f and $pred(g)$ are true events such that $pred(g) \in \mathcal{RCH}$ but $f \notin \mathcal{RCH}$. Note, however, that $f \underline{\rightarrow} succ(pred(g))(= g)$ implying that there is an edge from $pred(g)$ to f with weight $(0, 1)$. Thus f is reachable from an initial event via edges with weight $(0, 1)$ only because $pred(g) \in \mathcal{RCH}$ and $w(pred(g), f) = (0, 1)$. This implies that f belongs to $\mathcal{RCH}$—a contradiction. Thus there is no path in $\rightarrow \cup \xrightarrow{\mathcal{I}}$ of the form $e \xrightarrow{\mathcal{I}} f \underline{\rightarrow} g \xrightarrow{\mathcal{I}} h$, thereby ensuring that $\rightarrow \cup \xrightarrow{\mathcal{I}}$ is acyclic.

Now, $\leadsto \;=\; (\rightarrow \cup \xrightarrow{\mathcal{I}})^{+}$. Consider the *least* consistent cut of $\langle E, \leadsto \rangle$, say $C_{least} \cdot (\mathcal{I}.lo)$, that contains $\mathcal{I}.lo$. By definition of $C_{least} \cdot (\mathcal{I}.lo)$, we have,

$$\langle \forall\, e :: e \in C_{least} \cdot (\mathcal{I}.lo) \;\Rightarrow\; \langle \exists\, f : f \in \mathcal{I}.lo : e \underline{\leadsto} f \rangle \rangle \tag{4.13}$$

We prove that the frontier of $C_{least}.(\mathcal{I}.lo)$ lies wholly within $\mathcal{I}$. To that end, it suffices to show that $C_{least}.(\mathcal{I}.lo)$ does not contain any event from $succ(\mathcal{I}.hi)$. Assume the contrary. Then,

$$
\begin{array}{cl}
 & \langle \exists\, e : e \in succ(\mathcal{I}.hi) : e \in C_{least}.(\mathcal{I}.lo) \rangle \\
\Rightarrow & \{ \text{ using (4.13) } \} \\
 & \langle \exists\, e, f : (e \in succ(\mathcal{I}.hi)) \;\wedge\; (f \in \mathcal{I}.lo) : e \underline{\rightsquigarrow} f \rangle \\
\Rightarrow & \{ \text{ by definition of } \xrightarrow{\mathcal{I}},\ f \xrightarrow{\mathcal{I}} e \ \text{ and } \ \xrightarrow{\mathcal{I}} \subseteq \rightsquigarrow \} \\
 & \langle \exists\, e, f : (e \in succ(\mathcal{I}.hi)) \;\wedge\; (f \in \mathcal{I}.lo) : (e \underline{\rightsquigarrow} f) \;\wedge\; (f \rightsquigarrow e) \rangle \\
\Rightarrow & \{ \rightsquigarrow \text{ is an irreflexive partial order } \} \\
 & \text{a contradiction}
\end{array}
$$

This establishes the lemma. $\square$

The necessary and sufficient condition for the rank of a weighted true event graph to be zero can now be furnished easily.

Theorem 4.21 *The rank of a weighted true event graph is zero if and only if the disjunctive predicate is invariant in the computation. Formally,*

$$\langle E, \rightarrow \rangle \;\models\; invariant\!: b \;\iff\; rank(G) = 0$$

Proof: $(\Rightarrow)$ Follows from Lemma 4.20.

$(\Leftarrow)$ From Lemma 4.18, the shortest permissible path, say s—which exists because $rank(G) \neq \bot$—corresponds to an admissible sequence of events with respect to b and $\langle E, \rightarrow \rangle$. Since b is a disjunctive predicate, by Observation 4.4, $\xrightarrow{S^{(2)}}$ is sufficient to control b in $\langle E, \rightarrow \rangle$. Let $\xrightarrow{C} = (\rightarrow \,\cup\, \xrightarrow{S^{(2)}})^{+}$. By definition of controllability, b is invariant in $\langle E, \xrightarrow{C} \rangle$. Furthermore, by definition of the weight function, $\xrightarrow{S^{(2)}} \subseteq \rightarrow$ which implies that $\xrightarrow{C} = \rightarrow$. $\square$

We now present the main result of this section.

Theorem 4.22 (minimum controlling synchronization) *The shortest permissible path in a weighted true event graph, if it exists, corresponds to a minimum controlling synchronization for the disjunctive predicate in the computation.*

Proof: Assume that the weighted true event graph G does contain a permissible path. From Theorem 4.17, b is controllable in $\langle E, \rightarrow \rangle$. Let $\stackrel{\min}{\rightarrow}$ denote a minimum controlling synchronization for b in $\langle E, \rightarrow \rangle$. Further, let $\{G^{(k)}\}$ represent the sequence of weighted true event graphs generated by adding synchronization dependencies from $\stackrel{\min}{\rightarrow}$ one-by-one, where $G^{(0)} = G$. Note that b is invariant in the computation obtained by adding all synchronization dependencies from $\stackrel{\min}{\rightarrow}$. From the bounded reduction lemma,

$$rank(G^{(i)}) - rank(G^{(i+1)}) \leqslant 1, \quad 0 \leqslant i < | \stackrel{\min}{\rightarrow} |$$

Adding the above inequality for all values of i, we obtain,

$$\begin{array}{ll} & rank(G^{(0)}) - rank(G^{|\stackrel{\min}{\rightarrow}|}) \leqslant | \stackrel{\min}{\rightarrow} | \\ \equiv & \{ \text{ using Theorem 4.21 } \} \\ & rank(G) - 0 \leqslant | \stackrel{\min}{\rightarrow} | \\ \equiv & \{ \text{ simplifying } \} \\ & rank(G) \leqslant | \stackrel{\min}{\rightarrow} | \\ \equiv & \{ \stackrel{\min}{\rightarrow} \text{ corresponds to a minimum controlling synchronization } \} \\ & rank(G) = | \stackrel{\min}{\rightarrow} | \end{array}$$

This establishes the theorem. □

The algorithm to compute a minimum controlling synchronization has $O(|E|^2)$ time-complexity because the weighted true event graph has $O(|E|)$ vertices, $O(|E|^2)$ edges, and the shortest permissible path in the graph can be determined using Dijkstra's shortest path algorithm [CLR91] in $O(|E|^2)$ time.

Chapter 5

Slicing Distributed Computations

In this chapter, we discuss in detail our results pertaining to slicing distributed computations with respect to global predicates.

5.1 Overview

We first extend the model of distributed computation, described in Chapter 2, in Section 5.2. Specifically, we relax the restriction that events can only be partially ordered and allow cycles to be present in the computation. The reason is because whereas, in the traditional model, a computation specifies the "observable" order of execution of events, in the extended model, it captures the set of "possible" consistent cuts that are currently relevant for our purpose. The extended model enables us to model both computation and slice in a uniform and coherent fashion.

We formally define the notion of "slice" in Section 5.3. Informally, the slice of a computation with respect to a predicate is the "smallest" computation that contains all consistent cuts of the original computation that satisfy the predicate. In case the slice contains only those consistent cuts of the computation that satisfy the predicate, it is referred to as "lean".

A natural question to ask is: "Is such a smallest computation uniquely defined for every predicate?" To prove that it is indeed the case, we define a new class of predicates in Section 5.4 called "regular predicates". Informally, a predicate is regular if the set of consistent cuts that satisfy the predicate is closed under set union and set intersection. Some examples of regular predicates are conjunctive predicates such as "no process is in red state" and certain monotonic channel predicates such as "all channels are empty" and "all green messages have been acknowledged". The class of regular predicates is closed under conjunction. We prove in Section 5.5 that the slice for a predicate is lean if and only if the predicate is regular. For the general case, when the predicate may not be regular, we define a closure operator that returns the "strongest" regular predicate weaker than the given predicate. We show that such a predicate exists and is uniquely defined for every predicate. This in turn proves that the slice exists and is uniquely defined for every predicate.

In Section 5.7, we develop a polynomial-time algorithm for computing the slice for a regular predicate. The algorithm has an overall time-complexity of $O(n^2|E|)$, where n is the number of processes and E is the set of events. In case the regular predicate can be decomposed into a conjunction of clauses, where each clause itself is a regular predicate, however, depending on variables of only a small subset of processes, we given an optimized salgorithm for computing the slice. The optimized version may yield a speedup of as much as n for many regular predicates. We also provide *optimal* algorithms for special cases of regular predicates, namely conjunctive predicates and monotonic channel predicates of the form "$\bigwedge_{i,j}$ (at most k_{ij} messages in transit from process p_i to process p_j)" and "$\bigwedge_{i,j}$ (at least k_{ij} messages in transit from process p_i to process p_j)", which have the time-complexity of $O(|E|)$.

We demonstrate how slicing can be used to monitor a regular predicate under various modalities. Furthermore, we argue that many results pertaining to consistent global checkpoints [NX95, Wan97] can be derived as special cases of slicing.

We establish in Section 5.8 that it is intractable in general to compute the slice for an arbitrary predicate. Nevertheless, polynomial-time algorithms can be developed for certain special classes of predicates. In particular, we provide an efficient algorithm to compute the slice for a linear predicate and its dual—a post-linear predicate [CG98]. We next introduce the notion of "grafting" which is useful in composing two slices. Given two slices, grafting can be used to either compute the smallest slice that contains all consistent cuts common to both slices or compute the smallest slice that contains consistent cuts of both slices. As a corollary, the slice for a predicate in disjunctive normal form (DNF) can now be easily obtained. We demonstrate how grafting can be employed to compute the slice for a "co-regular predicate" (that is, complement of a regular predicate) in polynomial-time. We also use grafting to efficiently compute the slice for a "k-local predicate" (depends on at most k processes) for constant k [SS95]. Furthermore, grafting can also be applied to compute an "approximate" slice—in polynomial-time—for a predicate composed from linear predicates, post-linear predicates, co-regular predicates and k-local predicates for constant k using $\wedge$ and $\vee$ operators.

Finally, in Section 5.9, we discuss our experimental results in evaluating the effectiveness of slicing in reducing the search-space for detecting a predicate under *possibly* modality. Our results indicate that computation slicing can lead to an exponential improvement over existing techniques both in terms of time as well as space.

5.2 Extending the Model

In this section, we extend the model of distributed computation and related notions that we described in Chapter 2. In this chapter, we relax the restriction that the order on events must be a partial order. More precisely, we use directed graphs to model distributed computations as well as slices. Directed graphs allow us to handle both of them in a uniform and convenient manner.

Given a directed graph G, let $\mathsf{V}(G)$ and $\mathsf{E}(G)$ denote its set of vertices and edges, respectively. A subset of vertices of a directed graph forms a *consistent cut* if the subset contains a vertex only if it also contains all its incoming neighbours. Formally,

$$C \text{ is a consistent cut of } G \;\triangleq\; \langle \forall e, f \in \mathsf{V}(G) : (e, f) \in \mathsf{E}(G) : f \in C \;\Rightarrow\; e \in C \rangle$$

Observe that a consistent cut either contains all vertices in a cycle or none of them. This observation can be generalized to a strongly connected component. Traditionally, the notion of consistent cut (*down-set* or *order ideal*) is defined for partially ordered sets [DP90]. Here, we extend the notion to sets with arbitrary orders. Let $\mathcal{C}(G)$ denote the set of consistent cuts of a directed graph G. Observe that the empty set $\emptyset$ and the set of vertices $\mathsf{V}(G)$ trivially belong to $\mathcal{C}(G)$. We call them *trivial* consistent cuts. Let $\mathcal{P}(G)$ denote the set of paths in a directed graph G, that is, the set of pairs of vertices (u, v) such that there is a path from u to v in G. We assume that each vertex has a path to itself.

5.2.1 Directed Graphs: Path- and Cut-Equivalence

A directed graph G is *cut-equivalent* to a directed graph H, denoted by $G \overset{C}{\cong} H$, if they have the same set of consistent cuts. Formally,

$$G \overset{C}{\cong} H \;\triangleq\; \mathcal{C}(G) = \mathcal{C}(H)$$

Likewise, a directed graph G is *path-equivalent* to a directed graph H, denoted by $G \overset{P}{\cong} H$, if a path from vertex u to vertex v in G implies a path from vertex u to vertex v in H and vice versa. Formally,

$$G \overset{P}{\cong} H \;\triangleq\; \mathcal{P}(G) = \mathcal{P}(H)$$

The next lemma explores the relation between the two notions.

Lemma 5.1 *Let G and H be directed graphs with the same set of vertices. Then,*

$$\mathcal{P}(G) \subseteq \mathcal{P}(H) \;\equiv\; \mathcal{C}(G) \supseteq \mathcal{C}(H)$$

Evidently, Lemma 5.1 implies that two directed graphs are cut-equivalent if and only if they are path-equivalent. In other words, in order to determine whether two directed graphs are cut-equivalent, it is *necessary and sufficient* to ascertain that they are path-equivalent. This is significant because, whereas path-equivalence can be verified in polynomial-time ($|\mathcal{P}(G)| = O(|\mathsf{V}(G)|^2)$), cut-equivalence is computationally expensive to ascertain in general ($|\mathcal{C}(G)| = O(2^{|\mathsf{V}(G)|})$). In the rest of the chapter, we use $\cong$ to denote both $\overset{C}{\cong}$ and $\overset{P}{\cong}$.

5.2.2 Distributed Computations as Directed Graphs

We model a distributed computation $\langle E, \rightarrow \rangle$ as a directed graph with vertices as the set of events E and edges as $\rightarrow$. To limit our attention to only those consistent cuts that can actually occur during an execution, we assume that $\mathcal{P}(\langle E, \rightarrow \rangle)$ contains at least the Lamport's happened-before relation [Lam78].

We assume the presence of a fictitious final event on each process which occurs after all other events on the process. Recall that a final event on process p_i is denoted by $\top_i$ which now refers to the aforementioned fictitious event. We assume that all initial events belong to the same strongly connected component. Similarly, all final events belong to the same strongly connected component. This ensures that any non-trivial consistent cut will contain all initial events and none of the final events. As a result, every consistent cut of a computation in the traditional model is a non-trivial consistent cut of the corresponding computation in the extended model and vice versa. Only non-trivial consistent cuts are of real interest to us. As we will see later, the extended model allows us to capture empty slices in a very convenient fashion.

A distributed computation in the extended model can contain cycles. This is because whereas a computation in the traditional (happened-before) model captures the *observable* order of execution of events, a computation in the extended model captures the set of possible consistent cuts.

Although, given a computation $\langle E, \rightarrow \rangle$, the relation $\rightarrow$ may contain cycles, the order of events on a process, that in turn refers to the sequence in which the events on a process were executed in real-time, is still a total order. Thus the notion of predecessor and successor events of an event defined in Chapter 2 is well-defined and so are the notions that depend on it such as "frontier" and "passes through".

Recall that two events are said to be consistent if they are contained in the frontier of some consistent cut, otherwise they are inconsistent. More precisely, it can be verified that events e and f are consistent if and only if there is no path in the computation from $succ(e)$, if it exists, to f and from $succ(f)$, if it exists, to e. Note that, in the extended model, in contrast to the traditional model, an event can be inconsistent with itself.

As before, a predicate is evaluated with respect to the values of variables resulting after executing all events in the cut. We leave the predicate undefined for the trivial consistent cuts.

5.3 Problem Statement

Informally, a *computation slice* (or simply a *slice*) is a concise representation of all those consistent cuts of the computation that satisfy the predicate. Formally,

Definition 5.1 (slice) *The* slice *of a computation with respect to a predicate is the smallest directed graph—with the least number of consistent cuts—that contains all consistent cuts of the given computation for which the predicate evaluates to true.*

We will later show that the notion of *smallest directed graph* in the definition is well-defined for every predicate. The slice of computation $\langle E, \rightarrow \rangle$ with respect to a predicate b is denoted by $\langle E, \rightarrow \rangle_b$. Note that $\langle E, \rightarrow \rangle = \langle E, \rightarrow \rangle_{\mathsf{true}}$. In the rest of the paper, we use the terms "computation", "slice" and "directed graph" interchangeably.

Note that every slice derived from the computation $\langle E, \rightarrow \rangle$ will have the trivial consistent cuts ($\emptyset$ and E) among its set of consistent cuts. Thus a slice is *empty* if it has no non-trivial consistent cuts. In the rest of the paper, unless otherwise stated, a consistent cut refers to a non-trivial consistent cut. In general, a slice will contain consistent cuts that do not satisfy the predicate (besides trivial consistent cuts). In case a slice does not contain any such cut, it is called *lean*. Formally,

Definition 5.2 (lean slice) *The slice of a computation with respect to a predicate is* lean *if every consistent cut of the slice satisfies the predicate.*

An interesting question to ask is for what class of predicates is the slice always lean? To answer this question, we introduce the class of regular predicates.

5.4 Regular Predicates

A global predicate is called *regular* if the set of consistent cuts that satisfy the predicate is closed under set intersection and set union. Formally, given a regular predicate b and consistent cuts C_1 and C_2,

$$(C_1 \models b) \wedge (C_2 \models b) \Rightarrow (C_1 \cap C_2 \models b) \wedge (C_1 \cup C_2 \models b)$$

Remark 5.1 *More precisely, given a set of elements that forms a lattice under some partial order, a subset of elements forms a sublattice of the lattice if the subset is closed under the meet and join operators of the lattice. In our case, the meet and join operators are set intersection and set union, respectively.*

If the set of consistent cuts that satisfy a predicate is closed under set intersection then the predicate is said to be linear *[CG98]. Dually, if the set of consistent cuts that satisfy a predicate is closed under set union then the predicate is said to be* post-linear *[CG98]. The class of regular predicates is, therefore, given by the intersection of the class of linear predicates and the class of post-linear predicates.*

It can be verified that a local predicate is a regular predicate. Therefore the following predicates are regular.

- process p_i is in "red" state
- the leader has sent all "prepare to commit" messages

We now provide more examples of regular predicates. Consider a function $f(x, y)$ with two arguments such that it is monotonic in its first argument x but anti-monotonic in its second argument y. Some examples of the function f are: $x - y$, $3x - 5y$, x/y when $x, y > 0$, and $\log_y x$ when $x, y \geqslant 1$. We establish that the predicates of the form $f(x, y) < c$ and $f(x, y) \leqslant c$, where c is some constant, are regular when either both x and y are monotonically non-decreasing variables or both x and y are monotonically non-increasing variables.

Lemma 5.2 *Let x and y be monotonically non-decreasing variables. Then the predicates $f(x,y) < c$ and $f(x,y) \leqslant c$ are regular predicates.*

Proof: We show that the predicate $f(x,y) < c$ is regular. The proof for the other predicate is similar and has been omitted. For a consistent C, let $x(C)$ and $y(C)$ denote the values of variables x and y, respectively, immediately after all events in C are executed. Consider consistent cuts C_1 and C_2 that satisfy the predicate $f(x,y) < c$. Note that, by definition of $C_1 \cap C_2$, $y(C_1 \cap C_2)$ is either $y(C_1)$ or $y(C_2)$. Without loss of generality, assume that $y(C_1 \cap C_2) = y(C_1)$. Then,

$$
\begin{array}{cl}
 & f(x(C_1 \cap C_2), y(C_1 \cap C_2)) \\
= & \{ \text{ assumption } \} \\
 & f(x(C_1 \cap C_2), y(C_1)) \\
\leqslant & \left\{ \begin{array}{l} x \text{ is monotonically non-decreasing implies } x(C_1 \cap C_2) \leqslant x(C_1), \\ \text{and } f \text{ is monotonic in } x \end{array} \right\} \\
 & f(x(C_1), y(C_1)) \\
< & \{ \ C_1 \text{ satisfies the predicate } f(x,y) < c \ \} \\
 & c
\end{array}
$$

Thus $C_1 \cap C_2$ satisfies the predicate $f(x,y) < c$. Also, note that, by definition of $C_1 \cup C_2$, $x(C_1 \cup C_2)$ is either $x(C_1)$ or $x(C_2)$. Without loss of generality, assume that $x(C_1 \cup C_2) = x(C_1)$. Then,

$$
\begin{array}{cl}
 & f(x(C_1 \cup C_2), y(C_1 \cup C_2)) \\
= & \{ \text{ assumption } \} \\
 & f(x(C_1), y(C_1 \cup C_2)) \\
\leqslant & \left\{ \begin{array}{l} y \text{ is monotonically non-decreasing implies } y(C_1 \cup C_2) \geqslant y(C_1), \\ \text{and } f \text{ is anti-monotonic in } y \end{array} \right\} \\
 & f(x(C_1), y(C_1))
\end{array}
$$

$$
\begin{aligned}
< \quad & \{\ C_1 \text{ satisfies the predicate } f(x,y) < c\ \} \\
& c
\end{aligned}
$$

Thus $C_1 \cup C_2$ also satisfies the predicate $f(x, y) < c$. $\square$

We now establish that Lemma 5.2 holds even when both x and y are monotonically non-increasing variables.

Lemma 5.3 *Let x and y be monotonically non-increasing variables. Then the predicates $f(x,y) < c$ and $f(x,y) \leqslant c$ are regular predicates.*

Proof: We show that the predicate $f(x, y) < c$ is regular. The proof for the other predicate is similar and has been omitted. For a consistent C, let $x(C)$ and $y(C)$ denote the values of variables x and y, respectively, immediately after all events in C are executed. Consider consistent cuts C_1 and C_2 that satisfy the predicate $f(x,y) < c$. Note that, by definition of $C_1 \cap C_2$, $x(C_1 \cap C_2)$ is either $x(C_1)$ or $x(C_2)$. Without loss of generality, assume that $x(C_1 \cap C_2) = x(C_1)$. Then,

$$
\begin{aligned}
& f(x(C_1 \cap C_2), y(C_1 \cap C_2)) \\
= \quad & \{\ \text{assumption}\ \} \\
& f(x(C_1), y(C_1 \cap C_2)) \\
\leqslant \quad & \left\{ \begin{array}{l} y \text{ is monotonically non-increasing implies } y(C_1 \cap C_2) \geqslant y(C_1), \\ \text{and } f \text{ is anti-monotonic in } y \end{array} \right\} \\
& f(x(C_1), y(C_1)) \\
< \quad & \{\ C_1 \text{ satisfies the predicate } f(x,y) < c\ \} \\
& c
\end{aligned}
$$

Thus $C_1 \cap C_2$ satisfies the predicate $f(x, y) < c$. Also, note that, by definition of $C_1 \cup C_2$, $y(C_1 \cup C_2)$ is either $y(C_1)$ or $y(C_2)$. Without loss of generality, assume that $y(C_1 \cup C_2) = y(C_1)$. Then,

$$
\begin{array}{cl}
& f(x(C_1 \cup C_2), y(C_1 \cup C_2)) \\
= & \{ \text{ assumption } \} \\
& f(x(C_1 \cup C_2), y(C_1)) \\
\leqslant & \left\{ \begin{array}{l} x \text{ is monotonically non-decreasing implies } x(C_1 \cup C_2) \leqslant x(C_1), \\ \text{and } f \text{ is monotonic in } x \end{array} \right\} \\
& f(x(C_1), y(C_1)) \\
< & \{ \ C_1 \text{ satisfies the predicate } f(x,y) < c \ \} \\
& c
\end{array}
$$

Thus $C_1 \cup C_2$ also satisfies the predicate $f(x, y) < c$. □

Combining the above two lemmas, we obtain the following:

Lemma 5.4 *The predicates of the form $f(x, y) < c$ and $f(x, y) \leqslant c$, where c is some constant, are regular when either both x and y are monotonically non-decreasing variables or both x and y are monotonically non-increasing variables.*

As a corollary of Lemma 5.4, it can be proved that Lemma 5.4 still holds when $<$ and $\leqslant$ are replaced by $>$ and $\geqslant$, respectively.

Corollary 5.5 *Let x and y be monotonically non-decreasing variables. Then the predicates $f(x, y) > c$ and $f(x, y) \geqslant c$ are regular predicates.*

Proof: Define $g(y, x) = -f(x, y)$ and $d = -c$. Observe that the predicate $f(x, y) > c$ is equivalent to the predicate $g(y, x) < d$. Furthermore, the function g is monotonic in its first argument y and anti-monotonic in its second argument x. From Lemma 5.4, the predicate $g(y, x) < d$ is regular and hence the predicate $f(x, y) > c$ is also regular. Similarly, the predicate $f(x, y) \geqslant c$ is regular. □

Similarly, it follows that:

Corollary 5.6 *Let x and y be monotonically non-increasing variables. Then the predicates $f(x, y) > c$ and $f(x, y) \geqslant c$ are regular predicates.*

The following theorem combines all the above results.

Theorem 5.7 *Let f be a function with two arguments such that it is monotonic in its first argument and anti-monotonic in its second argument. Then the predicate of the form $f(x, y)$ `relop` c, where `relop` $\in \{<, \leqslant, >, \geqslant\}$ and c is some constant, is regular when either both x and y are monotonically non-decreasing variables or both x and y are monotonically non-increasing variables.*

Remark 5.2 *Let x_i and y_i be variables on process p_i, where $1 \leqslant i \leqslant n$. Consider $j \in [1 \ldots n]$, $I \subseteq \{1, 2, \ldots, n\}$ and some constant c.*

Let $\mathbf{x}(I) = \{ x_i \mid i \in I \}$ and let f be a function on the variables in $\mathbf{x}(I)$ and y_j such that it is monotonic in each $x_i \in \mathbf{x}(I)$ but anti-monotonic in y_j. If each $x_i \in \mathbf{x}(I)$ is a monotonically non-decreasing variable then it can be established that the predicates $f(\mathbf{x}(I), y_j) < c$ and $f(\mathbf{x}(I), y_j) \leqslant c$ are linear predicates. Similarly, if each $x_i \in \mathbf{x}(I)$ is a monotonically non-increasing variable then it can be proved that the predicates $f(\mathbf{x}(I), y_j) > c$ and $f(\mathbf{x}(I), y_j) \geqslant c$ are also linear predicates. None of the predicates mentioned above is regular in general.

Dually, let $\mathbf{y}(I) = \{ y_i \mid i \in I \}$ and let f be a function on x_j and the variables in $\mathbf{y}(I)$ such that it is monotonic in x_j but anti-monotonic in each $y_i \in \mathbf{y}(I)$. If each $y_i \in \mathbf{x}(I)$ is a monotonically non-decreasing variable then it can be established that the predicates $f(x_j, \mathbf{y}(I)) < c$ and $f(x_j, \mathbf{y}(I)) \leqslant c$ are post-linear predicates. Similarly, if each $y_i \in \mathbf{y}(I)$ is a monotonically non-increasing variable then it can be proved that the predicates $f(x_j, \mathbf{y}(I)) > c$ and $f(x_j, \mathbf{y}(I)) \geqslant c$ are also post-linear predicates. As before, none of the predicates mentioned above is regular in general.

Theorem 5.7 therefore corresponds to the case when I is a singleton set. Further, observe that Theorem 5.7 holds not only for scalar variables, but also for *vector variables. As in the case of scalars, in the case of vectors, the variable should either be monotonically non-decreasing, that is, the value of the variable for the successor event either stays the same or strictly increases, or monotonically non-increasing, that is, the value of the variable for the successor event either stays the same or strictly decreases, as the case may be.*

By substituting $f(x, y)$ with $x - y$, x with "the number of messages that process p_i has sent to process p_j so far" and y with "the number of messages sent by process p_i that process p_j has received so far", it can be verified that the following predicates are regular.

- no outstanding message in the channel from process p_i to process p_j
- the channel from process p_i to process p_j is non-empty
- at most k messages in transit from process p_i to process p_j
- at least k messages in transit from process p_i to process p_j

We next show that the conjunction of two regular predicates is also a regular predicate.

Theorem 5.8 *The class of regular predicates is closed under conjunction.*

Proof: We have to prove that if b_1 and b_2 are regular predicates then so is $b_1 \wedge b_2$. Consider consistent cuts C_1 and C_2 that satisfy $b_1 \wedge b_2$. By semantics of conjunction, both C_1 and C_2 satisfy b_1 as well as b_2. Since b_1 and b_2 are regular predicates, $C_1 \cap C_2$ satisfies b_1 and b_2. Again, by semantics of conjunction, $C_1 \cap C_2$ satisfies $b_1 \wedge b_2$. Likewise, $C_1 \cup C_2$ satisfies $b_1 \wedge b_2$. Thus $b_1 \wedge b_2$ is a regular predicate. $\Box$

The closure under conjunction implies that the following predicates are also regular.

- any conjunction of local predicates
- no process has the token and no channel has the token
- every "request" message has been "acknowledged" in the system

5.5 Establishing the Existence and Uniqueness of Slice

In this section, we show that the slice exists and is uniquely defined for all predicates. Our approach is to first prove that the slice not only exists for a regular predicate, but is also lean. Using this fact we next establish that the slice exists even for a predicate that is not regular.

5.5.1 Regular Predicates

It is well known in distributed systems that the set of all consistent cuts of a computation forms a lattice under the subset relation [JZ88, Mat89]. We ask the question does the lattice of consistent cuts satisfy any additional property? It turns out that the answer to this question is in affirmative. Specifically, we show that the set of consistent cuts of a directed graph not only forms a lattice but that the lattice is *distributive*. A lattice is said to be *distributive* if meet distributes over join [DP90]. Formally,

$$a \sqcap (b \sqcup c) \;\equiv\; (a \sqcap b) \sqcup (a \sqcap c)$$

where $\sqcap$ and $\sqcup$ denote the meet (infimum) and join (supremum) operators, respectively. (It can be proved that meet distributes over join if and only if join distributes over meet.)

Theorem 5.9 *Given a directed graph G, $\langle \mathcal{C}(G); \subseteq \rangle$ forms a distributive lattice.*

Proof: Let C_1 and C_2 be consistent cuts of G. We define their meet and join as follows:

$$C_1 \sqcap C_2 \triangleq C_1 \cap C_2$$
$$C_1 \sqcup C_2 \triangleq C_1 \cup C_2$$

It is sufficient to establish that $C_1 \cap C_2$ and $C_1 \cup C_2$ are consistent cuts of G which can be easily verified. $\square$

The above theorem is a generalization of the result in lattice theory that the set of down-sets of a partially ordered set forms a distributive lattice [DP90]. We further prove that the set of consistent cuts (of a directed graph) does not satisfy any additional structural property. To that end, we need the notion of *join-irreducible* element defined as follows.

Definition 5.3 (join-irreducible element [DP90]) *An element of a lattice is* join-irreducible *if (1) it is not the least element of the lattice, and (2) it cannot be expressed as join of two distinct elements, both different from itself. Formally, $a \in L$ is join-irreducible if*

$$\langle \exists\, x :: x < a \rangle \ \bigwedge\ \langle \forall\, x, y \in L : a = x \sqcup y : (a = x) \vee (a = y) \rangle$$

Pictorially, an element of a lattice is join-irreducible if and only if it has exactly one lower cover, that is, it has exactly one incoming edge in the corresponding Hasse diagram. The notion of *meet-irreducible element* can be similarly defined. It turns out that a distributive lattice is uniquely characterized by the set of its join-irreducible elements. In particular, every element of the lattice can be written as join of some subset of its join-irreducible elements and vice versa. This is formally captured by the next theorem.

Theorem 5.10 (Birkhoff's Representation Theorem for Finite Distributive Lattices [DP90]) *Let L be a finite distributive lattice and*

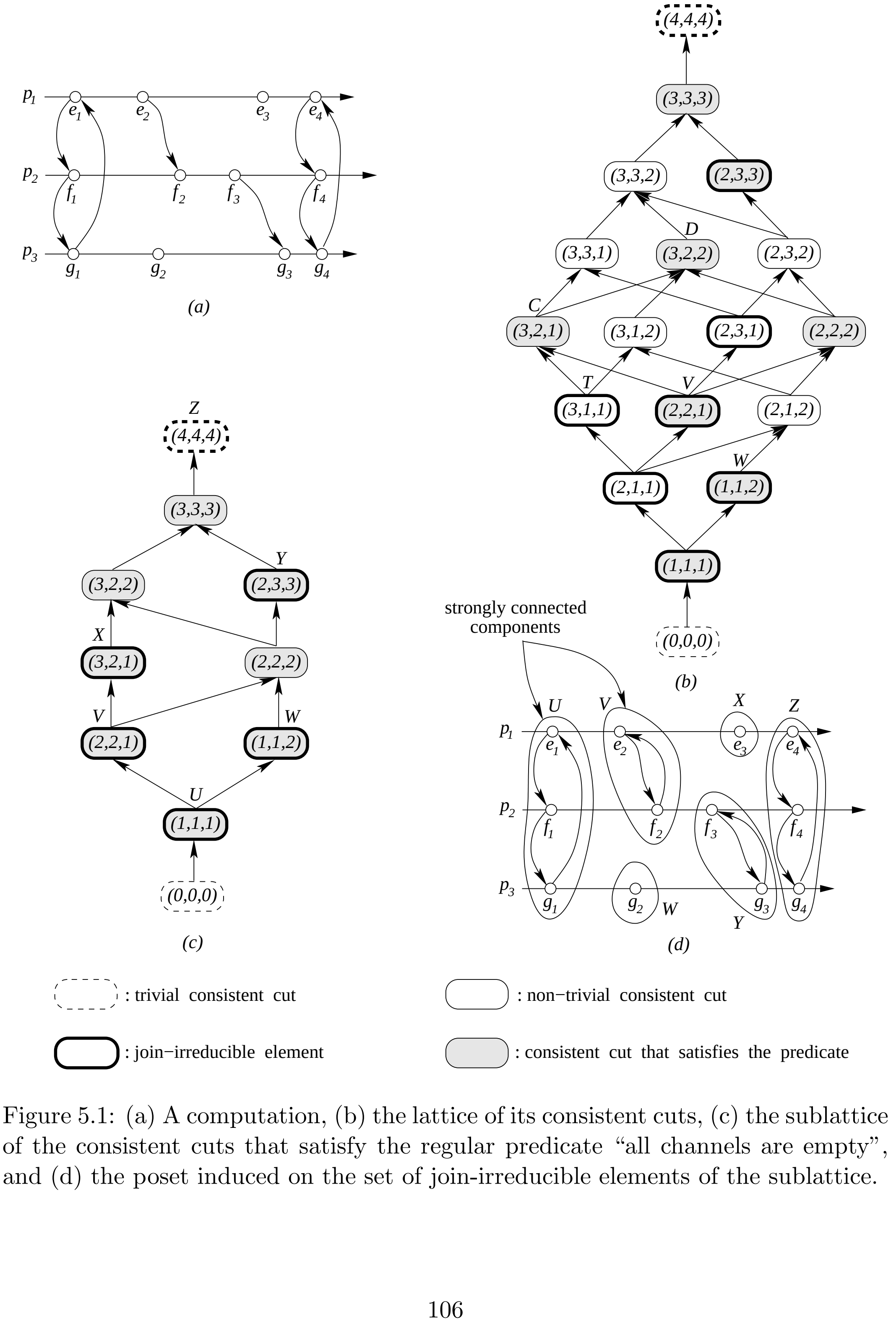


Figure 5.1: (a) A computation, (b) the lattice of its consistent cuts, (c) the sublattice of the consistent cuts that satisfy the regular predicate "all channels are empty", and (d) the poset induced on the set of join-irreducible elements of the sublattice.

$\mathcal{JI}(L)$ be the set of its join-irreducible elements. Then the map $f : L \longrightarrow \mathcal{C}(\mathcal{JI}(L))$ defined by

$$f(a) = \{\, x \in \mathcal{JI}(L) \mid x \leqslant a \,\}$$

is an isomorphism of L onto $\mathcal{C}(\mathcal{JI}(L))$. Dually, let P be a finite poset. Then the map $g : P \longrightarrow \mathcal{JI}(\mathcal{C}(P))$ defined by

$$g(a) = \{\, x \in P \mid x \leqslant a \,\}$$

is an isomorphism of P onto $\mathcal{JI}(\mathcal{C}(P))$.

Note that the above theorem can also be stated in terms of meet-irreducible elements.

Example 5.1 *Consider the computation shown in Figure 5.1(a). Figure 5.1(b) depicts the lattice of consistent cuts of the computation. In the figure, the label of a consistent cut indicates the number of events that have to be executed on each process to reach the cut. For example, the label of the consistent cut C is $(3, 2, 1)$ implying that to reach C, three events have to executed on process p_1, two on p_2 and one on p_3. Mathematically, $C = \{e_1, e_2, e_3, f_1, f_2, g_1\}$.*

In Figure 5.1(b), the consistent cuts of the computation corresponding to the join-irreducible elements of the lattice have been drawn in thick lines. There are exactly eight join-irreducible elements which is same as the number of strongly connected components of the computation. Note that the poset induced on the set of strongly connected components of the computation is isomorphic to the poset induced on the set of join-irreducible elements of the lattice. It can be verified that every consistent cut of the computation can be expressed as the join of some subset of these join-irreducible elements. For example, the consistent cut C can be written as the join of the consistent cuts T and V. Moreover, the join of every subset of these join-irreducible elements is a consistent cut of the computation. For instance, the join of the consistent cuts T, V and W is given by the consistent cut D.

In this chapter, we are concerned with only a subset of consistent cuts and not the entire set of consistent cuts. To that end, the notion of sublattice of a lattice comes in useful [DP90]. Given a lattice, a subset of its elements forms a *sublattice* if the subset is closed under the meet and join operators of the given lattice. In our case, the meet and join operators are set intersection and set union, respectively. Clearly, the set of consistent cuts satisfying a regular predicate forms a sublattice of the lattice of consistent cuts. Finally, we make an important observation regarding a sublattice which will help us prove the desired result.

Lemma 5.11 ([DP90]) *A sublattice of a distributive lattice is also a distributive lattice.*

Example 5.2 *In Figure 5.1(b), the consistent cuts for which the regular predicate "all channels are empty" evaluates to true have been shaded. Figure 5.1(c) depicts the poset induced on these consistent cuts. It can be verified that the poset forms a sublattice of the lattice in Figure 5.1(b). Moreover, the sublattice is, in fact, a distributive lattice.*

We now prove that the slice for a predicate is lean if and only if the predicate is regular.

Theorem 5.12 *The slice of a computation with respect to a predicate is lean if and only if the predicate is regular.*

Proof: (*if*) Assume that the predicate, say b, is regular. Thus the set of consistent cuts that satisfy the predicate, denoted by $\mathcal{C}_b$, forms a sublattice of the lattice of consistent cuts (of the computation). From Lemma 5.11, $\mathcal{C}_b$ is in fact a distributive lattice. Let $\mathcal{JI}(\mathcal{C}_b)$ denote the set of join-irreducible elements of $\mathcal{C}_b$. From Birkhoff's Representation Theorem, $\mathcal{C}_b$ is isomorphic to $\mathcal{C}(\mathcal{JI}(\mathcal{C}_b))$. Thus the required slice is given by the poset induced on $\mathcal{JI}(\mathcal{C}_b)$ by $\subseteq$. Moreover, every consistent cut of the slice satisfies the predicate and therefore the slice is lean.

(*only if*) Assume that the slice of a computation with respect to a predicate is lean. From the proof of Theorem 5.9, the set of consistent cuts of the slice is closed under set union and set intersection. This in turn implies that the set of consistent cuts that satisfy the predicate is closed under set union and set intersection. Thus the predicate is regular. $\square$

Example 5.3 *The sublattice shown in Figure 5.1(c) has exactly six join-irreducible elements, namely U, V, W, X, Y and Z. These elements or consistent cuts have been drawn in thick lines. It can be ascertained that every consistent cut in the sublattice can be written as the join of some subset of the consistent cuts in $\mathcal{J} = \{U, V, W, X, Y, Z\}$. In other words, every consistent cut of the computation that satisfies the regular predicate "all channels are empty" can be represented as the join of some subset of the elements in $\mathcal{J}$. Moreover, the join of every subset of elements in $\mathcal{J}$ yields a consistent cut contained in the sublattice and hence a cut that satisfies the regular predicate. The poset induced on the elements of $\mathcal{J}$ by the relation $\subseteq$ is shown in Figure 5.1(d). This poset corresponds to the slice of the computation shown in Figure 5.1(a) with respect to the regular predicate "all channels are empty".*

5.5.2 General Predicates

To prove that the slice exists even for a predicate that is not a regular predicate, we define a closure operator, denoted by reg, which, given a computation, converts an arbitrary predicate into a regular predicate satisfying certain properties. Given a computation $\langle E, \rightarrow \rangle$, let $\mathcal{R}(E)$ denote the set of predicates that are regular with respect to the computation ($\rightarrow$ is implicit).

Definition 5.4 (reg) *Given a predicate b, we define $reg\,(b)$ as the predicate that satisfies the following conditions:*

1. *it is regular, that is, $reg\,(b) \in \mathcal{R}(E)$,*

2. *it is weaker than b, that is, $b \Rightarrow reg\,(b)$, and*

3. *it is stronger than any other predicate that satisfies (1) and (2), that is, $\langle \forall\, b' : b' \in \mathcal{R}(E) : (b \Rightarrow b') \Rightarrow (reg\,(b) \Rightarrow b')\rangle$.*

Informally, $reg\,(b)$ is the *strongest regular predicate weaker* than b. In general, $reg\,(b)$ not only depends on the predicate b, but also on the computation under consideration. We assume the dependence on computation to be implicit and make it explicit only when necessary. The next theorem establishes that $reg\,(b)$ exists for every predicate b. Observe that the slice for b is given by the slice for $reg\,(b)$. Thus slice exists and is uniquely defined for all predicates.

Theorem 5.13 *Given a predicate b, $reg\,(b)$ exists and is uniquely defined.*

Proof: Let $\mathcal{R}_b(E)$ be the set of regular predicates in $\mathcal{R}(E)$ weaker than b. Observe that $\mathcal{R}_b(E)$ is non-empty because `true` is a regular predicate weaker than b and therefore contained in $\mathcal{R}_b(E)$. We set $reg\,(b)$ to the conjunction of all predicates in $\mathcal{R}_b(E)$. Formally,

$$reg\,(b) \;\triangleq\; \bigwedge_{q \,\in\, \mathcal{R}_b(E)} q$$

It remains to be shown that $reg\,(b)$ as defined satisfies the three required conditions. Now, condition (1) holds because the class of regular predicates is closed under conjunction. Condition (2) holds because every predicate in $\mathcal{R}_b(E)$ is weaker than b and hence their conjunction is weaker than b. Finally, let b' be a predicate that satisfies conditions (1) and (2). Note that $b' \in \mathcal{R}_b(E)$. Since conjunction of

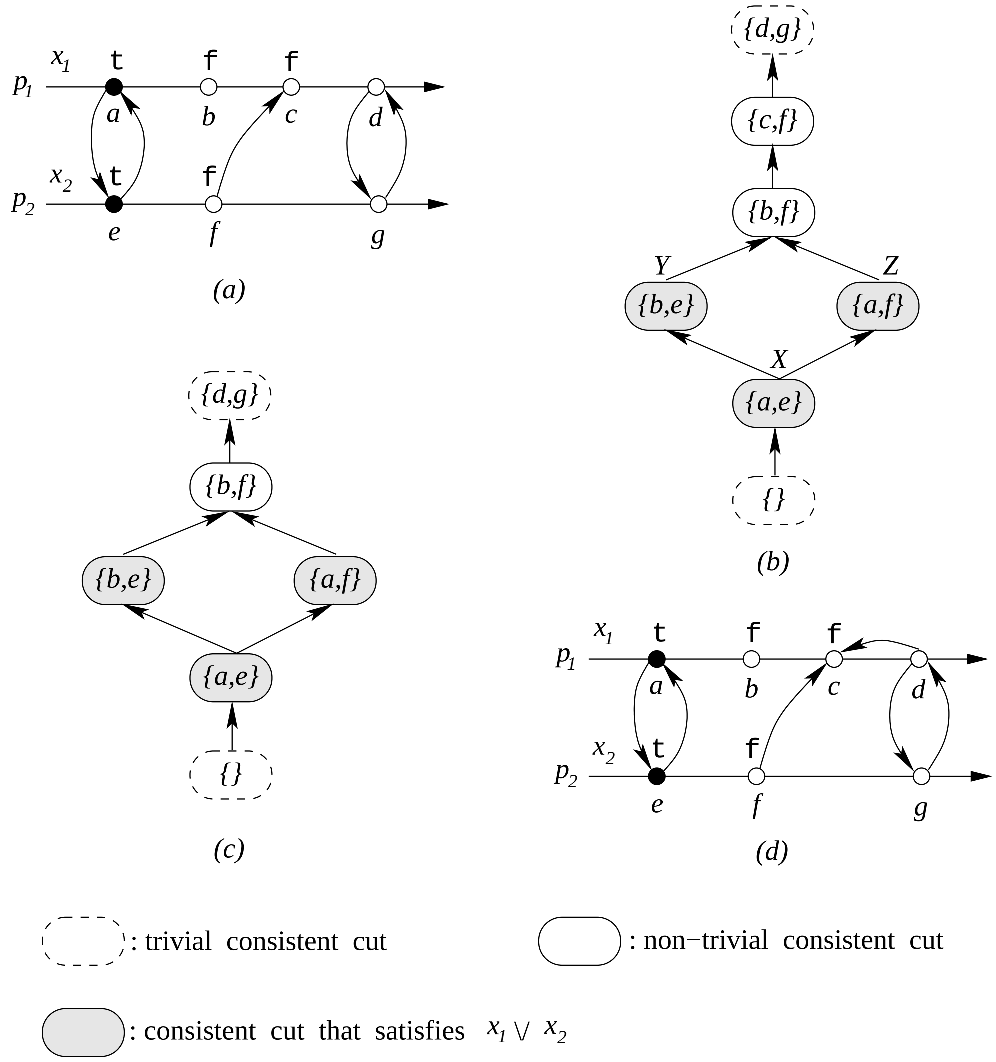


Figure 5.2: (a) A computation, (b) the lattice of its consistent cuts, (c) the sublattice of its consistent cuts that satisfy $reg\,(x_1 \vee x_2)$, and (d) its slice with respect to $reg\,(x_1 \vee x_2)$ (and therefore also with respect to $x_1 \vee x_2$).

predicates is stronger than any of its conjunct, $reg\,(b)$ is stronger than b'. Thus $reg\,(b)$ satisfies condition (3). □

Thus, given a computation $\langle E, \rightarrow \rangle$ and a predicate b, the slice of $\langle E, \rightarrow \rangle$ with respect to b can be obtained by first applying reg operator to b to get $reg\,(b)$ and then computing the slice of $\langle E, \rightarrow \rangle$ with respect to $reg\,(b)$.

Example 5.4 *Consider the computation shown in Figure 5.2(a). The lattice of its consistent cuts is depicted in Figure 5.2(b). Each consistent cut is labeled with its frontier. The consistent cuts for which the predicate $x_1 \vee x_2$ evaluates to true have been shaded in the figure. Clearly, the set of consistent cuts that satisfy $x_1 \vee x_2$ does not form a sublattice. The smallest sublattice that contains the subset is shown in Figure 5.2(c); the sublattice corresponds to the predicate $reg\,(x_1 \vee x_2)$. The slice for the regular predicate $reg\,(x_1 \vee x_2)$ and hence for the predicate $x_1 \vee x_2$ is portrayed in Figure 5.2(d).*

Theorem 5.14 *reg is a closure operator. Formally,*

1. *$reg\,(b)$ is weaker than b, that is, $b \;\Rightarrow\; reg\,(b)$,*
2. *reg is monotonic, that is, $(b \Rightarrow b') \;\Rightarrow\; (reg\,(b) \Rightarrow reg\,(b'))$, and*
3. *reg is idempotent, that is, $reg\,(reg\,(b)) \;\equiv\; reg\,(b)$.*

Proof: (*$reg\,(b)$ is weaker than b*) Follows from the definition.

(*reg is monotonic*) Since $reg\,(b')$ is weaker than b', it is also weaker than b. That is, $reg\,(b')$ is a regular predicate weaker than b. By definition, $reg\,(b)$ is the strongest regular predicate weaker than b. Therefore $reg\,(b)$ is stronger than $reg\,(b')$ or, in other words, $reg\,(b) \Rightarrow reg\,(b')$.

(*reg is idempotent*) Follows from the fact that $reg\,(b)$ is a regular predicate and is weaker than $reg\,(b)$. □

From the above theorem it follows that [DP90, Theorem 2.21],

Corollary 5.15 *$\langle \mathcal{R}(E); \Rightarrow \rangle$ forms a lattice.*

The meet and join of two regular predicates b_1 and b_2 is given by

$$\begin{aligned} b_1 \sqcap b_2 &\triangleq b_1 \wedge b_2 \\ b_1 \sqcup b_2 &\triangleq reg\,(b_1 \vee b_2) \end{aligned}$$

The dual notion of $reg\,(b)$, the weakest regular predicate stronger than b, is also conceivable. However, such a predicate may not always be unique.

Example 5.5 *In the previous example, three consistent cuts satisfy the predicate $x_1 \vee x_2$, namely X, Y and Z, as shown in Figure 5.2(b). Two distinct subsets of the set $\mathcal{S} = \{X, Y, Z\}$, given by $\{X, Y\}$ and $\{X, Z\}$, form maximal sublattices of $\mathcal{S}$ implying that there is no weakest regular predicate that is stronger than $x_1 \vee x_2$.*

5.6 Representing a Slice

Any directed graph that is cut-equivalent to a slice constitutes a valid representation of the slice. However, for computational purposes, it is preferable to select those graphs to represent a slice that have fewer edges and can be constructed cheaply. In this section, we show that every slice can be represented by a directed graph with $O(|E|)$ vertices and $O(n|E|)$ edges.

Consider a regular predicate b and a computation $\langle E, \rightarrow \rangle$. Recall that $\mathcal{C}(\langle E, \rightarrow \rangle_b)$ denote the set of consistent cuts of $\langle E, \rightarrow \rangle_b$, or, in other words, the set of consistent cuts of $\langle E, \rightarrow \rangle$ that satisfy b. For reasons of clarity, we abbreviate $\mathcal{C}(\langle E, \rightarrow \rangle_b)$ by $\mathcal{C}_b(E)$. From Birkhoff's Representation Theorem, the poset induced on $\mathcal{JI}(\mathcal{C}_b(E))$ by the relation $\subseteq$ is cut-equivalent to the slice $\langle E, \rightarrow \rangle_b$. It can be proved that $|\mathcal{JI}(\mathcal{C}_b(E))|$ is upper-bounded by $|E|$. Therefore the directed graph corresponding to $\langle \mathcal{JI}(\mathcal{C}_b(E)); \subseteq \rangle$ may have $\Omega(|E|^2)$.

In order to reduce the number of edges, we exploit properties of join-irreducible elements. For an event e, let $J_b(e)$ denote the *least consistent cut* of

$\langle E, \rightarrow\rangle$ that satisfies b and contains e. In case no consistent cut containing e that also satisfies b exists or when $e \in \top$, $J_b(e)$ is set to E—one of the trivial consistent cuts. Here, we use E as a *sentinel* cut. We first show that $J_b(e)$ is uniquely defined. Let i_e be the predicate defined as follows:

$$C \models i_e \triangleq (e \in C)$$

It can be proved that i_e is a regular predicate. Next, consider the predicate b_e defined as the conjunction of b and i_e. Since the class of regular predicates is closed under conjunction, b_e is also a regular predicate. The consistent cut $J_b(e)$ can now be reinterpreted as the least consistent that satisfies b_e. Since b_e is regular, the notion of least consistent cut that satisfies b_e is uniquely defined, thereby implying that $J_b(e)$ is uniquely defined. For purposes of computing the slice only, we assume that both trivial consistent cuts satisfy the given regular predicate. That is, $\{\emptyset, E\} \subseteq \mathcal{C}_b(E)$. The next lemma establishes that $J_b(e)$ is a join-irreducible element of $\mathcal{C}_b(E)$.

Lemma 5.16 *$J_b(e)$ is a join-irreducible element of the distributive lattice $\langle\mathcal{C}_b(E); \subseteq\rangle$.*

Proof: Suppose $J_b(e)$ can be expressed as the join (in our case, set union) of two consistent cuts in $\mathcal{C}_b(E)$, say C and D. That is, $J_b(e) = C \cup D$, where both C and D satisfy b. Our obligation is to show that either $J_b(e) = C$ or $J_b(e) = D$. Since $J_b(e)$ contains e, either C or D must contain e. Without loss of generality, assume that e belongs to C. By definition of union, $C \subseteq J_b(e)$. Further, since C is a consistent cut containing e that satisfies b and $J_b(e)$ is the *least* such cut, $J_b(e) \subseteq C$. Thus $J_b(e) = C$. $\square$

It is possible that $J_b(e)$s are not all distinct. Let $\mathcal{J}_b(E)$ denote the set $\{J_b(e) \mid e \in E\}$. Does $\mathcal{J}_b(e)$ capture all join-irreducible elements of $\mathcal{C}_b(E)$? The following lemma provides the answer.

Lemma 5.17 *Every consistent cut in $\mathcal{C}_b(E)$ can be expressed as the join of some subset of consistent cuts in $\mathcal{J}_b(E)$.*

Proof: Consider a consistent cut C in $\mathcal{C}_b(E)$. Let $D(C)$ be the consistent cut defined as follows:

$$D(C) = \bigcup_{e \in C} J_b(e)$$

We prove that $D(C)$ is actually equal to C. Since, by definition, $e \in J_b(e)$, each event in C is also present in $D(C)$. Thus $C \subseteq D(C)$. To prove that $D(C) \subseteq C$, consider an event $e \in C$. Since C is a consistent cut containing e that satisfies b and $J_b(e)$ is the *least* such cut, $J_b(e) \subseteq C$. More precisely, for each event $e \in C$, $J_b(e) \subseteq C$. This implies that $D(C) \subseteq C$. $\square$

From the previous two lemmas, it follows that $\mathcal{J}_b(E) = \mathcal{JI}(\mathcal{C}_b(E))$. Combining it with Birkhoff's Representation Theorem, we can deduce that:

Theorem 5.18 *Given a computation $\langle E, \rightarrow \rangle$ and a regular predicate b, the poset $\langle \mathcal{J}_b(E); \subseteq \rangle$ is cut-equivalent to the slice $\langle E, \rightarrow \rangle_b$.*

Next, to reduce the number of edges, rather than constructing a directed graph with join-irreducible elements as vertices, we construct a directed graph with events as vertices. Theorem 5.18 implies that:

Observation 5.1 *The directed graph $\mathcal{G}_b(E)$ with the set of vertices as E and an edge from an event e to an event f if and only if $J_b(e) \subseteq J_b(f)$ is cut-equivalent to the slice $\langle E, \rightarrow \rangle_b$.*

Whereas the *poset representation* of a slice is better for presentation purposes, the *graph representation* is more suited for slicing algorithms. From the way the graph $\mathcal{G}_b(E)$ is constructed, clearly, two events e and f belong to the same strongly connected component of $\mathcal{G}_b(E)$ if and only if $J_b(e) = J_b(f)$. As a result, there is

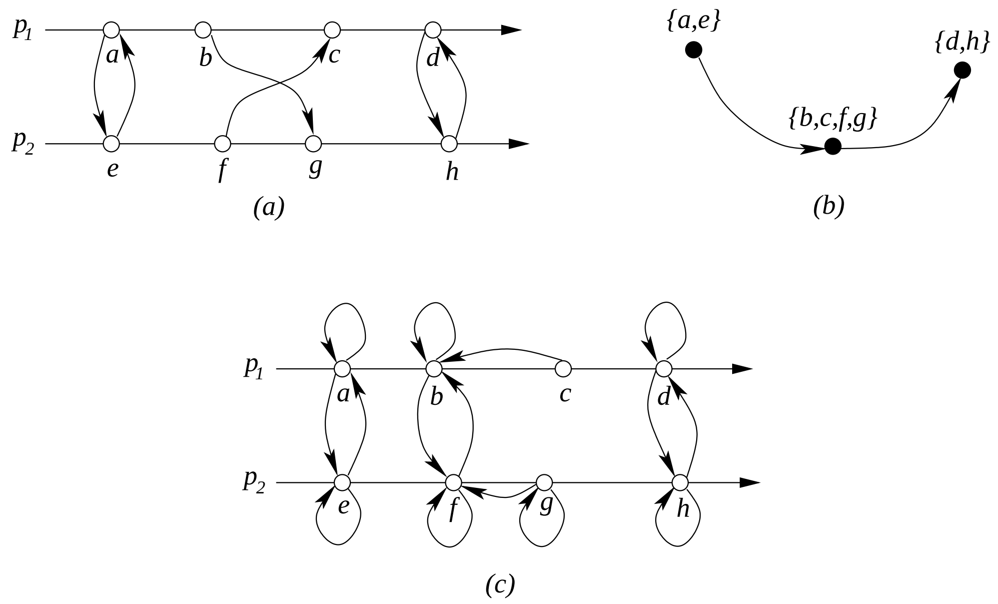


Figure 5.3: (a) A computation, (b) its slice with respect to the predicate "all channels are empty", and (c) the skeletal representation of the slice.

a one-to-one correspondence between the strongly connected components of $\mathcal{G}_b(E)$ and the join-irreducible elements of $\mathcal{C}_b(E)$.

Now, let $F_b(e)$ be a vector whose i^{th} entry denotes the earliest event f on process p_i such that $J_b(e) \subseteq J_b(f)$. Informally, $F_b(e)[i]$ is the earliest event on p_i that is reachable from e in the slice $\langle E, \rightarrow \rangle_b$. Using $F_b(e)$s, we construct a directed graph we call the *skeletal representation* of the slice and denote it by $\mathcal{S}_b(E)$. The graph $\mathcal{S}_b(E)$ has E as the set of vertices and the following edges:

1. for each event $e \notin \top$, there is an edge from e to $succ(e)$, and
2. for each event e and process p_i, there is an edge from e to $F_b(e)[i]$.

Example 5.6 *Consider the computation shown in Figure 5.3(a) and the predicate "all channels are empty". The slice with respect to the predicate is depicted in Figure 5.3(b). Here,* $J_b(c) = \{a, b, c, e, f, g\}$ *and* $J_b(g) = \{a, b, c, e, f, g\} = J_b(c)$.

Also, $F_b(c) = [b, f]$ and $F_b(g) = [b, f]$. The skeletal representation of the slice is shown in Figure 5.3(c).

To prove that $\mathcal{S}_b(E)$ faithfully captures the slice $\langle E, \rightarrow \rangle_b$, we prove the following two lemmas. The first lemma establishes that J_b is order-preserving.

Lemma 5.19 (J_b is order-preserving) *Given events e and f,*

$$e \rightarrow f \;\Rightarrow\; J_b(e) \subseteq J_b(f)$$

Proof: Consider $J_b(f)$. Since $e \rightarrow f$ and $f \in J_b(f)$, $e \in J_b(f)$. Thus $J_b(f)$ is a consistent cut that contains e and satisfies b. Since $J_b(e)$ is the *least* such cut, $J_b(e) \subseteq J_b(f)$. $\Box$

The second lemma shows that if $J_b(e) \subseteq J_b(f)$ then there is a path from event e to event f in $\mathcal{S}_b(E)$ and vice versa.

Lemma 5.20 *Given events e and f,*

$$J_b(e) \subseteq J_b(f) \;\equiv\; (e, f) \in \mathcal{P}(\mathcal{S}_b(E))$$

Proof: $(\Rightarrow)$ Assume that $J_b(e) \subseteq J_b(f)$. Let $proc(f) = p_i$ and $g = F_b(e)[i]$. Since, by definition, g is the earliest event on p_i such that $J_b(e) \subseteq J_b(g)$, $g \xrightarrow{P} f$. This implies that $(g, f) \in \mathcal{P}(\mathcal{S}_b(E))$. Further, by construction, $(e, g) \in \mathcal{P}(\mathcal{S}_b(E))$. Thus $(e, f) \in \mathcal{P}(\mathcal{S}_b(E))$.

$(\Leftarrow)$ It suffices to show that for each edge (u, v) in $\mathcal{S}_b(E)$, $J_b(u) \subseteq J_b(v)$. If $v = succ(u)$ then $J_b(u) \subseteq J_b(v)$ follows from Lemma 5.19. If $v = F_b(u)[i]$, where $p_i = proc(v)$, then $J_b(u) \subseteq J_b(v)$ follows from the definition of $F_b(u)$. $\Box$

Finally, from Observation 5.1 and Lemma 5.20, we can conclude that:

Theorem 5.21 *Given a computation $\langle E, \rightarrow \rangle$ and a regular predicate b, $\mathcal{S}_b(E)$ is cut-equivalent to $\langle E, \rightarrow \rangle_b$.*

It is easy to see that $\mathcal{S}_b(E)$ has $O(|E|)$ vertices and $O(n|E|)$ edges. In the next section we give efficient polynomial-time algorithms to compute $J_b(e)$ and $F_b(e)$ for each event e when b is a regular predicate.

5.7 Slicing for Regular Predicates

In this section, we discuss our results on slicing with respect to a regular predicate. They are discussed here separately from our results on slicing with respect to a general predicate because, as proved in Section 5.5.1, the slice for a regular predicate is lean and therefore furnishes more information than the slice for a general predicate. First, we present an efficient $O(n^2|E|)$ algorithm to compute the slice for a regular predicate. The algorithm is then optimized for the case when a regular predicate can be decomposed into a conjunction of clauses, where each clause itself is a regular predicate but depends on variables of only a small subset of processes. We also provide *optimal* algorithms for special cases of regular predicates such as conjunctive predicates and certain monotonic channel predicates. Next, we show how a regular predicate can be monitored under various modalities [CM91, GW91, TG99, MG00, SUL00], specifically *possibly*, *invariant* and *controllable*, using slicing. Finally, we demonstrate that results pertaining to consistent global checkpoints can be derived as special cases of slicing.

5.7.1 Computing the Slice for Regular Predicates

In this section, given a computation $\langle E, \rightarrow \rangle$ and a regular predicate b, we describe an efficient $O(n^2|E|)$ algorithm to compute the slice $\langle E, \rightarrow \rangle_b$. In particular, we construct $\mathcal{S}_b(E)$—the skeletal representation of $\langle E, \rightarrow \rangle_b$. To that end, it suffices to give an algorithm to compute $F_b(e)$ for each event e.

Our approach is to first compute $J_b(e)$ for each event e. To that end, consider the predicate b_e defined in Section 5.6. Since b_e is a regular predicate, it is also a

```
Algorithm Algo_5.1:

    Input: (1) a computation ⟨E, →⟩, (2) a regular predicate b, and
           (3) a process p_i

    Output: J_b(e) for each event e on p_i

    C := ⊥;

    for each event e on p_i do                      // visited in the order given by -P->
        done := false;
        if C = E then done := true;
        while not(done) do
            if there exist events f and g in frontier(C)
              such that succ(f) → g then            // C is not a consistent cut
                C := C ∪ {succ(f)};                 // advance beyond f
            else                                    // C is a consistent cut
                if C = E or C ⊨ b_e then done := true;
                else
                    f := forbidden_{b_e}(C);        // invoke the linearity property
                    C := C ∪ {succ(f)};             // advance beyond f
                endif;
            endif;
        endwhile;
        J_b(e) := C;
    endfor;
```

Figure 5.4: The algorithm $\mathsf{Algo}_{5.1}$ to compute $J_b(e)$ for each event e on process p_i.

linear predicate. (A predicate is said to be linear if, given two consistent cuts that satisfy the predicate, the consistent cut given by their set intersection also satisfies the predicate.) Chase and Garg [CG98] give an efficient algorithm to find the least consistent cut that satisfies a linear predicate. Their algorithm is based on the *linearity property* defined in Chapter 4. Please refer to the chapter for details.

Figure 5.4 describes the algorithm $\mathsf{Algo}_{5.1}$ to compute $J_b(e)$ for each event e on process p_i, using the linearity property, in a single scan of the computation from

Algorithm Algo $_{5.2}$:

<u>Input</u>: (1) a computation $\langle E, \rightarrow \rangle$, (2) $J_b(e)$ for each event e, and
(3) a process p_i

<u>Output</u>: $F_b(e)$ for each event e on p_i

for each process p_j do
    $f := \bot_j$;
    for each event e on p_i do   *// visited in the order given by* $\stackrel{P}{\rightarrow}$
        while $J_b(e) \not\subseteq J_b(f)$ do $f := succ(f)$; endwhile;
        $F_b(e)[j] := f$;
    endfor;
endfor;

Figure 5.5: The algorithm Algo $_{5.2}$ to compute $F_b(e)$ for each event e on process p_i.

left to right. This is possible because, from Lemma 5.19, once we have computed $J_b(e)$, we do not need to start all over again to determine $J_b(succ(e))$ but can rather continue on from $J_b(e)$ itself. The algorithm basically adds events one-by-one to the cut constructed so far until either all events are exhausted or the desired consistent cut is reached.

The time-complexity analysis of the algorithm Algo $_{5.1}$ is as follows. Each iteration of the while loop at line 5 has $O(n)$ time-complexity assuming that the time-complexity of invoking $forbidden_{b_e}$ at line 9 once is $O(n)$. Moreover, the while loop is executed at most $O(|E|)$ times because in each iteration either we succeed in finding the required consistent cut or we add a new event to C. Since there are at most $|E|$ events in the computation, the while loop cannot be executed more than $O(|E|)$ times. Thus the overall time-complexity of the algorithm Algo $_{5.1}$ is $O(n|E|)$ implying that $J_b(e)$ for each event e can be computed in $O(n^2|E|)$ time.

Finally, we give an algorithm to compute $F_b(e)$ for each event e provided $J_b(e)$

for each event e is given to us. We first establish a lemma similar to Lemma 5.19 for F_b. The lemma allows us to compute the j^{th} entry of $F_b(e)$ for each event e on process p_i in a single scan of the events on process p_j from left to right.

Lemma 5.22 *Given events e and f and a process p_i,*

$$e \rightarrow f \;\Rightarrow\; F_b(e)[i] \stackrel{P}{\rightarrow} F_b(f)[i]$$

Proof: Assume that $e \rightarrow f$. Let $g = F_b(e)[i]$ and $h = F_b(f)[i]$. Note that $proc(g) = proc(h) = p_i$. By definition of $F_b(f)$, $J_b(f) \subseteq J_b(h)$. Since, from Lemma 5.19, $J_b(e) \subseteq J_b(f)$, $J_b(e) \subseteq J_b(h)$. Again, by definition of $F_b(e)$, g is the earliest event on p_i such that $J_b(e) \subseteq J_b(g)$. Therefore $g \stackrel{P}{\rightarrow} h$. $\square$

Figure 5.5 depicts the algorithm $\mathsf{Algo}_{\,5.2}$ to compute $F_b(e)$ for each event e on process p_i. The algorithm is self-explanatory and its time-complexity analysis is as follows. Let E_j denote the set of events on process p_j. The outer for loop at line 1 is executed exactly n times. For j^{th} iteration of the outer for loop, the while loop at line 4 is executed at most $O(|E_i| + |E_j|)$ times. Each iteration of the while loop has $O(1)$ time-complexity because whether $J_b(e) \subseteq J_b(f)$ can be ascertained by performing only a single comparison. Thus the overall time-complexity of the algorithm $\mathsf{Algo}_{\,5.2}$ is $O(n|E_i| + |E|)$. Summing up over all processes, $F_b(e)$ for each event e can be determined in $O(n|E|)$ time. A summary of the overall algorithm is presented in Figure 5.6.

5.7.2 Optimizing for the Special Case: Computing the Slice for Decomposable Regular Predicates

In this section, we explore the possibility of a faster algorithm for the case when a regular predicate can be expressed as a conjunction of clauses such that each clause is again a regular predicate but spans a small fraction of processes. An example of

Algorithm $\mathsf{Algo}_{5.3}$:

Input: (1) a computation $\langle E, \rightarrow \rangle$, and (2) a regular predicate b

Output: the slice $\langle E, \rightarrow \rangle_b$

compute $J_b(e)$ for each event e using $\mathsf{Algo}_{5.1}$;
compute $F_b(e)$ for each event e using $\mathsf{Algo}_{5.2}$;
construct $\mathcal{S}_b(E)$ the skeletal representation of $\langle E, \rightarrow \rangle_b$;

Figure 5.6: The algorithm $\mathsf{Algo}_{5.3}$ to compute the slice for a regular predicate.

such a predicate is $\bigwedge_{1 \leqslant i,j \leqslant n}(|counter_i - counter_j| \leqslant \triangle_{ij})$, where each $counter_i$ is a monotonically non-decreasing variable on process p_i. In this example, each clause depends on variables of at most two processes. We describe the algorithm in two steps. In the first step, we give a fast algorithm to compute the slice for each clause. In the second step, we describe how to combine the slices for all clauses efficiently to obtain the slice for the desired regular predicate.

Step 1

Consider a computation $\langle E, \rightarrow \rangle$ and a regular predicate b that depends on variables of a subset Q of the set of processes P. *Without loss of generality, assume that $\rightarrow$ is a transitive relation.* We denote the projection of E on Q by $E(Q)$ and that of $\rightarrow$ on $Q \times Q$ by $\rightarrow(Q)$. Thus the projection of the computation $\langle E, \rightarrow \rangle$ on Q is given by $\langle E(Q), \rightarrow(Q) \rangle$.

We first show that the slice $\langle E, \rightarrow \rangle_b$ of the computation $\langle E, \rightarrow \rangle$ can be recovered exactly from the slice $\langle E(Q), \rightarrow(Q) \rangle_b$ of the projected computation $\langle E(Q), \rightarrow(Q) \rangle$. To that end, we extend the definition of $F_b(e)$ and define $F_b(e, Q)$ to be a vector whose i^{th} entry represents the earliest event on process p_i that is reachable from e in the slice $\langle E(Q), \rightarrow(Q) \rangle_b$. Thus $F_b(e) = F_b(e, P)$, $F(e, Q) =$

$F_{\mathsf{true}}(e, Q)$ and $F(e) = F_{\mathsf{true}}(e)$. We next define $K_b(e)$ as follows:

$$K_b(e)[i] = \begin{cases} F_b(e, Q)[i] & : \quad (e \in E(Q)) \wedge (p_i \in Q) \\ F(e)[i] & : \quad \text{otherwise} \end{cases}$$

We claim that it suffices to know $K_b(e)$ for each event e to be able to compute the slice $\langle E, \rightarrow \rangle_b$. Before we establish our claim, we define some notation. When events e and f occur on the same process and e occurred before f in real-time, then we write $e \overset{P}{\rightarrow} f$, and let $\overset{P}{\underline{\rightarrow}}$ be the reflexive closure of $\overset{P}{\rightarrow}$. We now build a graph $\mathcal{H}_b(E)$ that is similar to the skeletal representation $\mathcal{S}_b(E)$ of $\langle E, \rightarrow \rangle_b$ except that we use K_b instead of F_b in its construction. The next lemma proves that every path in $\mathcal{H}_b(E)$ is also a path in $\mathcal{S}_b(E)$.

Lemma 5.23 *For each event e and process p_i, $F_b(e)[i] \overset{P}{\underline{\rightarrow}} K_b(e)[i]$.*

Proof: Clearly, for each event e and process p_i, $F_b(e)[i] \overset{P}{\underline{\rightarrow}} F(e)[i]$. Thus we only need to prove that $F_b(e)[i] \overset{P}{\underline{\rightarrow}} F_b(e, Q)[i]$ when $e \in E(Q)$ and $p_i \in Q$.

Assume, on the contrary, that, for some event $e \in E(Q)$ and process $p_i \in Q$, $F_b(e, Q)[i] \overset{P}{\rightarrow} F_b(e)[i]$. For convenience, let $f = F_b(e, Q)[i]$ and $g = F_b(e)[i]$. Consider the *least* consistent cut C of the slice $\langle E, \rightarrow \rangle_b$ that contains f. Note that C does not contain e. This is because, by definition of $F_b(e)[i]$, g is the earliest event on p_i that is reachable from e in $\langle E, \rightarrow \rangle_b$. Since f occurs before g on p_i, f is not reachable from e in $\langle E, \rightarrow \rangle_b$ and therefore e is not contained in C. Let $C(Q)$ denote the projection of C on Q. Since C satisfies b and b depends only on variables of processes in Q, $C(Q)$ satisfies b. However, any consistent cut of $\langle E(Q), \rightarrow(Q) \rangle_b$ that contains f must contain e. This is because, by definition of $F_b(e, Q)[i]$, there is a path from e to f in $\langle E(Q), \rightarrow(Q) \rangle_b$. Thus $C(Q)$ is not a consistent cut of $\langle E(Q), \rightarrow(Q) \rangle_b$ which contradicts the fact that $\langle E(Q), \rightarrow(Q) \rangle_b$ contains all consistent cuts of $\langle E(Q), \rightarrow(Q) \rangle$ that satisfy b. This establishes the lemma. $\square$

We now prove the converse, that is, every path in $\mathcal{S}_b(E)$ is also a path in

Algorithm Algo $_{5.4}$:

<u>Input</u>: (1) a computation $\langle E, \rightarrow \rangle$, (2) a subset of processes Q, and
(3) a regular predicate b that depends only on variables of Q

<u>Output</u>: the slice $\langle E, \rightarrow \rangle_b$

1 compute $F(e)$ for each event e;
2 compute the projected computation $\langle E(Q), \rightarrow(Q) \rangle$;
3 compute the slice of the projected computation $\langle E(Q), \rightarrow(Q) \rangle_b$ using the algorithm Algo $_{5.3}$;
Also, compute $F_b(e, Q)$ for each event e;
4 compute $K_b(e)$ for each event e as follows:

$$K_b(e)[i] = \begin{cases} F_b(e, Q)[i] & : \quad (e \in E(Q)) \wedge (p_i \in Q) \\ F(e)[i] & : \quad \text{otherwise} \end{cases}$$

5 construct the directed graph $\mathcal{H}_b(E)$ with E as its set of vertices and edges as follows:

1. for each event $e \notin \top$, there is an edge from e to $succ(e)$, and
2. for each event e and process p_i, there is an edge from e to $K_b(e)[i]$.

Figure 5.7: The algorithm Algo $_{5.4}$ to compute the slice for a regular predicate that depends on variables of only a subset of processes.

$\mathcal{H}_b(E)$. To that end, by virtue of Lemma 5.1, it suffices to show that every consistent cut of $\mathcal{H}_b(E)$ is also a consistent cut of $\mathcal{S}_b(E)$ or, equivalently, every consistent cut of $\mathcal{H}_b(E)$ satisfies b.

Lemma 5.24 *Every consistent cut of $\mathcal{H}_b(E)$ satisfies b.*

Proof: Consider a consistent cut C of $\mathcal{H}_b(E)$. It is sufficient to prove that the projection of C on Q, denoted by $C(Q)$, is a consistent cut of $\langle E(Q), \rightarrow(Q) \rangle_b$. Assume, on the contrary, that $C(Q)$ is not a consistent cut of $\langle E(Q), \rightarrow(Q) \rangle_b$. Thus there exist events e and f such that there is a path from e to f in $\langle E(Q), \rightarrow(Q) \rangle_b$,

f is in $C(Q)$ but e is not. Let p_i denote the process on which f occurs. Clearly, $F_b(e, Q)[i] \overset{P}{\rightarrow} f$. This implies that there is a path from e to f in $\mathcal{H}_b(E)$ or, in other words, C is not a consistent cut of $\mathcal{H}_b(E)$—a contradiction. $\square$

Finally, the previous two lemmas can be combined to give the following theorem:

Theorem 5.25 *$\mathcal{H}_b(E)$ is cut-equivalent to $\mathcal{S}_b(E)$.*

Note that the graph $\mathcal{H}_b(E)$ may in fact be different from the skeletal representation $\mathcal{S}_b(E)$. However, the above theorem guarantees that the two will be path-equivalent. Figure 5.7 describes the algorithm $\mathsf{Algo}_{5.4}$ to compute the slice for a regular predicate that depends on variables of only a subset of processes in detail. We assume that the computation is given to us as n queues of events—one for each process. Further, the Fidge/Mattern's timestamp $ts(e)$ for each event e is also available to us. The algorithm $\mathsf{Algo}_{5.2}$ can be used to compute $F(e)$ for each event e in $O(n|E|)$ (b is true in this case). The projected computation can then be computed at line 2 in a straightforward fashion. The slice of the projected computation can be computed at line 3 in $O(|Q|^2|E(Q)|)$ time. The vector $K_b(e)$ for each event e can be determined at line 4 in $O(n|E|)$ time. Finally, the graph $\mathcal{H}_b(E)$ can be constructed at line 5 in $O(n|E|)$ time. Thus the overall time-complexity of the algorithm is $O(|Q|^2|E(Q)| + n|E|)$. If $|Q|$ is small, say at most $\sqrt{n}$, then the time-complexity of the algorithm is $O(n|E|)$—a factor of n faster than computing the slice directly using the algorithm $\mathsf{Algo}_{5.3}$.

A natural question to ask is: "Can this technique of taking a projection of a computation on a subset of processes, then computing the slice of the projection and finally mapping the slice back to the original set of processes be used for a non-regular predicate as well?" The answer is no in general as the following example suggests.

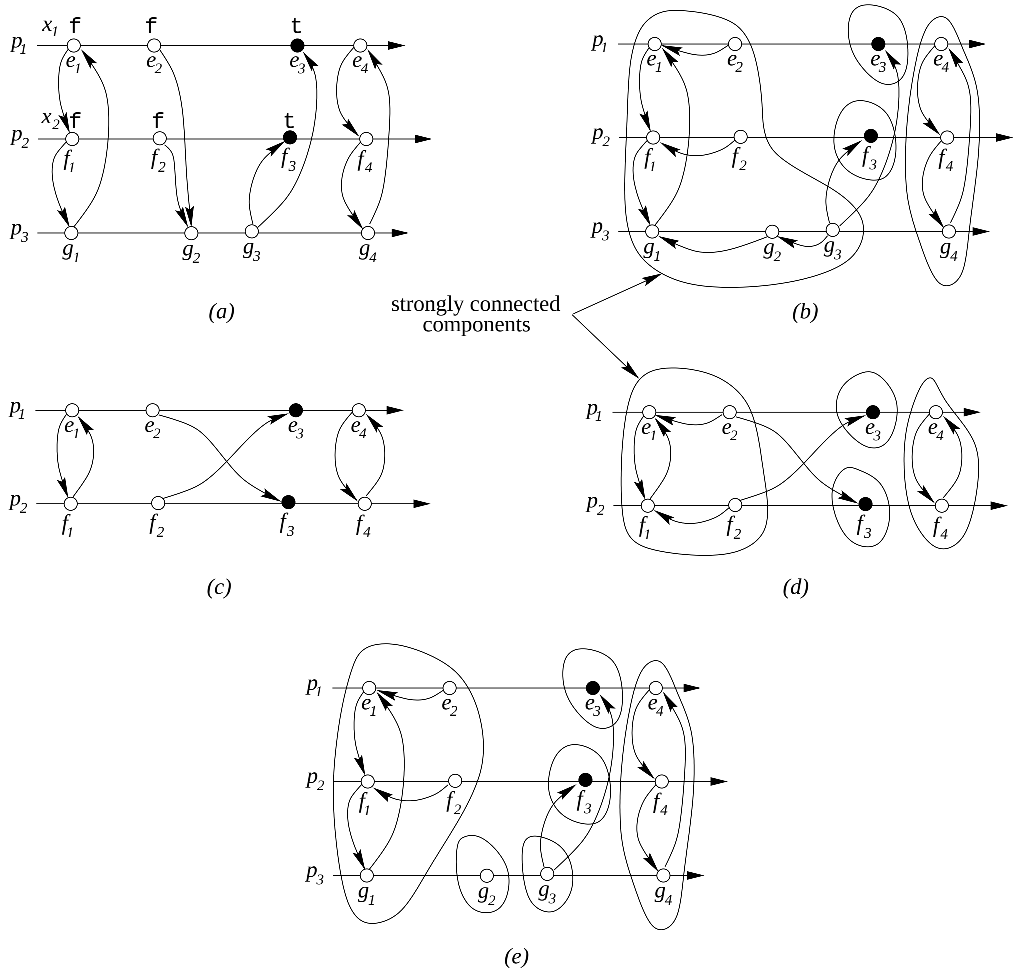


Figure 5.8: (a) A computation, (b) its slice with respect to the predicate $x_1 \vee x_2$, (c) its projection on processes p_1 and p_2, (d) the slice of the projected computation with respect to the predicate $x_1 \vee x_2$, and (e) the slice computed in (d) mapped to the original set of processes.

Example 5.7 *Consider the computation shown in Figure 5.8(a) involving three processes p_1, p_2 and p_3. Let x_1 and x_2 be boolean variables on processes p_1 and p_2, respectively. In the figure, the solid events, namely e_3 and f_3, satisfy the respective boolean variable. The slice of the computation for the (non-regular) predicate $x_1 \vee x_2$ is depicted in Figure 5.8(b). Figure 5.8(c) displays the projection of the computation* *on processes on which the predicate* $x_1 \vee x_2$ *depends, namely* p_1 *and* p_2. *The slice of the projected computation is shown in Figure 5.8(d) and its mapping back to the original set of processes is depicted in Figure 5.8(e). As it can be seen, the slice computed using the algorithm* $\mathsf{Algo}_{5.4}$ *(Figure 5.8(e)) is different from the actual slice (Figure 5.8(b)). For instance, events* g_2 *and* g_3 *belong to the same meta-event in the actual slice but not in the slice computed using the algorithm* $\mathsf{Algo}_{5.4}$. *The reason for this difference is as follows. Since the predicate* $x_1 \vee x_2$ *is non-regular, the slice of the projected computation shown in Figure 5.8(d) contains the consistent cut* $X = \{e_1, e_2, f_1, f_2\}$ *which does not satisfy* $x_1 \vee x_2$ *but has to be included anyway so as to complete the sublattice. Now, on mapping this slice back to the original set of processes, the resulting slice depicted in Figure 5.8(e) will contain* all *consistent cuts of the original computation whose projection on* $\{p_1, p_2\}$ *is* X. *There are three such consistent cuts, namely* $X \cup \{g_1\}$, $X \cup \{g_1, g_2\}$ *and* $X \cup \{g_1, g_2, g_3\}$. *However, only one of these consistent cuts, given by* $X \cup \{g_1, g_2, g_3\}$, *is required to complete the sublattice for the actual slice.*

It can be verified that the slice computed using the algorithm $\mathsf{Algo}_{5.4}$ for a non-regular predicate will, in general, be bigger than the actual slice. Thus the algorithm $\mathsf{Algo}_{5.4}$ gives a fast way to compute an approximate slice for a non-regular predicate (e.g., linear predicate).

Step 2

We use the above algorithm to devise a faster algorithm for computing the slice for a regular predicate b that can be expressed as conjunction of regular predicates $b^{(j)}$, $1 \leqslant j \leqslant m$, such that each $b^{(j)}$ is a function of variables on a subset of at most k processes Q_j. Let l denote the maximum number of subsets from the set $\{Q_j | 1 \leqslant j \leqslant m\}$, that contain a given process. For example, for the regular predicate $\bigwedge_{1 \leqslant i,j \leqslant n}(|counter_i - counter_j| \leqslant \triangle_{ij})$, where each $counter_i$ is a monotonically non-

```
for each event e ∈ E do
    K_b(e) := F(e);
endfor;

for each conjunct b^(j) do
    for each event e ∈ E(Q_j) do
        for each process p_i ∈ Q_j do
            K_b(e)[i] := min{ K_b(e)[i], F_{b^(j)}(e, Q_j)[i] };
        endfor;
    endfor;
endfor;
```

Figure 5.9: Computing $K_b(e)$ for each event e.

decreasing variable on process p_i, $k = 2$ and $l = n$.

To obtain the slice with respect to b, we first compute the slice $\langle E(Q_j), \rightarrow(Q_j)\rangle_{b^{(j)}}$ with respect to each conjunct $b^{(j)}$ using the algorithm in [GM01], thereby giving us the vector $F_{b^{(j)}}(e, Q_j)$ for each event $e \in Q_j$. We next compute the vector $K_b(e)$ for each event e as shown in Figure 5.9.

Intuitively, among all slices for clauses that contain some variable on process p_i, $K_b(e)[i]$ is the earliest event on p_i reachable from e in some slice. Formally, let Cl_i denote the set of clauses that depend on some variable on p_i. Then,

$$K_b(e)[i] = \min_{b^{(j)} \in Cl_i} \{F_{b^{(j)}}(e, Q_j)[i]\}$$

It can be easily verified that the graph $\mathcal{H}_b(E)$ then constructed using $K_b(e)$ for each event e (in a similar fashion as in Step 1) is actually cut-equivalent to the slice $\langle E, \rightarrow\rangle_b$. The proof is similar to the proof in Step 1 and has been omitted. The overall time-complexity of the algorithm is given by:

$$O(n|E|) + \sum_{1 \leqslant j \leqslant m} O(|Q_j|^2 |E(Q_j)|)$$

$$
\begin{aligned}
= & \quad \{ \text{ for each } Q_j,\ |Q_j| \leqslant k \ \} \\
& \quad O(n|E| + k^2 \sum_{1 \leqslant j \leqslant m} |E(Q_j)|) \\
= & \quad \{ \text{ simplifying } \} \\
& \quad O(n|E| + k^2 l|E|) = O((n + k^2 l)|E|)
\end{aligned}
$$

If k is constant and l is $O(n)$ then the overall time-complexity is $O(n|E|)$ which is a factor of n less than computing the slice directly using the algorithm $\mathsf{Algo}_{5.3}$.

5.7.3 Optimal Algorithms for Special Cases

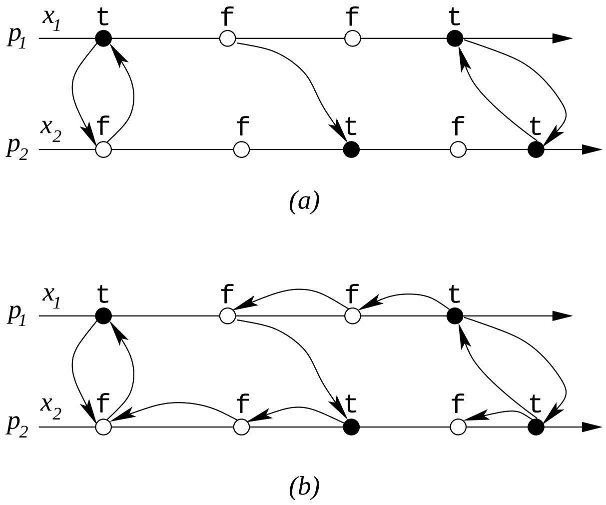
We now present optimal algorithms for computing the slice for special cases of regular predicates, namely conjunctive predicates and certain monotonic channel predicates. Our algorithms have $O(|E|)$ time-complexity.

Computing the Slice for Conjunctive Predicates

Consider a computation $\langle E, \rightarrow \rangle$ and a conjunctive predicate b. The first step is to partition events on each process into true events and false events. Having done that, we then construct a graph $\mathcal{H}_b(E)$ with vertices as the events in E and the following edges:

1. from an event, that is not a final event, to its successor,
2. from a send event to the corresponding receive event, and
3. from the successor of a false event to the false event.

For the purpose of building the graph, we assume that all final events are true events. Therefore every false event has a successor. The first two types of edges ensure that the Lamport's happened-before relation [Lam78] is contained in



●: true event ○: false event

Figure 5.10: (a) A computation, and (b) its slice with respect to the conjunctive predicate $x_1 \wedge x_2$.

$\mathcal{P}(\mathcal{H}_b(E))$. Consider the computation depicted in Figure 5.10(a) and the conjunctive predicate $x_1 \wedge x_2$. The corresponding graph constructed as described is shown in Figure 5.10(b). We now establish that the above-mentioned edges are sufficient to eliminate all those consistent cuts of the computation that do not satisfy the conjunctive predicate.

Lemma 5.26 *Every consistent cut of $\mathcal{H}_b(E)$ satisfies b.*

Proof: It is sufficient to prove that no consistent cut of $\mathcal{H}_b(E)$ contains a false event in its frontier. Consider a consistent cut C of $\mathcal{H}_b(E)$. Assume, on the contrary, that C contains a false event, say e, in its frontier. Since every false event has a successor, by construction, there is an edge from the successor of e, say f, to e. Therefore f also belongs to C. This contradicts the fact that e is the last event on its process to be contained in C. $\square$

We next show that the above constructed graph retains all consistent cuts of the computation that satisfy the conjunctive predicate.

Lemma 5.27 *Every consistent cut of $\langle E, \rightarrow \rangle$ that satisfies b is a consistent cut of $\mathcal{H}_b(E)$.*

Proof: Consider a consistent cut C of $\langle E, \rightarrow \rangle$ that satisfies b. Assume, on the contrary, that C is not a consistent cut of $\mathcal{H}_b(E)$. Thus there exist events e and f such that there is an edge from e to f in $\mathcal{H}_b(E)$, f belongs to C but e does not. Since C is a consistent cut of $\langle E, \rightarrow \rangle$, the edge from e to f could only of type (3). Equivalently, e and f occur on the same process, e is the successor of f, and f is a false event. Again, since f is contained in C but its successor e is not, f belongs to the frontier of C. However, C satisfies b and therefore cannot contain any false event in its frontier. $\square$

From the previous two lemmas, it follows that:

Theorem 5.28 *$\mathcal{H}_b(E)$ is cut-equivalent to $\langle E, \rightarrow \rangle_b$.*

It is easy to see that the graph $\mathcal{H}_b(E)$ has $O(|E|)$ vertices, $O(|E|)$ edges (at most three edges per event assuming that an event that is not local either sends at most one message or receives at most one message but not both) and can be built in $O(|E|)$ time. Thus the algorithm has $O(|E|)$ overall time-complexity. It also gives us an $O(|E|)$ algorithm to evaluate *possibly*: b when b is a conjunctive predicate.

Computing the Slice for Monotonic Channel Predicates

We present an optimal algorithm to compute the slice with respect to monotonic channel predicates such as:

- $\bigwedge_{i,j \in [1..n]}$ (at most k_{ij} messages in transit from process p_i to process p_j), and
- $\bigwedge_{i,j \in [1..n]}$ (at least k_{ij} messages in transit from process p_i to process p_j)

We only provide the slicing algorithm for the former predicate here. The slicing algorithm for the latter predicate is very similar and has been omitted. Let $snd\langle i,j\rangle(m)$ denote the send event on process p_i that corresponds to the send of m^{th} message to process p_j. Similarly, let $rcv\langle i,j\rangle(m)$ denote the receive event on process p_i that corresponds to the receive of m^{th} message from process p_j.

Consider a computation $\langle E, \rightarrow\rangle$ and a monotonic channel predicate b discussed in the previous paragraph. As in the case of conjunctive predicate, we construct a graph $\mathcal{H}_b(E)$ with vertices as the events in E and the following edges:

1. from an event, that is not a final event, to its successor,
2. from a send event to the corresponding receive event, and
3. from a receive event $rcv\langle j,i\rangle(m)$ to the send event $snd\langle i,j\rangle(m + k_{ij})$, if it exists.

As before, the first two types of edges ensure that the Lamport's happened-before relation [Lam78] is contained in $\mathcal{P}(\mathcal{H}_b(E))$. Consider the computation shown in Figure 5.11(a) and the monotonic channel predicate "at most one message in transit in any channel". Here, $k_{12} = k_{21} = 1$. The corresponding graph constructed as described is shown in Figure 5.11(b). We now establish that the above-mentioned edges are sufficient to eliminate all those consistent cuts of the computation that do not satisfy the channel predicate.

Lemma 5.29 *Every consistent cut of $\mathcal{H}_b(E)$ satisfies b.*

Proof: Consider a consistent cut C of $\mathcal{H}_b(E)$ and processes p_i and p_j. Let $snd\langle i,j\rangle(m)$ be the send event corresponding to the last message sent by p_i to

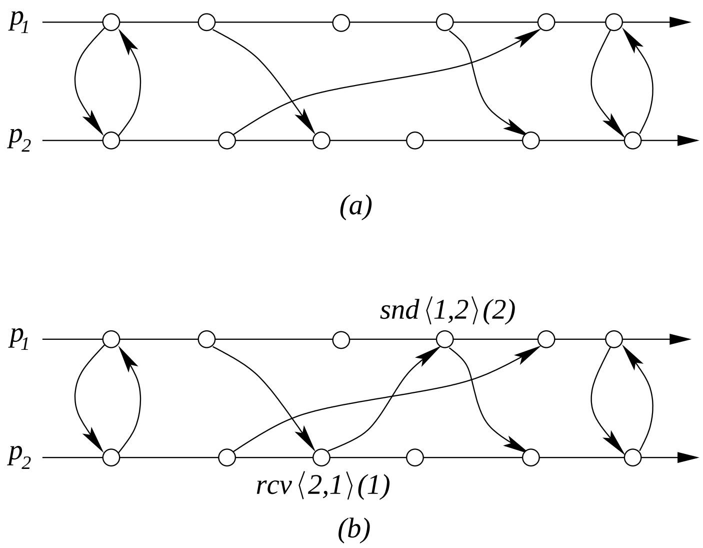


Figure 5.11: (a) A computation, and (b) its slice with respect to the monotonic channel predicate "at most one message in transit in any channel".

p_j such that $snd\langle i,j\rangle(m) \in C$. Since C is a consistent cut of $\mathcal{H}_b(E)$ and there is an edge from $rcv\langle j,i\rangle(m-k_{ij})$ to $snd\langle i,j\rangle(m)$, $rcv\langle j,i\rangle(m-k_{ij})$ also belongs to C. This implies that there are at most k_{ij} messages in transit from p_i to p_j. $\square$

We next show that the above constructed graph retains all consistent cuts of the computation that satisfy the channel predicate.

Lemma 5.30 *Every consistent cut of $\langle E, \rightarrow\rangle$ that satisfies b is a consistent cut of $\mathcal{H}_b(E)$.*

Proof: Consider a consistent cut C of $\langle E, \rightarrow\rangle$ that satisfies b. Assume, on the contrary, that C is not a consistent cut of $\mathcal{H}_b(E)$. Thus there exist events e and f such that there is a path from e to f in $\mathcal{H}_b(E)$, f belongs to C but e does not. Since C is a consistent cut of $\langle E, \rightarrow\rangle$, the edge from e to f could only of type (3). Let e be $rcv\langle j,i\rangle(m)$ and f be $snd\langle i,j\rangle(m+k_{ij})$. Since C satisfies b, $rcv\langle j,i\rangle(m)$

or, equivalently, e belongs to C—a contradiction. $\square$

From the previous two lemmas, it follows that:

Theorem 5.31 *$\mathcal{H}_b(E)$ is cut-equivalent to $\langle E, \rightarrow \rangle_b$.*

It is easy to see that the graph $\mathcal{H}_b(E)$ has $O(|E|)$ vertices, $O(|E|)$ edges (at most three edges per event assuming that an event that is not local either sends at most one message or receives at most one message but not both) and can be built in $O(|E|)$ time. Thus the algorithm has $O(|E|)$ overall time-complexity.

5.7.4 Applications of Slicing

In this section, we show that slicing can be used to solve other problems in distributed systems.

Monitoring Regular Predicates under Various Modalities

A predicate can be monitored under four modalities, namely *possibly*, *definitely*, *invariant* and *controllable* [CM91, GW91, TG99, MG00, SUL00]. A predicate is *possibly* true in a computation if there is a consistent cut of the computation that satisfies the predicate. On the other hand, a predicate *definitely* holds in a computation if it eventually becomes true in all runs of the computation (a *run* is a path in the lattice of consistent cuts starting from the initial consistent cut and ending at the final consistent cut). The modalities *invariant* and *controllable* are duals of the predicates *possibly* and *definitely*, respectively. Monitoring has applications in the areas of testing and debugging and software fault tolerance of distributed programs.

We show how to monitor a regular predicate under *possibly*: b, *invariant*: b and *controllable*: b modalities using slicing. Given a directed graph G, let $\mathsf{scc}(G)$ denote the number of strongly connected components of G.

Theorem 5.32 *A regular predicate is*

1. **possibly** *true in a computation if and only if the slice of the computation with respect to the predicate has at least one non-trivial consistent cut, that is, it has at least two strongly connected components. Formally,*

$$\textit{possibly}\colon b \;\equiv\; \mathsf{scc}(\langle E, \rightarrow\rangle_b) \geqslant 2$$

2. **invariant** *in a computation if and only if the slice of the computation with respect to the predicate is cut-equivalent to the computation. Formally,*

$$\textit{invariant}\colon b \;\equiv\; \langle E, \rightarrow\rangle_b \cong \langle E, \rightarrow\rangle$$

3. **controllable** *in a computation if and only if the slice of the computation with respect to the predicate has the same number of strongly connected components as the computation. Formally,*

$$\textit{controllable}\colon b \;\equiv\; \mathsf{scc}(\langle E, \rightarrow\rangle_b) = \mathsf{scc}(\langle E, \rightarrow\rangle)$$

Proof: The first two propositions are easy to verify. We only prove the last proposition. As for the last proposition, it can be verified that a regular predicate is controllable in a computation if and only if there exists a path from the initial to the final consistent cut in the lattice (of consistent cuts) such that every consistent cut along the path satisfies the predicate [TG98b]. Note that the path from the initial to the final consistent cut actually corresponds to a longest chain in the lattice of consistent cuts. For a lattice L, let $\mathsf{height}(L)$ denote the length of a longest chain in L. Therefore if b is controllable in $\langle E, \rightarrow\rangle$, then a longest chain in $\mathcal{C}(E)$ is contained in $\mathcal{C}_b(E)$ as well and vice versa. This implies that $\mathsf{height}(\mathcal{C}(E)) \leqslant \mathsf{height}(\mathcal{C}_b(E))$. However, $\mathcal{C}_b(E) \subseteq \mathcal{C}(E)$ implying that $\mathsf{height}(\mathcal{C}_b(E)) \leqslant \mathsf{height}(\mathcal{C}(E))$. Therefore we have:

$$\textit{controllable}\colon b \;\equiv\; \mathsf{height}(\mathcal{C}(E)) = \mathsf{height}(\mathcal{C}_b(E))$$

For a finite distributive lattice L, the length of its longest chain is equal to the number of its join-irreducible elements [DP90]. In other words, $\mathsf{height}(L) = \mathcal{JI}(L)$. Also, as observed before in Section 5.6, for a directed graph, the number of join-irreducible elements of the lattice generated by its set of consistent cuts—including the two trivial consistent cuts—is same as the number of its strongly connected components. As a result, $\mathsf{height}(\mathcal{C}(E)) = \mathcal{JI}(\mathcal{C}(E)) = \mathsf{scc}(\langle E, \rightarrow \rangle)$ and $\mathsf{height}(\mathcal{C}_b(E)) = \mathcal{JI}(\mathcal{C}_b(E)) = \mathsf{scc}(\langle E, \rightarrow \rangle_b)$. $\square$

Zig-Zag Consistency Theorem: A Special Case of Slicing

We now show how slicing relates to some of the well-known results in checkpointing. Consider a conjunctive predicate such that the local predicate for an event on a process is true if and only if the event corresponds to a local checkpoint. It can be verified that there is a *zigzag path* [NX95, Wan97] from a local checkpoint c to a local checkpoint c' in a computation if and only if there is a path from $succ(c)$, if it exists, to c' in the corresponding slice—which can be ascertained by comparing $J_b(succ(c))$ and $J_b(c')$. An alternative formulation of the consistency theorem in [NX95] can thus be obtained as follows:

Theorem 5.33 *A set of local checkpoints can belong to the same consistent global snapshot if and only if the local checkpoints in the set are mutually consistent (including with itself) in the corresponding slice.*

Moreover, the R-graph (rollback-dependency graph) [Wan97] is path-equivalent to the slice when each contiguous sequence of false events on a process is merged with the nearest true event that occurs later on the process. The minimum consistent global checkpoint that contains a set of local checkpoints [Wan97] can be computed by taking the set union of J_b's for each local checkpoint in the set. The maximum consistent global checkpoint can be similarly obtained by using the dual of J_b.

5.8 Slicing for General Predicates

In this section, we describe our results on slicing for general predicates. We first prove that it is in general NP-hard to compute the slice for an arbitrary predicate. Nonetheless, polynomial-time algorithms can be developed for certain special classes of predicates. In particular, we provide efficient algorithm to compute the slice for a linear predicate and its dual—a post-linear predicate [CG98]. We next present the notion of *grafting* which can be used to compose two slices; grafting can be done with respect to meet or join operator as explained later. We provide efficient algorithms for grafting two slices. Grafting can be used to compute the slice for a predicate in DNF (disjunctive normal form). We further give three more applications of grafting. First, we demonstrate how grafting can be employed to compute the slice for a *co-regular predicate*—complement of a regular predicate—in polynomial-time. Second, using grafting, we derive a polynomial-time algorithm to the compute the slice for a k*-local predicate* for constant k; a k-local predicate depends on variables of at most k processes [SS95]. Lastly, we employ grafting to compute an *approximate slice*—in polynomial-time—for a predicate composed from regular and co-regular predicates, linear predicates and post-linear predicates, and k-local predicates, for constant k, using $\wedge$ and $\vee$ operators.

5.8.1 NP-Hardness Result

It is evident from the definition of slice that the following is true:

Observation 5.2 *The necessary and sufficient condition for the slice of a computation with respect to a predicate to be non-empty is that there exists a consistent cut of the computation that satisfies the predicate.*

However, finding out whether some consistent cut of the computation satisfies a predicate is an NP-complete problem [CG95]. Thus it is in general NP-complete to determine whether the slice for a predicate is non-empty. This further implies that computing the slice for an arbitrary predicate is an NP-hard problem. From the results of Chapter 3, it follows that this is the case even when the predicate is a singular 2-CNF (conjunctive normal form) predicate.

5.8.2 Computing the Slice for Linear Predicates and their Dual

Recall that a predicate is linear if given two consistent cuts that satisfy the predicate, the cut given by their set intersection also satisfies the predicate [CG98]. A post-linear predicate can defined dually [CG98]. In this section we prove that the slicing algorithm $\mathsf{Algo}_{5.3}$ for a regular predicate described in Section 5.7.1 can be used for a linear predicate as well. For a post-linear predicate, however, a slightly different version of the algorithm based on the notion of meet-irreducible element will be applicable.

Consider a computation $\langle E, \rightarrow \rangle$ and a linear predicate b. First, we extend the definition of $J_b(e)$ for an event e and a regular predicate b to the case when b is a linear predicate. It can be easily verified that $J_b(e)$ is uniquely defined for each event e even when b is a linear predicate. Now, consider the directed graph $\mathcal{G}_b(E)$ with vertices as events in E and an edge from an event e to an event f if and only if $J_b(e) \subseteq J_b(f)$. We establish that the directed graph $\mathcal{G}_b(E)$ is cut-equivalent to the slice $\langle E, \rightarrow \rangle_b$. It suffices to prove that $\mathcal{C}(\mathcal{G}_b(E))$ is the *smallest* sublattice of $\mathcal{C}(E)$ that contains $\mathcal{C}_b(E)$. To that end, the following lemma comes in useful. The lemma basically states that, for each event e, $J_b(e)$ is the *least consistent cut* of $\mathcal{G}_b(E)$ that contains e. (Note that $J_b(e) \subseteq J_b(f)$ is equivalent to saying that there is an path from e to f in $\mathcal{G}_b(E)$.)

Lemma 5.34 *Given events e and f, $e \in J_b(f) \equiv J_b(e) \subseteq J_b(f)$.*

Proof: ($\Rightarrow$) Assume that $e \in J_b(f)$. Let $C = J_b(e) \cap J_b(f)$. Since $e \in J_b(e)$, $e \in C$. Note that $J_b(e)$ and $J_b(f)$ are consistent cuts of $\langle E, \rightarrow \rangle$. Moreover, both of them satisfy b. Since b is a linear predicate, their conjunction, given by C, also satisfies b. This implies that C is a consistent cut of $\langle E, \rightarrow \rangle$ which contains e and satisfies b. However, $J_b(e)$ is the *least* such cut. Therefore $J_b(e) \subseteq C$ or $J_b(e) \subseteq J_b(e) \cap J_b(f)$. This implies that $J_b(e) = J_b(e) \cap J_b(f)$. Equivalently, $J_b(e) \subseteq J_b(f)$.

($\Leftarrow$) Assume that $J_b(e) \subseteq J_b(f)$. Since $e \in J_b(E)$, trivially, $e \in J_b(f)$. $\square$

Again, as before, let $\mathcal{J}_b(E) = \{ J_b(e) \mid e \in E \}$. Using Lemma 5.34, the following theorem can be proved in a similar fashion as Lemma 5.16 and Lemma 5.17.

Theorem 5.35 *$\mathcal{C}(\mathcal{G}_b(E))$ forms a distributive lattice under $\subseteq$. Further, the set of join-irreducible elements of $\mathcal{C}(\mathcal{G}_b(E))$ is given by $\mathcal{J}_b(E)$.*

The next lemma demonstrates that $\mathcal{C}(\mathcal{G}_b(E))$ contains at least $\mathcal{C}_b(E)$.

Lemma 5.36 *Every consistent cut in $\mathcal{C}_b(E)$ can be written as the join of some subset of elements in $\mathcal{J}_b(E)$.*

The proof of the above lemma is similar to the proof of Lemma 5.17 and therefore has been omitted. Observe that, for every event e, by definition, either $J_b(e)$ satisfies b or is same as E. In either case, $J_b(e) \in \mathcal{C}_b(E)$. Therefore we have,

Observation 5.3 $\mathcal{J}_b(E) \subseteq \mathcal{C}_b(E)$.

Finally, the next theorem establishes that $\mathcal{C}(\mathcal{G}_b(E))$ is indeed the smallest sublattice of $\mathcal{C}(E)$ that contains all consistent cuts satisfying b.

Theorem 5.37 *Any sublattice of $\mathcal{C}(E)$ that contains $\mathcal{C}_b(E)$ also contains $\mathcal{C}(\mathcal{G}_b(E))$.*

Proof: Consider a sublattice $\mathcal{D}$ of $\mathcal{C}(E)$ such that $\mathcal{D}$ contains $\mathcal{C}_b(E)$. Also, consider a consistent cut C of $\mathcal{C}(\mathcal{G}_b(E))$. From Birkhoff's Representation Theorem and Theorem 5.35, C can be expressed as the join of some subset of elements in $\mathcal{J}_b(E)$.

Since $\mathcal{J}_b(E) \subseteq \mathcal{C}_b(E)$ and $\mathcal{C}_b(E) \subseteq \mathcal{D}$, $\mathcal{J}_b(E) \subseteq \mathcal{D}$. This implies that C can be written as the join of some subset of elements in $\mathcal{D}$. However, $\mathcal{D}$ is a sublattice and thus closed under set union. Therefore $C \in \mathcal{D}$. $\square$

The directed graph $\mathcal{G}_b(E)$ has $|E|$ vertices and can have as many as $\Omega(|E|^2)$ edges. However, by constructing $\mathcal{S}_b(E)$, the skeletal representation of $\langle E, \rightarrow \rangle_b$, instead of $\mathcal{G}_b(E)$, the number of edges and the time-complexity can be reduced to $O(n|E|)$ and $O(n^2|E|)$, respectively.

5.8.3 Grafting Two Slices

Given two slices, grafting can be used to either compute the smallest slice that contains all consistent cuts common to both slices—grafting with respect to meet—or compute the smallest slice that contains consistent cuts of both slices—grafting with respect to join. In other words, given slices $\langle E, \rightarrow \rangle_{b_1}$ and $\langle E, \rightarrow \rangle_{b_2}$, where b_1 and b_2 are regular predicates, grafting can be used to compute the slice $\langle E, \rightarrow \rangle_b$, where b is either $b_1 \sqcap b_2 = b_1 \wedge b_2$ or $b_1 \sqcup b_2 = reg\,(b_1 \vee b_2)$. Grafting enables us to compute the exact slice for an arbitrary boolean expression of local predicates—by rewriting it in DNF—although it may require exponential time in the worst case.

Grafting with respect to Meet: $\mathbf{b \equiv b_1 \sqcap b_2 \equiv b_1 \wedge b_2}$

In this case, the slice $\langle E, \rightarrow \rangle_b$ contains a consistent cut of $\langle E, \rightarrow \rangle$ if and only if the cut satisfies b_1 as well as b_2. Given an event e, let $F_{\min}(e)$ denote the vector obtained by taking componentwise minimum of $F_{b_1}(e)$ and $F_{b_2}(e)$. We first prove that no component of $F_{\min}(e)$ is less than (or occurs before) the corresponding component of $F_b(e)$.

Lemma 5.38 *For each event e and process p_i,*

$$F_b(e)[i] \overset{P}{\rightarrow} F_{\min}(e)[i]$$

Proof: It is sufficient to prove that $F_b(e)[i] \xrightarrow{P} F_{b_1}(e)[i]$ and $F_b(e)[i] \xrightarrow{P} F_{b_2}(e)[i]$ for each event e and process p_i. Assume, on the contrary, that $F_{b_1}(e)[i] \xrightarrow{P} F_b(e)[i]$ for some event e and process p_i. For convenience, let $F_{b_1}(e)[i] = f$. Consider $J_b(f)$. Observe that $J_b(f)$ contains f and is also a consistent cut of $\langle E, \rightarrow\rangle_{b_1}$. By definition of $\mathcal{S}_{b_1}(E)$, any consistent of $\langle E, \rightarrow\rangle_{b_1}$ that contains f also contains e because $f = F_{b_1}(e)[i]$. This implies that $J_b(f)$ contains e. Since $J_b(f)$ is the *least* consistent cut of $\langle E, \rightarrow\rangle_b$ that contains f, there is a path from e to f in $\mathcal{S}_b(E)$. Using Lemma 5.20, $J_b(e) \subseteq J_b(f)$ which contradicts our choice of $F_b(e)[i]$. $\square$

We now construct a directed graph $\mathcal{S}_{\min}(E)$ that is similar to $\mathcal{S}_b(E)$ except that we use $F_{\min}$ instead of F_b in its construction. The following theorem proves that $\mathcal{S}_{\min}(E)$ is in fact cut-equivalent to $\mathcal{S}_b(E)$.

Theorem 5.39 *$\mathcal{S}_{\min}(E)$ is cut-equivalent to $\mathcal{S}_b(E)$.*

Proof: We have,

$$
\begin{aligned}
& \{ \text{ definition of } F_{\min} \} \\
& \Big(\mathcal{P}(\mathcal{S}_{b_1}(E)) \subseteq \mathcal{P}(\mathcal{S}_{\min}(E))\Big) \bigwedge \Big(\mathcal{P}(\mathcal{S}_{b_2}(E)) \subseteq \mathcal{P}(\mathcal{S}_{\min}(E))\Big) \\
\equiv\ & \{ \text{ using Lemma 5.1 } \} \\
& \Big(\mathcal{C}(\mathcal{S}_{\min}(E)) \subseteq \mathcal{C}(\mathcal{S}_{b_1}(E))\Big) \bigwedge \Big(\mathcal{C}(\mathcal{S}_{\min}(E)) \subseteq \mathcal{C}(\mathcal{S}_{b_2}(E))\Big) \\
\equiv\ & \{ \text{ set calculus } \} \\
& \mathcal{C}(\mathcal{S}_{\min}(E)) \subseteq \Big(\mathcal{C}(\mathcal{S}_{b_1}(E)) \cap \mathcal{C}(\mathcal{S}_{b_2}(E))\Big) \\
\equiv\ & \{ b \equiv b_1 \wedge b_2 \} \\
& \mathcal{C}(\mathcal{S}_{\min}(E)) \subseteq \mathcal{C}(\mathcal{S}_b(E))
\end{aligned}
$$

Also, we have,

$$
\begin{aligned}
& \{ \text{ using Lemma 5.38 } \} \\
& \mathcal{P}(\mathcal{S}_{\min}(E)) \subseteq \mathcal{P}(\mathcal{S}_b(E)) \\
\equiv\ & \{ \text{ using Lemma 5.1 } \}
\end{aligned}
$$

$$\mathcal{C}(\mathcal{S}_b(E)) \subseteq \mathcal{C}(\mathcal{S}_{\min}(E))$$

Thus $\mathcal{C}(\mathcal{S}_{\min}(E)) = \mathcal{C}(\mathcal{S}_b(E))$. $\square$

Roughly speaking, the aforementioned algorithm computes the union of the sets of edges of each slice. Note that, in general, $F_b(e)[i]$ need not be same as $F_{\min}(e)[i]$. This algorithm can be generalized to conjunction of an arbitrary number of regular predicates.

Grafting with respect to Join: $\mathbf{b \equiv b_1 \sqcup b_2 \equiv reg\,(b_1 \vee b_2)}$

In this case, the slice $\langle E, \rightarrow \rangle_b$ contains a consistent cut of $\langle E, \rightarrow \rangle$ if the cut satisfies either b_1 or b_2. Given an event e, let $F_{\max}(e)$ denote the vector obtained by taking componentwise maximum of $F_{b_1}(e)$ and $F_{b_2}(e)$. We first prove that no component of $F_b(e)$ is less than (or occurs before) the corresponding component of $F_{\max}(e)$.

Lemma 5.40 *For each event e and process p_i,*

$$F_{\max}(e)[i] \stackrel{P}{\rightarrow} F_b(e)[i]$$

The proof of Lemma 5.40 is similar to that of Lemma 5.38 and therefore has been omitted. We now construct a directed graph $\mathcal{S}_{\max}(E)$ that is similar to $\mathcal{S}_b(E)$ except that we use $F_{\max}$ instead of F_b in its construction. The following theorem proves that $\mathcal{S}_{\max}(E)$ is in fact cut-equivalent to $\mathcal{S}_b(E)$.

Theorem 5.41 *$\mathcal{S}_{\max}(E)$ is cut-equivalent to $\mathcal{S}_b(E)$.*

Again, the proof of Theorem 5.41 is similar to that of Theorem 5.39 and hence has been omitted. Intuitively, the above-mentioned algorithm computes the intersection of the sets of edges of each slice. In this case, in contrast to the former case, $F_b(e)[i]$ is identical to $F_{\max}(e)[i]$. The reason is as follows. Recall that $F_b(e)[i]$ is the earliest event on p_i that is reachable from e in $\langle E, \rightarrow \rangle_b$. From Theorem 5.41, at least $F_{\max}(e)[i]$ is reachable from e in $\langle E, \rightarrow \rangle_b$. Thus $F_b(e)[i] \overset{P}{\rightarrow} F_{\max}(e)[i]$. Combining it with Lemma 5.40, we obtain the required result. This algorithm can be generalized to disjunction of an arbitrary number of regular predicates.

5.8.4 Computing the Slice for Co-Regular Predicates

Given a regular predicate, we give an algorithm to compute the slice of a computation with respect to its negation—a co-regular predicate. In particular, we express the negation as disjunction of polynomial number of regular predicates. The slice can then be computed by grafting together slices for each disjunct.

Consider a computation $\langle E, \rightarrow \rangle$ and a regular predicate b. For convenience, let $\rightarrow_b$ be the edge relation for the slice $\langle E, \rightarrow \rangle_b$. Without loss of generality, assume that both $\rightarrow$ and $\rightarrow_b$ are transitive relations. Our objective is to find a property that *distinguishes* the consistent cuts that belong to the slice from the consistent cuts that do not. Consider events e and f such that $e \not\rightarrow f$ but $e \rightarrow_b f$. Then, clearly, a consistent cut that contains f but does not contain e cannot belong to the slice. On the other hand, every consistent cut of the slice that contains f also contains e. This motivates us to define a predicate $\mathit{prevents}(f, e)$ as follows:

$$C \text{ satisfies } \mathit{prevents}(f, e) \;\triangleq\; (f \in C) \wedge (e \notin C)$$

We now prove that the predicate $\mathit{prevents}(f, e)$ is actually a regular predicate. Specifically, we establish that $\mathit{prevents}(f, e)$ is a conjunctive predicate.

Lemma 5.42 *$\mathit{prevents}(f, e)$ is a conjunctive predicate.*

Proof: Let $\mathit{proc}(e) = p_i$ and $\mathit{proc}(f) = p_j$. We define a local predicate $l_i(e)$ to be true for an event g on process p_i if $g \overset{P}{\rightarrow} e$. Similarly, we define a local predicate $m_j(f)$ to be true for an event h on process p_j if $f \overset{P}{\rightarrow} h$. Clearly, $\mathit{prevents}(f, e)$ is equivalent to $l_i(e) \wedge m_j(f)$. □

It turns out that every consistent cut that does not belong to the slice satisfies

$prevents(f,e)$ for some pair of events (e,f) such that $(e \not\rightarrow f) \wedge (e \rightarrow_b f)$ holds. Formally,

Theorem 5.43 *Let C be a consistent cut of $\langle E, \rightarrow \rangle$. Then,*

$$C \;\; \textit{satisfies} \;\; \neg b \;\equiv\; \langle \exists\, e, f : (e \not\rightarrow f) \wedge (e \rightarrow_b f) : C \;\; \textit{satisfies} \;\; \textit{prevents}(f,e) \rangle$$

Proof: We have,

$$
\begin{array}{cl}
 & C \text{ satisfies } \neg b \\
\equiv & \{\ b \text{ is a regular predicate }\} \\
 & \neg\Big(C \in \mathcal{C}(\langle E, \rightarrow \rangle_b)\Big) \\
\equiv & \{\ \text{definition of } \mathcal{C}(\langle E, \rightarrow \rangle_b)\ \} \\
 & \neg \langle \forall\, e, f : e \rightarrow_b f : f \in C \Rightarrow e \in C \rangle \\
\equiv & \{\ \text{predicate calculus }\} \\
 & \langle \exists\, e, f : e \rightarrow_b f : (f \in C) \wedge (e \notin C) \rangle \\
\equiv & \{\ \text{definition of } \textit{prevents}(f,e)\ \} \\
 & \langle \exists\, e, f : e \rightarrow_b f : C \;\; \text{satisfies} \;\; \textit{prevents}(f,e) \rangle \\
\equiv & \{\ \text{predicate calculus }\} \\
 & \langle \exists\, e, f : (e \rightarrow_b f) \bigwedge \Big((e \rightarrow f) \;\vee\; (e \not\rightarrow f)\Big) : C \;\; \text{satisfies} \;\; \textit{prevents}(f,e) \rangle \\
\equiv & \{\ e \rightarrow f \;\; \text{implies} \;\; e \rightarrow_b f\ \} \\
 & \langle \exists\, e, f : (e \rightarrow f) \bigvee \Big((e \rightarrow_b f) \;\wedge\; (e \not\rightarrow f)\Big) : C \;\; \text{satisfies} \;\; \textit{prevents}(f,e) \rangle \\
\equiv & \left\{ \begin{array}{l} \text{since } C \text{ is a consistent cut of } \langle E, \rightarrow \rangle,\ C \text{ satisfies } \textit{prevents}(f,e) \Rightarrow \\ e \not\rightarrow f \end{array} \right\} \\
 & \langle \exists\, e, f : (e \rightarrow_b f) \;\wedge\; (e \not\rightarrow f) : C \text{ satisfies } \textit{prevents}(f,e) \rangle
\end{array}
$$

This establishes the theorem. □

Theorem 5.43 can also be derived using the results in lattice theory [Riv74]. We now give the time-complexity of the algorithm. We start by making the following observation.

Observation 5.4 *Let e, f and g be events such that $f \rightarrow g$. Then,*

$$prevents(g, e) \;\Rightarrow\; prevents(f, e)$$

Let $K_b(e)$ denote the vector whose i^{th} entry denote the earliest event f on process p_i, if it exists, such that $(e \not\rightarrow f) \wedge (e \rightarrow_b f)$ holds. Observation 5.4 implies that $prevents(K_b(e)[i], e)$, whenever $K_b(e)[i]$ exists, is the weakest predicate among all predicates $prevents(f, e)$, where $proc(f) = p_i$ and $(e \not\rightarrow f) \wedge (e \rightarrow_b f)$. Thus we can ignore all other events on p_i for the purpose of computing the slice for a co-regular predicate. More precisely, Theorem 5.43 can be restated as:

Theorem 5.44 *Let C be a consistent cut of $\langle E, \rightarrow \rangle$. Then,*

$$C \;\; satisfies \;\; \neg b \;\equiv\; \langle \exists\, e, p_i :: C \;\; satisfies \;\; prevents(K_b(e)[i], e) \rangle$$

It turns out that $K_b(e)[i]$ and $F_b(e)[i]$ are closely related.

Observation 5.5 *$K_b(e)[i]$ exists if and only if $e \not\rightarrow F_b(e)[i]$. Moreover, whenever $K_b(e)[i]$ exists it is identical to $F_b(e)[i]$.*

Note that, to compute the slice for $\neg b$, we actually compute the slice for $reg\,(\neg b)$, that is, $\langle E, \rightarrow \rangle_{\neg b} = \langle E, \rightarrow \rangle_{reg\,(\neg b)}$. Theorem 5.44 implies that the number of disjuncts in the predicate equivalent to the negation of a regular predicate is at most $O(n|E|)$. Further, these disjuncts can be determined in $O(n^2|E|)$ time using the algorithm $\mathsf{Algo}_{\,5.2}$. The slice with respect to each disjunct can be computed in $O(|E|)$ time using the algorithm given in Section 5.7.3. Moreover, given a disjunct $b^{(i)}$, $J_{b^{(i)}}(e)$ for each event e can be computed in $O(n|E|)$ time which in turn can be used to determine $F_{b^{(i)}}(e)$ for each event e in $O(n|E|)$ time using the algorithm $\mathsf{Algo}_{\,5.2}$. Finally, these slices can be grafted together to produce the slice for a co-regular predicate in $O(n|E| \times n|E|) = O(n^2|E|^2)$ time. This is because, given an event e, computing each entry of $F_{b'}(e)$, where $b' = reg\,(\neg b)$, requires $O(n|E|)$ time. Thus the overall time-complexity of the algorithm is $O(n^2|E|+n^2|E|^2) = O(n^2|E|^2)$.

5.8.5 Computing the Slice for k-Local Predicates for Constant k

A predicate is called *k-local* if it depends on variables of at most k processes [SS95]. To compute the slice for a k-local predicate, we use the technique developed by Stoller and Schneider [SS95]. Given a computation, their technique can be used to transform a k-local predicate into a predicate in k-DNF (disjunctive normal form) with at most m^{k-1} clauses, where m is the maximum number of events on a process. For example, consider the predicate $x_1 \neq x_2$. Let V denote the set of values that x_1 can take in the given computation. Note that $|V| \leqslant m$. Then $x_1 \neq x_2$ can be rewritten as:

$$x_1 \neq x_2 \;\equiv\; \bigvee_{v \in V} \Big((x_1 = v) \wedge (x_2 \neq v) \Big)$$

Each clause in the resultant k-DNF predicate will be a conjunctive predicate. We can use the optimal $O(|E|)$ algorithm given in Section 5.7.3 to compute the slice for each clause. These slices can then be grafted together with respect to disjunction to obtain the slice for the given k-local predicate. The time-complexity of the algorithm is $O(m^{k-1}n|E|)$.

5.8.6 Computing Approximate Slices

Even though it is, in general, NP-hard to compute the slice for an arbitrary predicate, it is still possible to compute an *approximate slice* in many cases. The slice is "approximate" in the sense that it is bigger than the actual slice for the predicate. Nonetheless, it still contains all consistent cuts of the computation that satisfy the predicate. In many cases, the approximate slice that we obtain is much smaller than the computation itself and therefore can be used to prune the search-space for many intractable problems such as monitoring predicates under various modalities.

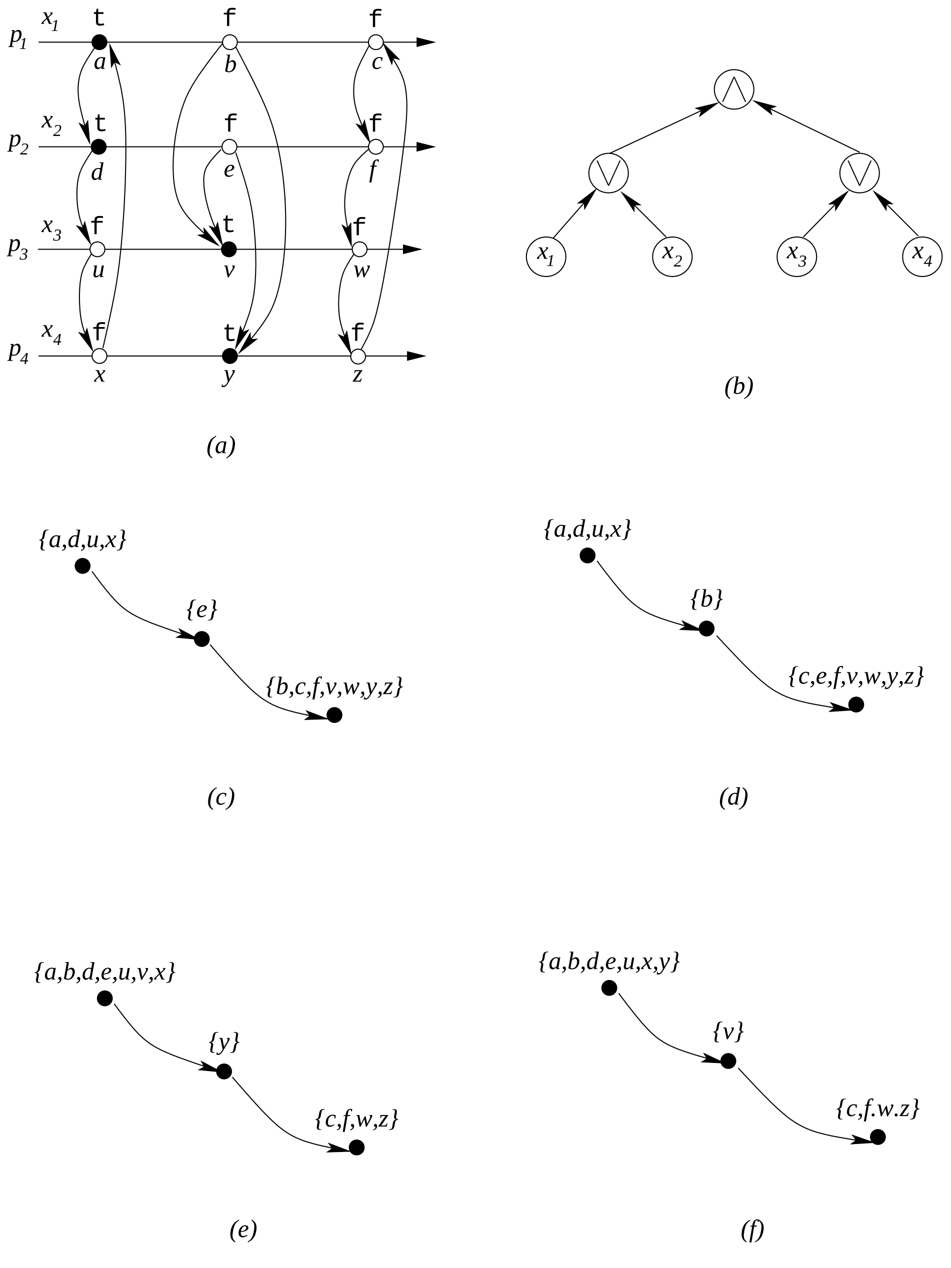


Figure 5.12: (a) A computation, (b) the parse tree for the predicate $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$, (c) the slice with respect to x_1, (d) the slice with respect to x_2, (e) the slice with respect to x_3, (f) the slice with respect to x_4.

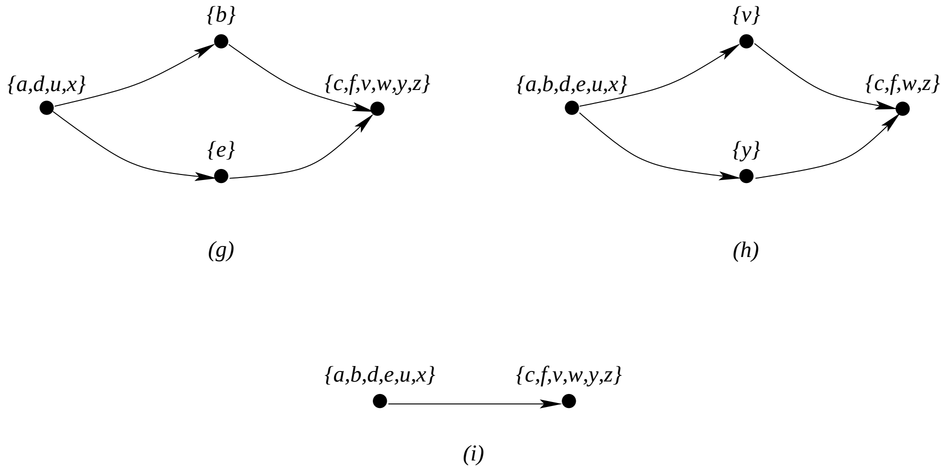


Figure 5.13: (continuation of Figure 5.12) (g) the slice with respect to $x_1 \vee x_2$, (h) the slice with respect to $x_3 \vee x_4$, and (i) the slice with respect to $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$.

In particular, using grafting and the algorithms for computing the slice for various classes of predicates, it is possible to efficiently compute an approximate slice for a predicate derived from linear predicates, post-linear predicates, regular predicates, co-regular predicates, and k-local predicates for constant k using $\wedge$ and $\vee$ operators.

To compute an approximate slice, we first build the parse tree for the given boolean expression; all predicates occupy leaf nodes whereas all operators occupy non-leaf nodes. We then recursively compute the slice working our way up from leaf nodes to the root. For a leaf node, we use the algorithm appropriate for the predicate corresponding to the leaf node. For example, if the leaf node corresponds to a linear predicate, we use the algorithm described in Section 5.8.2. For the conjunction and disjunction operators, $\wedge$ and $\vee$, we use the suitable grafting algorithm depending on the operator.

Example 5.8 *For example, consider the computation depicted in Figure 5.12(a) and the predicate* $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$*. The parse tree corresponding to the predicate* *is shown in Figure 5.12(b). To compute an approximate slice for the predicate, we first compute slices for the (local) predicates* x_1*,* x_2*,* x_3 *and* x_4 *as shown in Figure 5.12(c)-(f). We then graft the first two and the last two slices together with respect to join to obtain slices for the clauses* $x_1 \vee x_2$ *and* $x_3 \vee x_4$ *as portrayed in Figure 5.13(g) and Figure 5.13(h), respectively. For the ease of understanding, the events belonging to the same strongly connected component are shown together in a subset. Finally, we graft the slices for both clauses together with respect to meet. The slice obtained will contain all consistent cuts that satisfy the predicate* $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$*. The final slice is shown in Figure 5.13(i).*

As shown in the figure, the computation has seven non-trivial consistent cuts, namely $\{a, d, u, x\}$*,* $\{a, b, d, u, x\}$*,* $\{a, d, e, u, x\}$*,* $\{a, b, d, e, u, x\}$*,* $\{a, b, d, e, u, v, x\}$*,* $\{a, b, d, e, u, x, y\}$ *and* $\{a, b, d, e, u,\ v, x, y\}$*. On the other hand, the slice consists of only a single non-trivial consistent cut, which is given by* $\{a, b, d, e, u, x\}$*. The final slice corresponds to the predicate* $reg\left(reg\,(x_1 \vee x_2) \wedge reg\,(x_3 \vee x_4)\right)$ *and not to the predicate* $reg\left((x_1 \vee x_2) \wedge (x_3 \vee x_4)\right)$ *as desired. This is expected because detecting even a predicate in 2-CNF when no two clauses contain variables from the same process is NP-complete in general (see Chapter 3).*

5.9 Detecting Global Predicates using Slicing: An Experimental Study

In this section, we evaluate the effectiveness of slicing in pruning the search-space when detecting a predicate under *possibly* modality. We compare our approach with that of Stoller, Unnikrishnan and Liu [SUL00], which is based on *partial-order methods* [God96]. Intuitively, when searching the state-space, at each consistent cut, partial-order methods allow only a small subset of enabled transitions to be explored. In particular, we use partial-order methods employing both persistent and sleep sets for comparison. We consider two examples that were also used by Stoller, Unnikrishnan and Liu to evaluate their approach [SUL00]. We briefly describe the main idea behind partial-order methods approach here; details can be found elsewhere [God96, SUL00].

The material in the next two paragraphs is paraphrased from [SUL00]. In their full generality, partial-order methods can be used to locate deadlocks in a concurrent system. A *deadlock* is a state in which no transitions are enabled. Clearly, all reachable deadlocks can be identified by exploring all reachable states. This involves explicitly considering all possible execution orderings of transitions, even if some transitions are "independent" (that is, executing them in any order leads to the same state). Exploring one interleaving of independent transitions is sufficient for finding deadlocks. This causes fewer intermediate states (that is, states in which some but not all of the independent transitions have been executed) to be explored, but it does not affect the reachability of deadlocks. This is because the intermediate states cannot be deadlocks since some of the independent transitions are enabled in those states. Partial-order methods attempt to eliminate exploration of multiple interleavings of independent transitions, thereby saving time and space.

Consider a state s. A set T of transitions enabled in s is said to be *persistent* in s if, for every sequence of transitions starting from s and not containing any transitions in T, all transitions in that sequence are independent with all transitions in T. As shown in [God96], in order to find all reachable deadlocks, it suffices to explore from each state s a set of transitions that is persistent in s. Note that the set of all enabled transitions in s trivially constitutes a persistent set in s. To save time and space, small persistent sets should be used. As further optimization, *sleep sets* can be employed to eliminate redundancy caused by exploring multiple interleavings of independent transitions in a persistent set [God96].

How do partial-order methods apply to detecting a predicate under *possibly* modality? Consider a predicate $b = b^{(1)} \wedge b^{(2)} \wedge \cdots \wedge b^{(l)}$. Let $support(b^{(i)})$ denote the subset of processes on which the conjunct $b^{(i)}$ depends. Suppose, when exploring the state-space of the computation, we reach a consistent cut C that does not satisfy b. Therefore there exists a conjunct $b^{(i)}$ that evaluates to false for C. A set T of transitions that constitutes a persistent set in C can be constructed as follows. For each process $p_j \in support(b^{(i)})$, in case the next transition t_j of p_j, if it exists, is enabled in C, add t_j to T; otherwise find some enabled transition t that must be executed before t_j and add t to T.

Now, with our approach based on computation slicing, in order to detect a predicate, we first compute an approximate slice of the computation with respect to the predicate, and then perform a simple search of the state-space of the resultant slice. Whereas, with the approach based on partial-order methods, we use persistent and sleep sets to search the state-space of the computation. To compare the two approaches, we consider two examples which were also used by Stoller, Unnikrishnan and Liu to evaluate their approach [SUL00].

The first example, called *primary-secondary*, concerns an algorithm designed to ensure that the system always contains a pair of processes acting together as primary and secondary. The invariant for the algorithm requires that there is a pair of processes p_i and p_j such that (1) p_i is acting as a primary and correctly thinks that p_j is its secondary, and (2) p_j is acting as a secondary and correctly thinks that p_i is its primary.

The first example, called *primary-secondary*, concerns an algorithm designed to ensure that the system always contains a pair of processes acting together as primary and secondary. The invariant for the algorithm requires that there is a pair of processes p_i and p_j such that (1) p_i is acting as a primary and correctly thinks that p_j is its secondary, and (2) p_j is acting as a secondary and correctly thinks that p_i is its primary. Both the primary and the secondary may choose new processes as their successor at any time; the algorithm must ensure that the invariant is never falsified. Mathematically, the invariant I_{ps} can be written as:

$$I_{ps} \quad = \bigvee_{i,\, j \,\in\, [1 \ldots n],\; i \neq j} \left(\begin{array}{l} isPrimary_i \,\bigwedge\, isSecondary_j \,\bigwedge \\ (secondary_i = p_j) \,\bigwedge\, (primary_j = p_i) \end{array} \right)$$

Here, the variable $isPrimary_i$ is true if and only if process p_i is acting as the primary; in that case, the variable $secondary_i$ points to the process that p_i thinks is acting as its secondary. The variables $isSecondary_i$ and $primary_i$ can be interpreted in a similar fashion. Both the primary and the secondary may choose new processes as their successor at any time; the algorithm must ensure that the invariant is never falsified. Stoller, Unnikrishnan and Liu provide an algorithm in [SUL00] to maintain the above invariant. We describe it here for the sake of completeness.

Initially, process p_1 is the primary and process p_2 is the secondary. At any time, the primary may choose a new primary as its successor by first informing the secondary of its intention, waiting for an acknowledgement, and then multicasting to the other processes a request for volunteers to be the new primary. It chooses the first volunteer whose reply it receives and sends message to that process stating that it is the new primary. The new primary sends a message to the current secondary which updates its state to reflect the change and then sends a message to the old primary stating that it can stop being the primary. The secondary can choose a new secondary using a similar protocol. Before initiating the protocol, however, the secondary must wait for an acknowledgement from the primary. If the secondary instead receives a message that the primary is searching for a successor as well, the secondary aborts its current attempt to find a successor, waits until it receives a message from the new primary, and then re-starts the protocol. This prevents the primary and secondary from trying to choose successors concurrently. A global fault corresponds to the complement of the invariant which can be expressed as:

$$\neg I_{ps} \;=\; \bigwedge_{i,j\,\in\,[1\ldots n],\; i\neq j} \left(\begin{array}{l} \neg isPrimary_i \bigvee \neg isSecondary_j \bigvee \\ (secondary_i \neq p_j) \bigvee (primary_j \neq p_i) \end{array} \right)$$

Note that $\neg I_{ps}$ is a predicate in CNF where each clause is a disjunction of two local predicates. An approximate slice for $\neg I_{ps}$ can be computed in $O(n^3|E|)$ time.

In the second example, called *database partitioning*, a database is partitioned among processes p_2 through p_n, while process p_1 assigns tasks to these processes based on the current partition. A process p_i, $i \in [2 \ldots n]$, can suggest a new partition at any time by setting variable $change_i$ to true and broadcasting a message containing the proposed partition and an appropriate version number. A recipient of this message accepts the proposed partition if its own version of the partition has a smaller version number or if its own version of the partition has the same version number and was proposed by a process with larger index. An invariant that should be maintained is: if no process is changing the partition, then all processes agree on the partition. Formally,

$$I_{db} = (\bigwedge_{i\in[2\ldots n]} \neg change_i) \Rightarrow (\bigwedge_{1\leqslant i<j\leqslant n} partition_i = partition_j)$$

Again, the algorithm described above was given by Stoller, Unnikrishnan and Liu in [SUL00]. The complement of the invariant, given by $\neg I_{db}$, can be written as:

$$\neg I_{db} = (\bigwedge_{i\in[2\ldots n]} \neg change_i) \bigwedge \Big(\bigvee_{i,j\in[1\ldots n],\, i\neq j} (partition_i \neq partition_j) \Big)$$

Note that the first $n-1$ clauses of $\neg I_{db}$ are local predicates and the last clause, say LC, is a disjunction of 2-local predicates. Thus, using the technique described in Section 5.8.5, LC can be rewritten as a predicate in DNF with $O(n|E|)$ clauses. To reduce the number of clauses, we proceed as follows. Let V denote the set of values that $partition_1$ assumes in the given computation. Then it can be

	No Faults				One Injected Fault			
Number of Processes	Partial-Order Methods		Computation Slicing		Partial-Order Methods		Computation Slicing	
n	T	M	T	M	T	M	T	M
6	0.07	0.62	0.36	1.21	0.05	0.41	0.37	1.38
7	0.16	1.11	0.61	1.34	0.11	0.81	0.58	1.41
8	0.37	2.06	0.90	1.54	0.31	1.79	0.91	1.61
9	0.83	4.37	1.24	1.70	0.59	3.05	1.21	1.77
10	1.52	7.26	1.73	1.81	1.12	5.54	1.70	2.00
11	2.99*	13.14*	2.15	1.93	2.09*	9.50*	2.13	2.27
12	5.0*	21.56*	2.85	2.16	3.51*	14.13*	2.77	2.43

n: number of processes T: amount of time spent (in s)

M: amount of memory used (in MB)

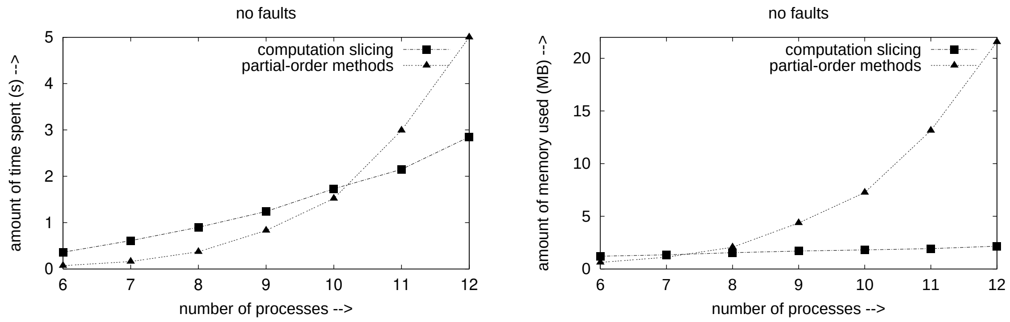
*: does not include the cases in which the technique runs out of memory

Table 5.1: Primary-Secondary example with the number of events on a process upper-bounded by 90.

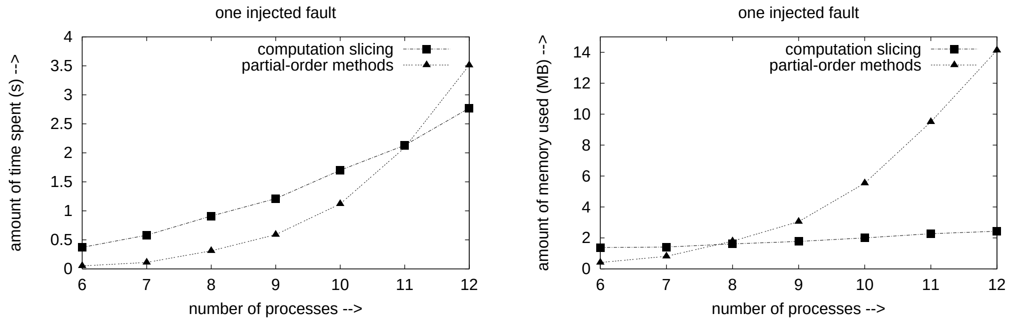
verified that LC is logically equivalent to:

$$\bigvee_{v \in V} \Big((partition_1 = v) \wedge \Big((partition_2 \neq v) \vee (partition_3 \neq v) \vee \cdots \vee (partition_n \neq v) \Big) \Big)$$

This decreases the number of clauses, when LC is rewritten in a form that can be used to compute a slice, to $O(n|V|)$. Note that $|V|$ is bounded by the number of events on the first process, and therefore we expect $n|V|$ to be $O(|E|)$. We use the simulator implemented in Java by Stoller, Unnikrishnan and Liu to generate computations of these protocols. Message latencies and other delays (*e.g.*, how long to wait before looking for a new successor) are selected randomly using the distribution $1+\exp(x)$, where $\exp(x)$ is the exponential distribution with mean x. Further details of the two protocols and the simulator can be found elsewhere [SUL00]. We consider two different scenarios: *fault-free* and *faulty*. The simulator always produces fault-free computations. A faulty computation is generated by



(a)



(b)

Figure 5.14: Primary-Secondary example with the number of events on a process upper-bounded by 90 for (a) no faults and (b) one injected fault.

randomly injecting faults into a fault-free computation. Note that in the first (fault-free) scenario, we know *a priori* that the computation does not contain a faulty consistent cut. We cannot, however, assume the availability of such knowledge in general. Thus it is important to study the behaviour of the two predicate detection techniques in the fault-free scenario as well. We implement the algorithm for slicing a computation in Java. We compare the two predicate detection techniques with respect to two metrics: amount of time spent and amount of memory used. In the case of the former technique, both metrics also include the overhead of computing the slice. We run our experiments on a machine with Pentium 4 processor operating at 1.8GHz clock frequency and 512MB of physical memory.

For primary-secondary example, the simulator is run until the number of events on some process reaches 90. The measurements averaged over 300 computations are displayed in Table 5.1. With computation slicing, for fault-free computations, the slice is always empty. As the number of processes is increased from 6 to 12, the amount of time spent increases from 0.36s to 2.85s, whereas the amount of memory used increases from 1.21M to 2.16M. On the other hand, with partial-order methods, they increase, almost exponentially, from 0.07s to 5.0s and 0.62M to 21.56M, respectively. Even on injecting a fault, the slice stays quite small. After computing the slice, in our experiments, we only need to examine at the most 13 consistent cuts to locate a faulty consistent cut, if any. The amount of time spent and the amount of memory used, with computation slicing, increase from 0.37s to 2.77s and 1.38M to 2.43M, respectively, as the number of processes is increased from 6 to 12. However, with partial-order methods, they again increase almost exponentially from 0.05s to 3.51s and 0.41M to 14.13M, respectively. Clearly, with slicing, both time and space complexities for detecting a global fault, if it exists, in primary-secondary example are polynomial in input size for the specified range of parameters. In contrast, with partial-order methods, they are exponential in input size. Figure 5.14(a) and Figure 5.14(b) plot the variation in the two metrics with the number of processes for the two approaches.

The worst-case performance of the partial-order methods approach is quite bad. With 12 processes in the system and the limit on the memory set to 100MB, the approach runs out of memory in approximately 6% of the cases. In around two-thirds of such cases, the computation actually contains a consistent cut that does not satisfy the invariant. It may be noted that we do not include the above-mentioned cases in computing the average amount of time spent and memory used. Including them will only make the average performance of the partial-order methods approach worse. Further, the performance of the partial-order methods approach appears to be very

	No Faults				One Injected Fault			
Number of Processes	Partial-Order Methods		Computation Slicing		Partial-Order Methods		Computation Slicing	
n	T	M	T	M	T	M	T	M
4	0.05	0.07	0.24	1.06	0.03	0.05	0.24	0.95
5	0.05	0.09	0.34	1.13	0.03	0.08	0.36	0.99
6	0.05	0.13	0.50	1.22	0.03	0.10	0.48	1.13
7	0.05	0.22	0.59	1.33	0.04	0.16	0.62	1.25
8	0.07	0.31	0.76	1.41	0.04	0.23	0.73	1.57
9	0.07*	0.36*	0.89	1.56	0.05	0.31	0.92	1.69
10	0.08*	0.40*	1.09	1 .80	0.05*	0.42*	1.07	1.80

n: number of processes T: amount of time spent (in s)

M: amount of memory used (in MB)

*: does not include the cases in which the technique runs out of memory

Table 5.2: Database partitioning example with the number of events on a process upper-bounded by 80.

sensitive to the location of the fault, in particular, whether it occurs earlier during the search or much later or perhaps does not occur at all. Consequently, the variation or standard deviation in the two metrics is very large. This has implications when predicate detection is employed for achieving software fault tolerance. Specifically, it becomes hard to provision resources (in our case, memory) when using partial-order methods approach. If too little memory is reserved, then, in many cases, the predicate detection algorithm will not be able to run successfully to completion. On the other hand, if too much memory is reserved, the memory utilization will be sub-optimal.

For database partitioning example, the simulator is run until the number of events on some process reaches 80. The measurements averaged over 300 computations are shown in Table 5.2. Figure 5.15(c) and Figure 5.15(d) plot the variation in the two metrics with the number of processes for the two approaches. As

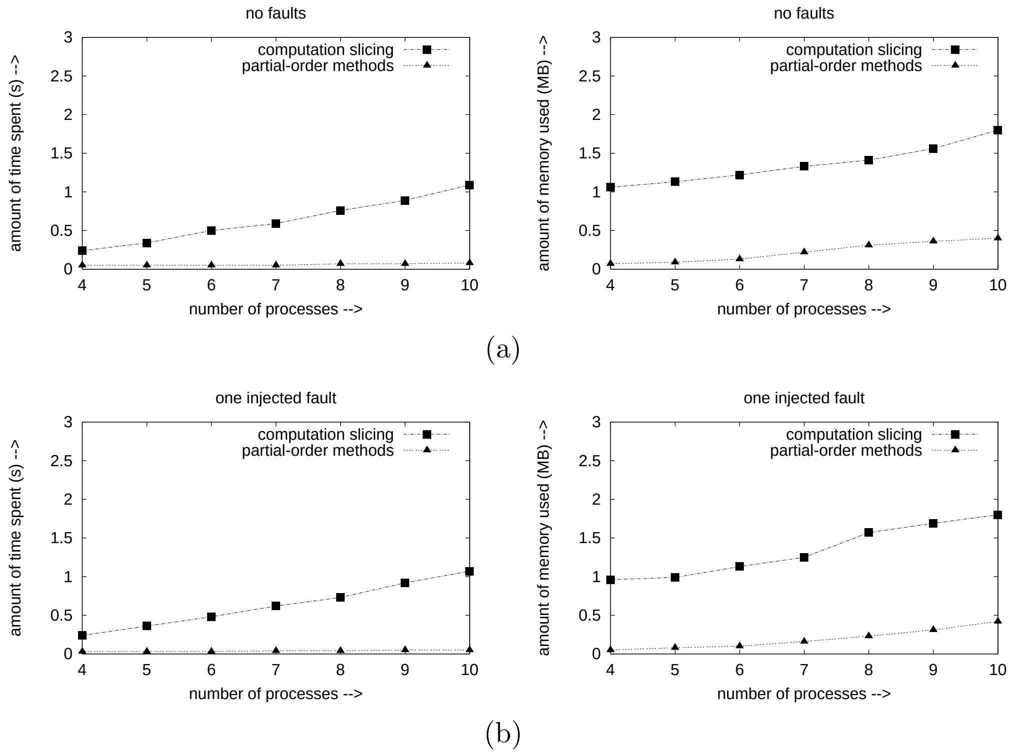


Figure 5.15: Database partitioning example with the number of events on a process upper-bounded by 80 for (c) no faults and (d) one injected fault.

it can be seen, the average performance of partial-order methods is much better than computation slicing. This is because substantial overhead is incurred in computing the slice. The slice itself is quite small. Specifically, for the fault-free scenario, the slice is always empty. On the other hand, for the faulty scenario, only at most 4 transitions need to be explored after computing the slice to locate a faulty consistent cut, if any.

Even for database partitioning example, for 10 processes, the partial-order methods approach runs out of memory in a small fraction—approximately 1%—of the cases. Therefore the worst-case performance of computation slicing is better than partial-order methods. To get the best of both worlds, predicate detection can be first done using the partial-order methods approach. In case it turns out that the approach is using too much memory, say more than $cn|E|$ for some small constant c, and still has not terminated, it can be aborted and the computation slicing approach can then be used for predicate detection.

Chapter 6

Related Work

In this chapter, we discuss the related research in three sections corresponding to the topics of each of the previous three chapters: detecting global predicates, controlling global predicates, and slicing distributed computations.

6.1 Detecting Global Predicates

The results in predicate detection presented in this dissertation were first published in [MG01b]. The predicate detection problem is known to be intractable in general under both *possibly* and *definitely* modalities [CG95, SS95, TG98b]. Approaches to detecting global predicates can be broadly classified into three categories. The first approach [CL85, SK86, Bou87, HJPR87] is based on repeatedly computing a consistent cut of the system using a global snapshot protocol and verifying whether the cut satisfies the given predicate until the predicate becomes true. This method is suitable only for stable predicates—the predicates which stay true once they become true. Some examples of stable predicates are deadlock and termination. The approach cannot be used to detect unstable predicates which may become

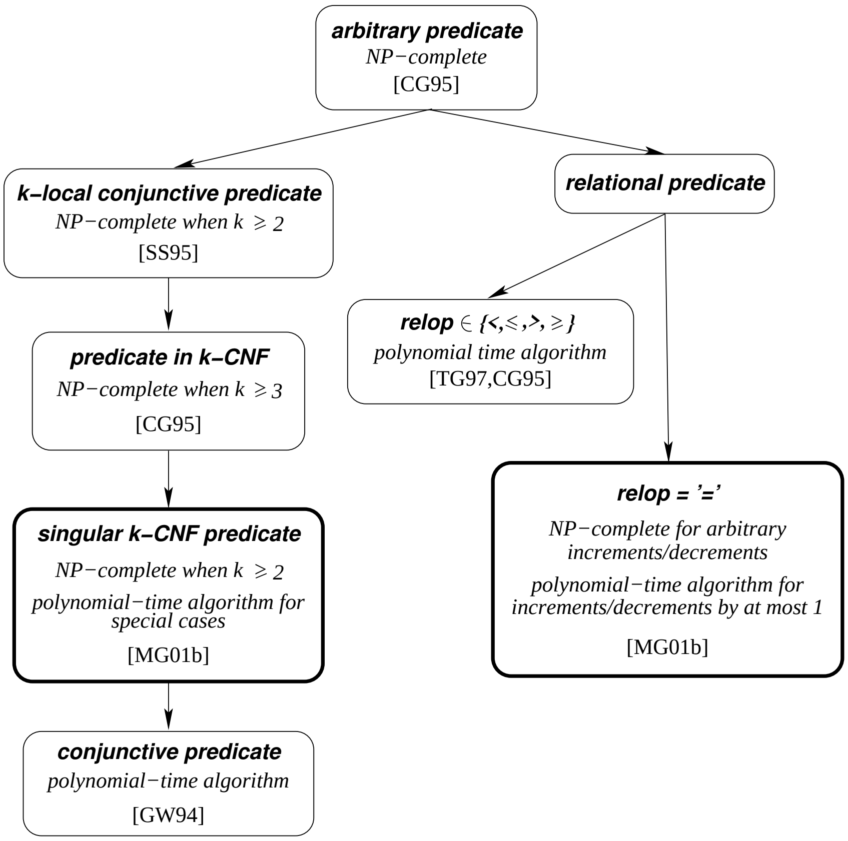


Figure 6.1: Relation of our work to the known results in predicate detection under *possibly* modality.

true only between consecutive snapshots. The second approach [CM91, MN91, AV94, JMN95] enumerates all consistent cuts of the computation to detect a given predicate. Although this approach is able to detect unstable predicates, it has an exponential time-complexity of $O(k^n)$, where k is the number of "relevant" local events on each process and n is the number of processes, making it prohibitively expensive in practice. Partial-order techniques can be employed to avoid examining many consistent cuts during the search [SUL00]. The third approach [MC88, MI92, AV93, HPR93, GW94, BR95, FRGT94, JJJR94, CG95, GCKM95, GC95, GTFR95, TG95, VD95, GW96, HMRS96, BFR96, HR96, GCKM97, TG97] relies on exploiting

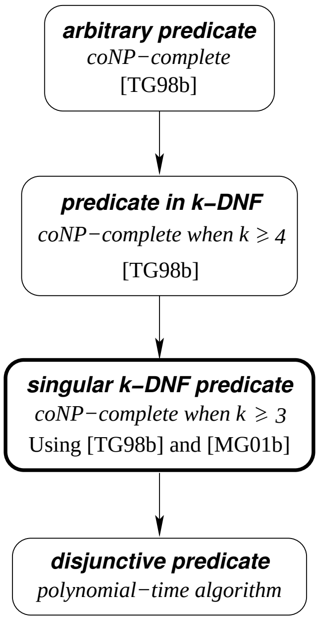


Figure 6.2: Relation of our work to the known results in detecting predicates under *definitely* modality.

the structure of the predicate. Rather than exploring all possible consistent cuts of the computation, the approach uses the computation directly. Thus efficient polynomial-time algorithms can be devised which, however, are restricted to certain special but useful classes of predicates such as conjunctive predicates, observer-independent predicates, linear and post-linear predicates, and relational predicates. Surveys of predicate detection may be found in [BM93, SM94, Gar96]. In this dissertation, our focus has been on the third approach. Within the third approach, we consider predicates that are defined on a single cut.

The predicate detection problem under *possibly* modality has been studied in more setting when events on a process may only be partially ordered [TG98a]. However, detecting even a conjunctive predicate becomes NP-complete in the general model.

Figure 6.1 and Figure 6.2 depict the relation of our work to the known results in detecting predicates under *possibly* and *definitely* modalities, respectively.

6.2 Controlling Global Predicates

The results in predicate control provided in this dissertation were first published in [MG00]. The predicate control problem is known to be NP-hard in general [TG98b]. However, as in the case of predicate detection, by exploiting the structure of the predicate, polynomial-time algorithms have been developed for certain classes of predicates such as "disjunctive predicates" [TG98b] and "mutual exclusion predicates" [TG99]. The latter problem becomes intractable when generalized to "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 [Tar00]. In contrast to Tarafdar and Garg's algorithm [TG98b] for controlling a disjunctive predicate, our algorithm can be modified to generate a *minimum* controlling synchronization.

The study in [MSWW81] allows global properties within the class of conditional elementary restrictions. 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 controlling global predicates off-line with the computation known *a priori*.

An efficient algorithm for detecting a conjunctive predicate under *definitely* modality can be found in [GW96]. Since the problem of detecting a predicate under *definitely* modality is dual of the problem of monitoring a predicate under *controllable* modality, the above algorithm can be used to determine whether a disjunctive predicate is controllable in a computation. However, the predicate detection problem (under *definitely* modality) is not concerned with finding the actual controlling synchronization, if it exists.

6.3 Slicing Distributed Computations

The results in computation slicing described in this dissertation were first published in [GM01, MG01a]. Analogous to the notion of computation slice, it is possible to define the notion of "program slice" [Wei82]. Given a program and a set of variables, a program slice consists of all statements in the program that may affect the value of the variables in the set at some given point. A program slice could be "static" [Wei82] or "dynamic" (for a specific program input) [KR97]. The notion of program slice has been extended to distributed programs as well [KF92]. Intuitively, program slicing can be used to perform data flow analysis in a program (static program slicing) or its trace (dynamic program slicing). The two slicing techniques, namely computation slicing and program slicing, serve different purposes and can, in fact, be used in a complimentary fashion. Computation slicing is useful for carrying out post-mortem analysis of a single execution of a distributed program in order to spot an incorrect behaviour of the program, if any, possibly involving multiple processes, in an automated fashion. On detecting a faulty behaviour, dynamic program slicing can be employed to obtain the relevant subset of the events that actually influenced the value of the variables under observation, using data flow analysis, thereby facilitating the localization of the bug. In other words, program slicing helps to reduce the size of the program that needs to be analyzed for locating the bug *after* a faulty behaviour has been observed using computation slicing or other techniques.

Model checking is an automated technique for verifying the correctness of concurrent programs [CE81, QS82]. Our technique differs from model checking in many aspects. First, model checking is concerned with ascertaining correctness of all computations of a program, whereas we focus on analyzing a single computation.

This is because our objective is to develop *fast* algorithms for the problems that arise in testing and debugging, and software fault tolerance of distributed programs in which a single execution trace of a program is observed. Second, even if model checking algorithms are used on a single computation of a program, their time-complexity would be, in general, proportional to the size of the state-space which is still exponential in the number of processes. Whereas, our emphasis is on developing algorithms that are polynomial in the number of processes. We accomplish this by exploiting the structure of the computation (specifically, the set of consistent cuts of a computation forms a distributive lattice) as well as the structure of the predicate (for example, whether it is a regular predicate or a linear predicate). As a result, model checking techniques, although more general in their applicability, are much more expensive in terms of time and space.

Chapter 7

Conclusions and Future Work

We give a *necessary and sufficient* characterization of the set of consistent cuts of a distributed computation. Specifically, we show that the set of consistent cuts forms a *distributive lattice* and, further, it does not satisfy any additional structural property. We exploit this observation to derive the notion of *computation slice*. Intuitively, computation slicing is useful for throwing away the *extraneous* consistent cuts of a computation, in an efficient manner, and focusing on only those that are currently relevant for our purpose. As an application, it can be used to view a computation at various levels of *atomicity*, thereby providing a flexible and powerful framework for visualization of executions of complex distributed applications [KG95, KBTB97]. Computation slicing can also be used to achieve an exponential improvement in time and space for locating a faulty consistent cut in a computation. The reason for the improvement is that other approaches view the set of consistent cuts simply as a partially ordered set. We, on the other hand, adopt a more aggressive approach and exploit an important structural property of the set of consistent cuts, namely that it forms a distributive lattice.

In this dissertation, we focus on 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. The fault handler uses this information to decide on appropriate corrective actions. In fact, Tarafdar and Garg [Tar00] show that in the case of synchronization faults, by tracing synchronization information during normal execution, one can take more effective corrective action during re-execution. We extend their work in investigating the corresponding off-line synchronization problem—*the predicate control problem*. Specifically, we develop efficient algorithms for solving the predicate control problem for some important classes of predicates.

Currently, our algorithms for computing the slice of a computation are *off-line* and assume that the entire set of events is available to them *a priori*. While this is quite acceptable for debugging, for software fault tolerance, however, it is desirable that the slice be computed in an *on-line and incremental* manner; upon generation of an event in the system, the current slice is updated to accommodate the new event and the resultant slice is checked for an occurrence of a fault. This accelerates the detection of a fault. Also, some applications may execute indefinitely and generate computations that are non-terminating. In order to deal with such computations, mechanism needs to be devised to garbage collect portions of the slice that can no longer contribute to a fault or otherwise will not be needed in the future.

In this dissertation, we give polynomial-time algorithms to monitor a regular predicate under three modalities, namely *possibly*, *invariant* and *controllable*. It still remains an open question whether a regular predicate can be monitored under *definitely* modality in an efficient manner. Note that a conjunctive predicate, which is a special case of regular predicate, can be monitored efficiently under *definitely* modality [GW96]. For the general case, Cooper and Marzullo's algorithm can be used which, however, has exponential time-complexity [CM91]. It is possible to improve the time-complexity of their algorithm using computation slicing as follows. We first compute the slice of the computation with respect to the *complement* of the regular predicate. In case the slice has different number of strongly connected components than the computation, then it can be shown that the regular predicate definitely holds in the computation. Otherwise, rather than applying Cooper and Marzullo's algorithm to the computation, it can now be applied to the slice, which may have much smaller state-space than the computation.

Our focus so far has been on building systems capable of tolerating software faults. A distributed system can fail because of other reasons as well including hardware faults. An interesting question that arises is: "How do we detect software faults *in presence of* hardware failures such as crashes?" A common way to model a crash failure is using the notion of *failure detector* proposed by Chandra and Toueg [CT96, GM98b]. A software fault, on the other hand, is modeled using predicate detection. An interesting research direction is to unify the two models so as to enable software faults to be detected in the presence of crashes. Gärtner and Pleisch prove some impossibility results about predicate detection when processes can fail by crashing [GP01b, GP01a]. Garg and Mitchell present an algorithm to detect predicates that are both *set decreasing* and *conjunctive* using *infinitely often accurate* detector [GM98a]. Can their work be extended to detect other important classes of predicates including regular predicates and relational predicates?

Clearly, it is possible to efficient evaluate whether some consistent cut of a computation satisfies a predicate given an efficient algorithm to compute the slice of the computation with respect to the predicate. Does the converse also hold? Specifically, is it possible to efficiently compute the slice for a predicate given an efficient algorithm to detect the predicate?

Bibliography

[AV93] S. Alagar and S. Venkatesan. Hierarchy in Testing Distributed Programs. In *Proceedings of the International Workshop on Automated Debugging (AADEBUG)*, pages 101–116, 1993.

[AV94] S. Alagar and S. Venkatesan. Techniques to Tackle State Explosion in Global Predicate Detection. In *Proceedings of the The Parallel and Distributed Systems Laboratory*, pages 412–417, December 1994.

[BFR96] Ö. Babaoğlu, E. Fromentin, and M. Raynal. A Unified Framework for the Specification and Run-time Detection of Dynamic Properties in Distributed Computations. *The Journal of Systems and Software*, 33(3):287–298, June 1996.

[BM93] Ö. Babaoğlu and K. Marzullo. Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. In S. Mullender, editor, *Distributed Systems*, pages 55–96. 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.

[BR95] Ö. Babaoğlu and M. Raynal. Specification and Verification of

Dynamic Properties in Distributed Computations. *Journal of Parallel and Distributed Computing*, 28(2):173–185, 1995.

[CE81] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, May 1981.

[CG95] C. Chase and V. K. Garg. On Techniques and their Limitations for the Global Predicate Detection Problem. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 303–317, France, September 1995.

[CG98] C. Chase and V. K. Garg. Detection of Global Predicates: Techniques and their Limitations. *Distributed Computing*, 11(4):191–201, 1998.

[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.

[CLR91] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1991.

[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.

[CT96] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[Fid91] C. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):28–33, August 1991.

[FRGT94] E. Fromentin, M. Raynal, V. K. Garg, and A. I. Tomlinson. On the Fly Testing of Regular Patterns in Distributed Computations. In *Proceedings of the 23rd International Conference on Parallel Processing*, number 2, pages 73–76, Chicago, Illinois, August 1994.

[Gar96] V. K. Garg. Observation of Global Properties in Distributed Systems. In *IEEE International Conference on Software and Knowledge Engineering*, pages 418–425, Lake Tahoe, Nevada, June 1996.

[Gar97] V. K. Garg. Observation and Control for Debugging Distributed Computations. In *Proceedings of the International Workshop on Automated Debugging (AADEBUG)*, pages 1–12, Linköping, Sweden, 1997. Keynote Presentation.

[GC95] V. K. Garg and C. Chase. Distributed Algorithms for Detecting Conjunctive Predicates. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 423–430, Vancouver, Canada, June 1995.

[GCKM95] V. K. Garg, C. Chase, R. Kilgore, and J. R. Mitchell. Detecting Conjunctive Channel Predicates in a Distributed Programming Environment. In *Proceedings of the International Conference on System Sciences*, volume 2, pages 232–241, Maui, Hawaii, January 1995.

[GCKM97] V. K. Garg, C. Chase, R. Kilgore, and J. R. Mitchell. Efficient Detection of Channel Predicates in Distributed Systems. *Journal of Parallel and Distributed Computing*, 45(2):134–147, September 1997.

[GJ91] M. R. Garey and D. S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1991.

[GM98a] V. K. Garg and J. R. Mitchell. Distributed Predicate Detection in a Faulty Environment. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 416–423, Amsterdam, The Netherlands, May 1998.

[GM98b] V. K. Garg and J. R. Mitchell. Implementable Failure Detectors in Asynchronous Systems. In *Proceedings of the Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 158–169, Chennai, India, 1998.

[GM01] V. K. Garg and N. Mittal. On Slicing a Distributed Computation. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 322–329, Phoenix, Arizona, April 2001.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[GP01a] F. C. Gärtner and S. Pleisch. (Im)Possibilities of Predicate Detection in Crash-Affected Systems. In *Proceedings of the 5th Workshop on Self-Stabilizing Systems (WSS)*, Lecture Notes in Computer

Science 2194, pages 98–113, Lisbon, Portugal, October 2001. Springer-Verlag.

[GP01b] F. C. Gärtner and S. Pleisch. (Im)Possibilities of Predicate Detection in Crash-Affected Systems using Interrupt-Style Failure Detectors. In *Proceedings of the Symposium on Distributed Computing (DISC)*, Lisbon, Portugal, October 2001.

[GR93] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[GTFR95] V. K. Garg, A. I. Tomlinson, E. Fromentin, and Michel Raynal. Expressing and Detecting General Control Flow Properties of Distributed Computations. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 432–438, San Antonio, Texas, October 1995.

[GW91] V. K. Garg and B. Waldecker. Detection of Unstable Predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, California, May 1991.

[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.

[HJPR87] J.-M. Helary, C. Jard, N. Plouzeau, and M. Raynal. Detection of Stable Properties in Distributed Applications. In *Proceedings of* *the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 125–136, 1987.

[HMRS96] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient Distributed Detection of Conjunctions of Local Predicates in Asynchronous Computations. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 588–594, New Orleans, October 1996.

[HPR93] M. Hurfin, N. Plouzeau, and M. Raynal. Detecting Atomic Sequences of Predicates in Distributed Computations. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 32–42, 1993.

[HR96] M. Hurfin and M. Raynal. Detecting Diamond Necklaces in Labeled Dags (A Problem from Distributed Debugging. In *Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, Lecture Notes in Computer Science, pages 211–223. Springer-Verlag, 1996.

[JJJR94] C. Jard, T. Jéron, G.-V. Jourdan, and J.-X. Rampon. A General Approach to Trace-Checking in Distributed Computing Systems. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 396–403, 1994.

[JMN95] R. Jegou, R. Medina, and L. Nourine. Linear Space Algorithm for On-line Detection of Global Predicates. In J. Desel, editor, *Proceedings of the International Workshop on Structures in Concurrency Theory (STRICT)*, pages 175–189. Springer-Verlag, 1995.

[JZ88] D. B. Johnson and W. Zwaenepoel. Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 171–181, August 1988.

[KBTB97] T. Kunz, J. P. Black, D. J. Taylor, and T. Basten. POET: Target-System Independent Visualizations of Complex Distributed-Applications Executions. *The Computer Journal*, 40(8), 1997.

[KF92] B. Korel and R. Ferguson. Dynamic Slicing of Distributed Programs. *Applied Mathematics and Computer Science Journal*, 2(2):199–215, 1992.

[KG95] J. A. Kohl and G. A. Geist. The PVM 3.4 Tracing Facility and XPVM 1.1. Computer Science and Mathematics Divison, Oak Ridge National Laboratory, TN, USA, 1995.

[Koh78] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, 2nd edition, 1978.

[KR97] B. Korel and J. Rilling. Application of Dynamic Slicing in Program Debugging. In Mariam Kamkar, editor, *Proceedings of the 3rd International Workshop on Automated Debugging (AADEBUG)*, pages 43–57, Linköping, Sweden, May 1997.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[Mat89] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: Proceedings of the Workshop* *on Distributed Algorithms (WDAG)*, pages 215–226. Elsevier Science Publishers B. V. (North-Holland), 1989.

[MC88] B. P. Miller and J. Choi. Breakpoints and Halting in Distributed Programs. In *Proceedings of the 8th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 316–323, 1988.

[MG00] N. Mittal and V. K. Garg. Debugging Distributed Programs Using Controlled Re-execution. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 239–248, Portland, Oregon, July 2000.

[MG01a] N. Mittal and V. K. Garg. Computation Slicing: Techniques and Theory. In *Proceedings of the Symposium on Distributed Computing (DISC)*, pages 78–92, Lisbon, Portugal, October 2001.

[MG01b] N. Mittal and V. K. Garg. On Detecting Global Predicates in Distributed Computations. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, Phoenix, Arizona, April 2001.

[MI92] Y. Manabe and M. Imase. Global Conditions in Debugging Distributed Programs. *Journal of Parallel and Distributed Computing*, 15(1):62–69, 1992.

[MN91] K. Marzullo and G. Neiger. Detection of Global State Predicates. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 254–272, 1991.

[MSWW81] A. Maggiolo-Schettini, H. Welde, and J. Winkowski. Modeling a Solution for a Control Problem in Distributed Systems by

Restrictions. *Theoretical Computer Science*, 13(1):61–83, January 1981.

[NX95] R. H. B. Netzer and J. Xu. Necessary and Sufficient Conditions for Consistent Global Snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, February 1995.

[QS82] J. P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *Proceedings of the 5th International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, New York, 1982. Springer-Verlag.

[Ray88] M. Raynal. *Distributed Algorithms and Protocols.* John Wiley and Sons Limited, 1988.

[Riv74] I. Rival. Maximal Sublattices of Finite Distributive Lattices II. *Proceedings of the American Mathematical Society*, 44(2):263–268, 1974.

[SK86] M. Spezialetti and P. Kearns. Efficient Distributed Snapshots. In *Proceedings of the 6th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 382–388, 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.

[SS95] S. D. Stoller and F. Schneider. Faster Possibility Detection by Combining Two Approaches. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, volume 972 of *Lecture Notes in Computer Science*, pages 318–332, France, September 1995.

[SUL00] S. D. Stoller, L. Unnikrishnan, and Y. A. Liu. Efficient Detection of Global Properties in Distributed Systems Using Partial-Order Methods. In *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 264–279. Springer-Verlag, July 2000.

[Tar00] A. Tarafdar. *Software Fault Tolerance in Distributed Systems Using Controlled Re-execution*. PhD thesis, The University of Texas at Austin, August 2000.

[TG94] A. I. Tomlinson and V. K. Garg. Maintaining Global Assertions in Distributed Systems. In *Computer Science 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 *Proceedings of the 15th Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 195–209, 1995.

[TG97] A. I. Tomlinson and V. K. Garg. Monitoring Functions on Global States of Distributed Programs. *Journal of Parallel and Distributed Computing*, 41(2):173–189, March 1997.

[TG98a] A. Tarafdar and V. K. Garg. Addressing False Causality while Detecting Predicates in Distributed Programs. In *Proceedings of the 9th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 94–101, Amsterdam, The Netherlands, May 1998.

[TG98b] A. Tarafdar and V. K. Garg. Predicate Control for Active Debugging of Distributed Programs. In *Proceedings of the 9th IEEE Symposium* *on Parallel and Distributed Processing (SPDP)*, pages 763–769, Orlando, 1998.

[TG99] A. Tarafdar and V. K. Garg. Software Fault Tolerance of Concurrent Programs Using Controlled Re-execution. In *Proceedings of the 13th Symposium on Distributed Computing (DISC)*, pages 210–224, Bratislava, Slovak Republic, September 1999.

[TP00] W. Torres-Pomales. Software Fault Tolerance: A Tutorial, 2000. NASA Langley Research Center.

[VD95] S. Venkatesan and B. Dathan. Testing and Debugging Distributed Programs Using Global Predicates. *IEEE Transactions on Software Engineering*, 21(2):163–177, February 1995.

[Wan97] Y.-M. Wang. Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints. *IEEE Transactions on Computers*, 46(4):456–468, April 1997.

[Wei82] M. Weiser. Programmers Use Slices when Debugging. *Communications of the ACM (CACM)*, 25(7):446–452, 1982.

[WG91] B. Waldecker and V. K. Garg. Unstable Predicate Detection in Distributed Program Debugging. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 276–278, Santa Cruz, California, May 1991.

[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 Transactions on Computers*, 46(10):1137–1141, October 1997.

Vita

Neeraj Mittal was born on May 07, 1974 in New Delhi, India, the son of Shankuntla Mittal and Suresh Chandra Mittal. He completed his schooling in New Delhi, India in June 1991. He received the Bachelor of Technology degree in Computer Science and Engineering from the Indian Institute of Technology at Delhi in July 1995. Thereafter, he joined graduate school at the University of Texas at Austin where he received the Master of Sciences degree in Computer Sciences in May 1997. He has been employed at Lucent Bell Laboratories and IBM Almaden Research Center during the summers of 1997 and 1999, respectively. He was awarded the MCD graduate fellowship in August 1995 by the University of Texas at Austin.

Permanent Address: c/o Mr. Suresh Chandra Mittal
A-2/263 Janakpuri
New Delhi 110058, 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.