Finding Satisfying Global States: All for One and One for All

Neeraj Mittal
Dept. of Computer Science
The University of Texas at Dallas
Richardson, TX 75083-0688, USA
neerajm@utdallas.edu

Alper Sen   Vijay K. Garg*
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084, USA
{sen, garg}@ece.utexas.edu

Abstract

Given a distributed computation and a global predicate, predicate detection is concerned with determining whether there exists *at least one* consistent cut (or global state) of the computation that satisfies the predicate. On the other hand, computation slicing involves computing the smallest subcomputation—with the least number of consistent cuts—that contains *all* consistent cuts of the computation satisfying the predicate. In this paper we study the relationship between the two problems. Specifically, we show that the two problems are actually equivalent. That is, given an algorithm A that solves the predicate detection problem, we can derive an algorithm B to compute the slice, where the time complexity of B is at most $n|E|$ times the time-complexity of A, where n is the number of processes and E is the set of events.

Predicate detection is a fundamental problem in distributed systems with applications in testing and debugging, and software fault tolerance of distributed programs. However, the problem is intractable in general [7]. In our earlier papers [8, 12, 14], we propose the technique of computation slicing, which can be used to ameliorate the state-space explosion problem in predicate detection. We discuss how results in this paper can be utilized to derive a much more efficient algorithm for predicate detection.

Finally, the slicing algorithms described in our earlier papers [8, 12, 14] are all *off-line* in nature. In this paper, we give an efficient *on-line* algorithm algorithm for computing a slice for the special class of *regular predicates* [8].

Keywords: analyzing distributed computations, computation slicing, predicate detection, global property evaluation, testing and debugging, software fault tolerance, on-line algorithm

ELIGIBLE FOR BEST STUDENT PAPER AWARD

1 Introduction

Writing correct distributed programs is a non-trivial task. Not surprisingly, distributed systems are particularly vulnerable to software faults. Testing and debugging is an effective way of improving the dependability of a software prior to its deployment. Software bugs that do persist after extensive testing and debugging have to be tolerated at runtime to ensure that the system continues to operate properly. Detecting a fault in an execution of a distributed system (*e.g.*, violation of mutual exclusion) is a fundamental problem that arises during testing and debugging as well as software fault tolerance.

In this paper, we focus on detecting those faults that can be expressed as predicates on variables of processes. For example, "no process has the token" can be written as $no\_token_1 \wedge no\_token_2 \wedge \cdots \wedge no\_token_n$, where no_token_i denotes the absence of token on process p_i. This gives rise to

*supported in part by the NSF Grants ECS-9907213, CCR-9988225, Texas Education Board Grant ARP-320, an Engineering Foundation Fellowship, and an IBM grant.

the *predicate detection problem* which involves determining whether there exists a consistent cut of a distributed computation that satisfies given predicate. (This problem is also referred to as detecting a predicate under *possibly* modality in the literature.)

Detecting a predicate in a distributed computation is a hard problem in general [15, 7]. The reason is the combinatorial explosion in the number of possible consistent cuts. Given n processes each with at most k local states, the number of possible consistent cuts in the computation could be as large as $O(k^n)$. Finding a consistent cut that satisfies the given predicate may, therefore, require looking at a large number of consistent cuts.

The approaches for solving the predicate detection problem can be divided into three categories. The first approach involves repeatedly computing global snapshots of the computation until the given predicate becomes true [2]. This approach works only for stable predicates, that is, predicates that stay true once they become true. Some examples of stable predicates are termination and deadlock. The given predicate may not be stable and may turn true only between two successive snapshots. The second approach is based on searching the state-space of the computation. This approach involves incrementally building the lattice corresponding to the computation until the desired predicate turns true [5, 10, 16, 1]. Unlike the first approach, this approach can be used to detect unstable predicates. However, the algorithms based on this approach may have exponential running time. The third approach exploits the structure of the predicate itself—by imposing restrictions—to evaluate its value efficiently for a given computation. Polynomial-time algorithms have been developed for several useful classes of predicates including conjunctive predicates [9, 7], linear and semi-linear predicates [7], and relational predicates [4].

In our earlier papers [8, 12], we introduce the notion of *computation slice*. Intuitively, slice is a concise representation of consistent cuts satisfying a certain condition. The slice of a computation with respect to a predicate is the directed graph with the least number of consistent cuts that contains all consistent cuts of the computation for which the predicate evaluates to true. 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. The number of consistent cuts of the slice is much smaller than those of the computation. Therefore, in order to detect a fault, rather than searching the state-space of the computation, it is much more efficient to search the state-space of the slice. We also identify a class of predicates, called *regular predicates*, for which the slice is *lean* [8, 12]. That is, the slice for a regular predicate contains precisely those consistent cuts for which the predicate evaluates to true.

As an illustration, suppose we want to detect the predicate $(x_1 * x_2 + x_3 < 5) \wedge (x_1 \geqslant 1) \wedge (x_3 \leqslant 3)$ in the computation shown in Figure 1(a). The computation consists of three processes p_1, p_2 and p_3 hosting integer variables x_1, x_2 and x_3, respectively. The events are represented by circles. Each event is labeled with the value of the respective variable immediately after the event is executed. For example, the value of variable x_1 immediately after executing the event c is -1. The first event on each process initializes the state of the process and every consistent cut contains these initial events. Without computation slicing, we are forced to examine all consistent cuts of the computation, twenty eight in total, to ascertain whether some consistent cut satisfies the predicate. Alternatively, we can compute the slice of the computation with respect to the predicate $(x_1 \geqslant 1) \wedge (x_3 \leqslant 3)$ as portrayed in Figure 1(b). The slice is modeled by a directed graph. Each vertex of the graph corresponds to a *meta-event*, that is, a subset of events. If a vertex is contained in a consistent cut, the interpretation is that all events corresponding to the meta-event are contained in the cut. Moreover, a meta-event belongs to a consistent cut only if all its incoming neighbours are also present in the cut. We can now restrict our search to the consistent cuts of the slice which are only six in number, namely $\{a, e, f, u, v\}$, $\{a, e, f, u, v, b\}$, $\{a, e, f, u, v, w\}$, $\{a, e, f, u, v, b, w\}$, $\{a, e, f, u, v, w, g\}$ and $\{a, e, f, u, v, b, w, g\}$. The slice has much fewer consistent cuts than the computation itself—exponentially smaller in many cases—resulting in substantial

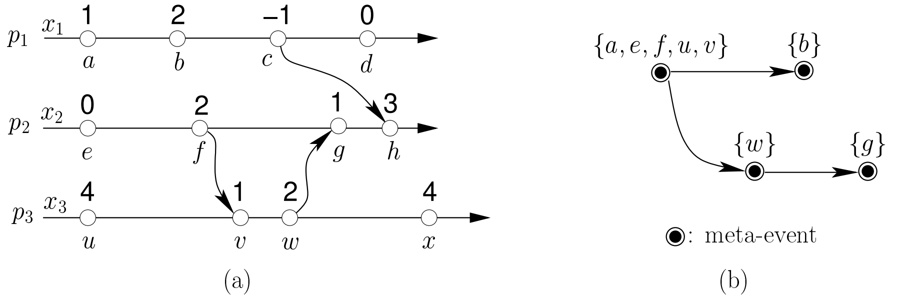

Figure 1: (a) A computation and (b) its slice with respect to $(x_1 \geqslant 1) \wedge (x_3 \leqslant 3)$.

savings.

The predicate detection problem is only concerned with answering the question whether there exists *at least one* consistent cut of the computation that satisfies the given predicate. Computation slicing, on the other hand, is concerned with computing (a succinct representation of) *all* consistent cuts of the computation for which the given predicate evaluates to true. Clearly, detecting a predicate is no harder than computing its slice in the sense that the predicate detection problem can be easily solved given the slice for the predicate (it suffices to test for the emptiness of the slice). In this paper, we prove a somewhat surprising result that detecting a predicate is no easier than computing its slice. In other words, given an algorithm A for detecting a predicate b, there exists an algorithm B for computing the slice for b such that the time-complexity of B is at most a small multiple of the time-complexity of A. In particular, the multiple is $n|E|$, where n is the number of processes and E is the set of events. As a corollary, it can be derived that there exists a polynomial-time algorithm for detecting a predicate if and only if there exists a polynomial-time algorithm for computing its slice.

At first glance, it may seem that we are not any better off than we were before. After all, if predicate detection is "equivalent" to computation slicing, then how can slicing be used to improve the complexity of predicate detection? Slicing can indeed be used to facilitate predicate detection as illustrated by the following example. Consider a predicate b that is a conjunction of two clauses b_1 and b_2. Now, assume that b_1 is such that it can be detected efficiently but b_2 has no structural property that can be exploited for efficient detection. To detect b, without computation slicing, we are forced to use techniques such as *breadth first search* [5], *depth first search* [1], and *partial-order methods* (a technique for model-checking) [16], which do not take advantage of the fact that b_1 can be detected efficiently. With computation slicing, however, we can first compute the slice for b_1. If only a small fraction of consistent cuts satisfy b_1, then instead of detecting b in the computation, it is much more efficient detect b in the slice. Therefore by spending only polynomial amount of time in computing the slice we can throw away exponential number of consistent cuts, thereby obtaining an exponential speedup overall. Consequently, the result of this paper, rather than diminishing the benefits of computation slicing, actually enhances them by greatly expanding the class of predicates for which the slice can be computed efficiently.

Note that other techniques for reducing the time-complexity [16] and/or the space-complexity [1] of predicate detection are orthogonal to slicing, and as such can actually be used in conjunction with slicing. Although, in this paper, we focus on application of slicing to predicate detection, slicing can be employed to reduce the search-space for detecting a predicate under other modalities as well including *definitely*, *invariant* and *controllable* [5, 12, 7] and their nestings, which generates a subset of temporal logic [14].

We consider another problem related to predicate detection, namely the problem of locating a consistent cut of a computation that satisfies the given predicate, if it exists. While it is sufficient to determine whether there exists a faulty consistent cut in a computation for testing purposes, for debugging purposes, it is desirable to actually *locate* the faulty consistent cut. An examination of such a cut may provide valuable insight into the bug that caused the fault. This problem is closely related to the problem of setting a *global breakpoint* when debugging a distributed program.

Finally, the algorithms described in our earlier papers [8, 12, 14] for computing a slice are all *off-line* in nature; they assume that the entire set of events is available *a priori*. While this is quite adequate for applications such as testing and debugging, for other applications such as software fault tolerance, it is desirable that the slice be computed incrementally in an *on-line* manner; as and when a new event is generated, the current slice is updated to reflect its arrival. The reason is that for software fault tolerance, it is important to detect the fault as early as possible before it can cause any severe damage. If we compute the slice only after a certain number of events have been collected and then analyze it for the presence of a faulty consistent cut, it may be too late for any meaningful recovery. At the same, whenever an event arrives, we want the cost of incrementally updating the slice to be less than the cost of recomputing the slice from scratch using an off-line algorithm. In this paper, we give an efficient incremental algorithm for computing the slice for the class of regular predicates [8].

To summarize, in this paper, we establish the equivalence of three important problems in distributed systems. The first problem of predicate detection is concerned with determining whether there exists a consistent cut of a computation that satisfies the given predicate. The second problem of computation slicing is concerned with computing the smallest graph that contains all consistent cuts of a computation for which the given predicate evaluates to true. The third problem, which is related to the problem of setting a global breakpoint, involves locating a consistent cut of a computation that satisfies the given predicate, if it exists. We also present an efficient algorithm for computing the slice for a regular predicate in an incremental manner. Due to the lack of space, the proofs of all lemmas and theorems have been moved to the appendix.

The paper is organized as follows. Section 2 describes our system model and the notation we use in this paper. In Section 3, we formally define the notion of computation slice. We establish the equivalence of the three problems in Section 4. We also discuss the implication of the equivalence result to predicate detection. Section 5 describes the on-line slicing algorithm for a regular predicate. Finally, Section 6 concludes the paper.

2 Model and Notation

Traditionally, a distributed computation is modeled as a partial order on a set of events [11]. In this paper 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 strongly connected component or none of them. 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. Also, let $\mathcal{P}(G)$ denote the set of pairs of vertices (u, v) such that there is a path from u to v in G. We assume that every vertex has a path to itself.

A *distributed computation* (or simply a *computation*) $\langle E, \rightarrow \rangle$ is 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 [11]. A distributed computation in our model can contain cycles. This is because whereas a computation in the traditional or happened-before model captures the *observable* order of execution of events, a computation in our model captures the set of possible consistent cuts. Intuitively, each strongly connected component of a computation can be viewed as a *meta-event*.

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. When two events e and f occurred on the same process and e occurred before f in real-time, then we write $e \stackrel{P}{\rightarrow} f$, and let $\stackrel{P}{\underline{\rightarrow}}$ be the reflexive closure of $\stackrel{P}{\rightarrow}$. We assume the presence of fictitious initial and final events on each process. The initial event on process p_i, denoted by $\bot_i$, occurs before any other event on p_i. Likewise, the final event on process p_i, denoted by $\top_i$, occurs after all other events on p_i. For convenience, let $\bot$ and $\top$ denote the set of all initial events and final events, respectively. 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 our model and vice versa. Only non-trivial consistent cuts are of real interest to us. The *frontier* of a cut C, denoted by $frontier(C)$, is defined as the set of those events in C whose successors are not in C. Formally, $frontier(C) \;\triangleq\; \{\, e \in C \mid e \notin \top \;\Rightarrow\; succ(e) \notin C \,\}$.

A *global predicate* (or simply a *predicate*) is 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". We leave the predicate undefined for the trivial consistent cuts. A global predicate is *local* if it depends on variables of a single process.

For an illustration of various concepts, please refer to Example 1 in Appendix A.

3 Computation Slice

The notion of computation slice is based on the Birkhoff's Representation Theorem for Finite Distributive Lattices [6]. The readers who are not familiar with our earlier papers on slicing [8, 12, 14] are strongly urged to read Appendix A. Roughly speaking, a computation slice (or simply a slice) is a concise representation of all those consistent cuts of the computation that satisfy the predicate. More precisely,

Definition 1 (slice [12]) *A* slice *of a computation with respect to a predicate is a 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 denote the slice of a computation $\langle E, \rightarrow\rangle$ with respect to a predicate b by $\mathsf{slice}(\langle E, \rightarrow\rangle, b)$. Note that $\langle E, \rightarrow\rangle = \mathsf{slice}(\langle E, \rightarrow\rangle, \mathsf{true})$. We prove in [12] that the slice exists and is uniquely defined for all predicates. Every slice derived from the computation $\langle E, \rightarrow\rangle$ has the trivial consistent cuts ($\emptyset$ and E) among its set of consistent cuts. A slice is *empty* if it has no non-trivial consistent cuts [12]. In the rest of the paper, unless otherwise stated, a consistent cut refers to a non-trivial consistent cut. In this paper, we use the terms "computation", "slice" and "directed graph" interchangeably.

4 The Three Problems

In this section, we study the relationship among the following three problems in distributed systems.

Containing Cut (CONTC) Given a directed graph G and a predicate b, is there a consistent cut of G that satisfies b?

Computing Cut (COMPC) Given a directed graph G and a predicate b, compute a consistent cut of G that satisfies b, if any.

Computing Slice (COMPS) Given a directed graph G and a predicate b, compute the slice of G with respect to b.

4.1 The Main Result: COMPS $\cong$ CONTC

In this section, we prove that the problem of computing a succinct representation of *all* consistent cuts satisfying a predicate is equivalent the problem of determining whether there exists *at least one* consistent cut satisfying the predicate. From the definition of slice, clearly, it follows that the slice for a directed graph with respect to a predicate (not necessarily regular [8]) is non-empty if and only if the graph contains a consistent cut that satisfies the predicate. Formally,

$$\mathsf{CONTC}(G,b) \quad \equiv \quad \mathsf{COMPS}(G,b) \text{ is non-empty}$$

Therefore COMPS is at least as hard as CONTC. We now prove the converse. Consider a directed graph G and a predicate b. Now, G and $\mathsf{slice}(G,b)$ are directed graphs on identical sets of vertices. However, more pairs of vertices are "connected" in $\mathsf{slice}(G,b)$ than in G. In the next lemma, we give a complete characterization of the pairs of vertices that are "connected" in $\mathsf{slice}(G,b)$. Let $G[e,f]$ denote the directed graph obtained by adding an edge from e to f in G.

Lemma 1 *There is a path from an event e to an event f in $\mathsf{slice}(G,b)$ if and only if no consistent cut in $\mathcal{C}(G) \setminus \mathcal{C}(G[e,f])$ satisfies b.*

Lemma 1 is useful provided it is possible to ascertain efficiently whether some consistent cut in $\mathcal{C}(G) \setminus \mathcal{C}(G[e,f])$ satisfies b. To that end, we show that the set $\mathcal{C}(G) \setminus \mathcal{C}(G[e,f])$ actually forms a sublattice and therefore can be captured faithfully using a directed graph. Let $\widehat{G}[e,f]$ denote the directed graph obtained by adding an edge from f to $\bot_1$ and an edge from $\top_1$ to e. It suffices to show the following:

Lemma 2 $\mathcal{C}(\widehat{G}[e,f]) \setminus \{\emptyset, E\} \quad = \quad \mathcal{C}(G) \setminus \mathcal{C}(G[e,f])$

An example illustrating the Lemma 1 and Lemma 2 can be found in Appendix C. Combining the two lemmas, we obtain the following:

Theorem 3 *There is a path from an event e to an event f in $\mathsf{slice}(G,b)$ if and only if no consistent cut in $\widehat{G}[e,f]$ satisfies b, that is, $\mathsf{CONTC}(\widehat{G}[e,f],b)$ evaluates to false.*

Figure 2 depicts the algorithm for solving COMPS using an algorithm that solves CONTC. The algorithm constructs a directed graph that is transitively closed.

Theorem 4 *The time-complexity of the algorithm for solving COMPS in Figure 2 is $O(|E|^2T)$, where E is the set of events and $O(T)$ is the worst-case time-complexity of solving CONTC.*

The time-complexity of the algorithm can be reduced to $O(n|E|T)$, where n is the number of processes, by constructing the skeletal representation of the slice discussed in Section A.3 in the appendix. To that end, it suffices to compute the vector $F_b(e)$ for each event e; the i^{th} entry of $F_b(e)$ denotes the earliest event on process p_i reachable from e in the slice. It is easy to verify that F_b is order-preserving which means that if $e \rightarrow f$ then $F_b(e)[j] \xrightarrow{P} F_b(f)[j]$ for each j. Consequently, it is possible to compute $F_b(e)[j]$ for each event e on process p_i by scanning the computation once from left to right. The algorithm is presented in Figure 3.

Input: (1) a directed graph G, (2) a predicate b, and
(3) an algorithm to evaluate CONTC(H, b) for an arbitrary directed graph H

Output: the slice of G with respect to b

$K := G$;
for every pair of events (e, f) do
    if not(CONTC$(\widehat{G}[e, f], b)$) then
        add an edge from e to f in K; // $K := K[e, f]$
    endif;
endfor;
output K;

Figure 2: An algorithm to solve COMPS using an algorithm to solve CONTC.

Input: (1) a directed graph G, (2) a predicate b, (3) a process p_i, and
(4) an algorithm to evaluate CONTC(H, b) for an arbitrary directed graph H

Output: $F_b(e)$ 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* $\xrightarrow{P}$
        while CONTC$(\widehat{G}[e, f], b)$ do $f := succ(f)$; endwhile;
        $F_b(e)[j] := f$;
    endfor;
endfor;

Figure 3: The algorithm to compute $F_b(e)$ for each event e on process p_i.

Theorem 5 *The time-complexity of the algorithm for computing $F_b(e)$ for each event e in Figure 3 is $O(n|E|T)$, where n is the number of processes, E is the set of events and $O(T)$ is the worst-case time-complexity of solving* CONTC.

4.2 CONTC $\cong$ COMPC

In this section, we establish that the problem COMPC is equivalent to the problem CONTC. While for testing purposes, it is sufficient determine whether there exists a faulty consistent cut in a computation, for debugging purposes, it is desirable to actually *locate* the faulty consistent cut. This is because an examination of such a cut may provide valuable insight into the bug that caused the fault. Evidently, COMPC is at least as hard as CONTC. We prove the converse. The main idea is follows. Starting with the lattice of consistent cuts of the given directed graph, successively shrink the lattice—by adding edges to the graph—until it consists of only a single non-trivial consistent cut. Of course, at each step, an edge is added in such a way that the resultant (shrunken) lattice contains at least one consistent cut that satisfies the predicate. The algorithm is presented in Figure 4. The next theorem establishes the correctness of the algorithm.

Theorem 6 *The algorithm in Figure 4 solves* COMPC.

An illustration of the algorithm is given in Appendix D. In general, a graph may contain more than one consistent cut satisfying the given predicate. The specific consistent cut that is output at line 7 depends on the order in which the events are considered at line 3.

Input: (1) a directed graph G, (2) a predicate b, and
(3) an algorithm to evaluate CONTC(H, b) for an arbitrary directed graph H

Output: a consistent cut of G that satisfies b, if any

if not(CONTC(G, b)) then output "no consistent cut of G satisfies b" endif;
$K := G$;
for each event e do
  if CONTC$(K[e, \bot_1], b)$ then *// $K[e, f]$ is the directed graph obtained*
    // by adding an edge from e to f in K
    add an edge from e to $\bot_1$ in K; *// $K := K[e, \bot_1]$*
  else add an edge from $\top_1$ to e in K; *// $K := K[\top_1, e]$*
  endif;
endfor;

// at this point, K has exactly one non-trivial consistent cut
output the non-trivial consistent cut of K;

Figure 4: An algorithm to solve COMPC using an algorithm to solve CONTC.

Theorem 7 *The time-complexity of the algorithm for solving* COMPC *in Figure 4 is $O(|E|T)$, where E is the set of events and $O(T)$ is the worst-case time-complexity of solving* CONTC.

It is possible to further reduce the time-complexity to $O(n \log(|E|)T)$ using a technique similar to binary search. It is discussed in Appendix E.

4.3 Applications of the Result

Predicate detection is an important problem in distributed systems. Efficient detection algorithms have been developed for several useful classes of predicates. Examples include stable predicates [2], observer-independent predicates [3], conjunctive predicates [9, 7], linear predicates [7], relational predicates [4], k-local predicates [15], and their complements such as co-stable predicates, and co-linear predicates [14]. In our earlier papers, we give efficient algorithms for computing the slice for regular predicates [8], co-regular predicates [12], linear predicates and k-local predicates for constant k [13]. Using the result of this paper, it is now possible to compute the slice efficiently for many more classes of predicates including stable and co-stable predicates, observer independent predicates, co-linear predicates, and relational predicates. For instance, an observer independent predicate can be detected in $O(n|E|)$ time using the algorithm presented in [3]. This implies that its slice can be computed in $O(n^2|E|^2)$ time using the algorithm given in Section 4.1. It is possible that a faster and more efficient slicing algorithm exists for an observer-independent predicate, which perhaps exploits the specific properties of the class of observer-independent predicates. Our result is still useful because it gives a ready-made algorithm for computing the slice.

Since the predicate detection problem is NP-complete in general [15, 7], the problem of computing the slice is also NP-complete. Nonetheless, it is still useful to be able to compute an *approximate slice* for such a predicate efficiently [12, 13]. An approximate slice may be bigger than the actual slice but may be much smaller than the computation itself. Therefore, in order to detect a predicate, rather than searching the state-space of the computation, it is much more efficient to search the state-space of the slice. Using our algorithms for composing two slices given in [12], it is now possible to compute an approximate slice in polynomial-time for a predicate composed from the predicates mentioned in the previous paragraph using $\wedge$ and $\vee$ operators. This is significant because our experimental results indicate that slicing can lead to an exponential improvement over existing techniques for predicate detection in terms of time and space [13].

5 On-Line Slicing Algorithm for Regular Predicate

In this section we give an incremental algorithm for computing the slice for the class of regular predicates. A predicate is said to be *regular* if, given two consistent cuts satisfying the predicate, the cuts given by their set intersection and set union also satisfy the predicate [8]. Some examples of regular predicates are: *conjunctive predicates*, which can be expressed as conjunction of local predicates, like "all processes are in red state" [7], and *monotonic channel predicates* such as "all control messages have been received" [7]. The class of regular predicates is closed under conjunction.

Now, consider a computation $\langle E, \rightarrow \rangle$ and a regular predicate b. For an event e, we define $J_b(e)$ as the *least consistent cut* of $\langle E, \rightarrow \rangle$ that contains e and satisfies b [8]. In case no consistent cut containing e that also satisfies b exists or when $e \in \top$, $J_b(e)$ is set to E. We use the non-empty trivial consistent cut E as a *sentinel* cut, and, hereafter in this section would refer to it as the *default cut*. We show in [8] that it is sufficient to know $J_b(e)$ for each event e in order to compute the slice of $\langle E, \rightarrow \rangle$ with respect to b. We also give an $O(n^2|E|)$ off-line algorithm to compute $J_b(e)$ for each event e. Our on-line algorithm for computing $J_b(e)$ for each event e is a modification of the off-line version, which we briefly describe next.

Consider the predicate b_e defined as follows: b_e evaluates to true for C if C satisfies b and contains e. Clearly, $J_b(e)$ can be reinterpreted as the *least consistent cut* of $\langle E, \rightarrow \rangle$ for which b_e evaluates to true. To compute the least consistent cut satisfying b_e, we scan the computation from left to right. Starting from the initial consistent cut $\perp$, we move forward one event at a time until either we reach a consistent cut that satisfies b_e or we run out of events. At each step, in order to determine which event to move to next, we observe that b_e is a *linear predicate* and therefore satisfies the *linearity property* [7]. Suppose we are currently at a consistent cut that does not satisfy b_e. Then, the linearity property states that there exists a process such that if we do not advance along that process, then we can never "reach" a consistent cut satisfying b_e, if it exists. The event in the frontier of the cut on that process is called a *forbidden event* [7]. In most cases, it is possible to determine a forbidden event in $O(n)$ time. This gives an $O(n|E|)$ algorithm to compute $J_b(e)$ for a *given event* e. We also prove in [8] that J_b—which can be viewed as a function mapping an event to a consistent cut—is order-preserving: if $e \rightarrow f$ then $J_b(e) \subseteq J_b(f)$. This implies that when computing $J_b(succ(e))$, rather that starting the scan from $\perp$, we can start the scan from $J_b(e)$. As a result, it is possible to compute $J_b(e)$ for each event e on a *given process* in a single scan of the computation in $O(n|E|)$ time.

Next, we modify the above algorithm to compute $J_b(e)$ for each event e in an incremental manner. To that end, we categorize events into two: those for which J_b evaluates to a non-trivial consistent cut and those for which J_b evaluates to the default cut. Note that for a non-final event e, $J_b(e)$ is the default cut if and only if no consistent cut of the computation satisfies b_e. We show that, given an event e, once $J_b(e)$ evaluates to a non-trivial consistent cut, it never changes again. Let the k^{th} arriving event be denoted by $e^{(k)}$, and let $\langle E^{(k)}, \rightarrow \rangle$ denote the resulting computation. We assume that $E^{(0)} = \perp \cup \top$. We use $J_b^{(k)}(e)$ to refer to the least consistent cut of $\langle E^{(k)}, \rightarrow \rangle$ that contains e and satisfies b.

Theorem 8 *For each event $e \in E^{(k)}$,*

$$J_b^{(k)}(e) \textit{ is a non-trivial consistent cut} \quad \Rightarrow \quad J_b^{(k+1)}(e) = J_b^{(k)}(e)$$

Theorem 8 is significant because it may greatly restrict the set of events for which J_b has to be recomputed. Specifically, J_b has to be updated for only those events for which it currently evaluates to the default cut. This is because if $J_b(e)$ for an event e is currently set to the default cut, it implies that the current computation does not contain any consistent cut that contains e and satisfies b. However, the arrival of a new event may result in the generation of such consistent cuts in the new computation.

To be able to efficiently determine for which events J_b may have to be updated, for each process p_i, we maintain a variable $recompute_i$. Intuitively, $recompute_i$ keeps track of the *earliest event* on process p_i for which J_b currently evaluates to the default cut. The order-preserving property of J_b guarantees that for every event e on p_i before $recompute_i$, the current value of $J_b(e)$ is a non-trivial consistent cut, and for every event e on p_i after $recompute_i$, the current value of $J_b(e)$ is the default cut. Additionally, we maintain another variable $current_i$ that keeps track of how far we have advanced in our scan of the computation when computing $J_b(recompute_i)$. Our on-line algorithm for computing J_b basically emulates the off-line version. Therefore we terminate the scan as soon as we realize that the next step involves advancing the scan to include a final event. The reason is as follows. Suppose the next step in the scan involves moving beyond event e on process p_i to the final event $\top_i$. It is possible that more events may be executed on p_i after e in the future. As a result, an off-line algorithm, which will have at its disposal all these events, will never make the corresponding move. Initially, $recompute_i$ is $\bot_i$ and $current_i$ is $\bot$.

The implementation and other details of the algorithm, referred to as RecomputeJ, can be found in Figure 10 in Appendix F. The correctness of the algorithm follows from the fact that the on-line algorithm moves to an event during the scan only if the off-line algorithm will also make the same move. Although the worst-case time-complexity of invoking the algorithm RecomputeJ on arrival of the event $e^{(k)}$ is $O(n^2|E^{(k)}|)$, it can be shown that its amortized time-complexity is much less.

Theorem 9 *The amortized time-complexity of the algorithm* RecomputeJ *is $O(n^2)$ where n is the number of processes.*

For a set of events E, the slice constructed using J_b can have $O(|E|^2)$ edges. Therefore we construct the skeletal representation of the slice discussed in Section A.3 in the appendix, which has only $O(n|E|)$ edges. To accomplish that, we have to compute $F_b(e)$ for each event e. Let $F_b^{(k)}(e)$ denote the value of $F_b(e)$ in the computation $\langle E^{(k)}, \rightarrow \rangle$. It is easy to verify that:

Lemma 10 *For every event $e \in E^{(k)}$ and each process p_i, $F_b^{(k)}(e)[i] \overset{P}{\underline{\rightarrow}} F_b^{(k+1)}(e)[i]$.*

We give a theorem similar to Theorem 8 for F_b, which restricts the number of entries of $F_b(e)$ that have to be recomputed for each event e on arrival of a new event.

Theorem 11 *Consider an event $e \in E^{(k)}$ and a process p_i. Let $f = F_b^{(k)}(e)[i]$. Then,*

$$J_b^{(k)}(e) \text{ and } J_b^{(k)}(f) \text{ are non-trivial consistent cuts} \;\Rightarrow\; F_b^{(k+1)}(e)[i] = F_b^{(k)}(e)[i] = f$$

The correctness of Theorem 11 follows from Theorem 8. The on-line algorithm for recomputing $F_b(e)$ for each event e can be derived easily using Lemma 10, Theorem 11 and the fact that F_b is order-preserving. The worst-case time-complexity of the algorithm for recomputing F_b is $O(n|E|)$, which is same as that for the off-line version. However, in practice, it may perform much better because of Theorem 11.

6 Conclusion

In this paper, we show the equivalence of three important problems in distributed systems. Using our equivalence results, we significantly enlarge the set of predicates for which it is possible to compute the slice of a computation efficiently. Furthermore, we present an efficient on-line algorithm for computing the slice for a regular predicate.

References

[1] S. Alagar and S. Venkatesan. Techniques to Tackle State Explosion in Global Predicate Detection. *IEEE Transactions on Software Engineering*, 27(8):704–714, August 2001.

[2] 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.

[3] B. Charron-Bost, C. Delporte-Gallet, and H. Fauconnier. Local and Temporal Predicates in Distributed Systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(1):157–179, 1995.

[4] 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.

[5] 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.

[6] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, Cambridge, UK, 1990.

[7] V. K. Garg. *Elements of Distributed Computing.* Wiley & Sons, New York, NY, 2002.

[8] V. K. Garg and N. Mittal. On Slicing a Distributed Computation. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS)*, pages 322–329, April 2001.

[9] 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.

[10] 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.

[11] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[12] 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.

[13] N. Mittal and V. K. Garg. Software Fault Tolerance of Distributed Programs using Computation Slicing. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS)*, Providence, Rhode Island, May 2003.

[14] A. Sen and V. K. Garg. Detecting Temporal Logic Predicates in the Happened-Before Model. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, April 2002.

[15] S. D. Stoller and F. Schneider. Faster Possibility Detection by Combining Two Approaches. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 318–332, September 1995.

[16] 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)*, pages 264–279. Springer-Verlag, July 2000.

A Notation and Background

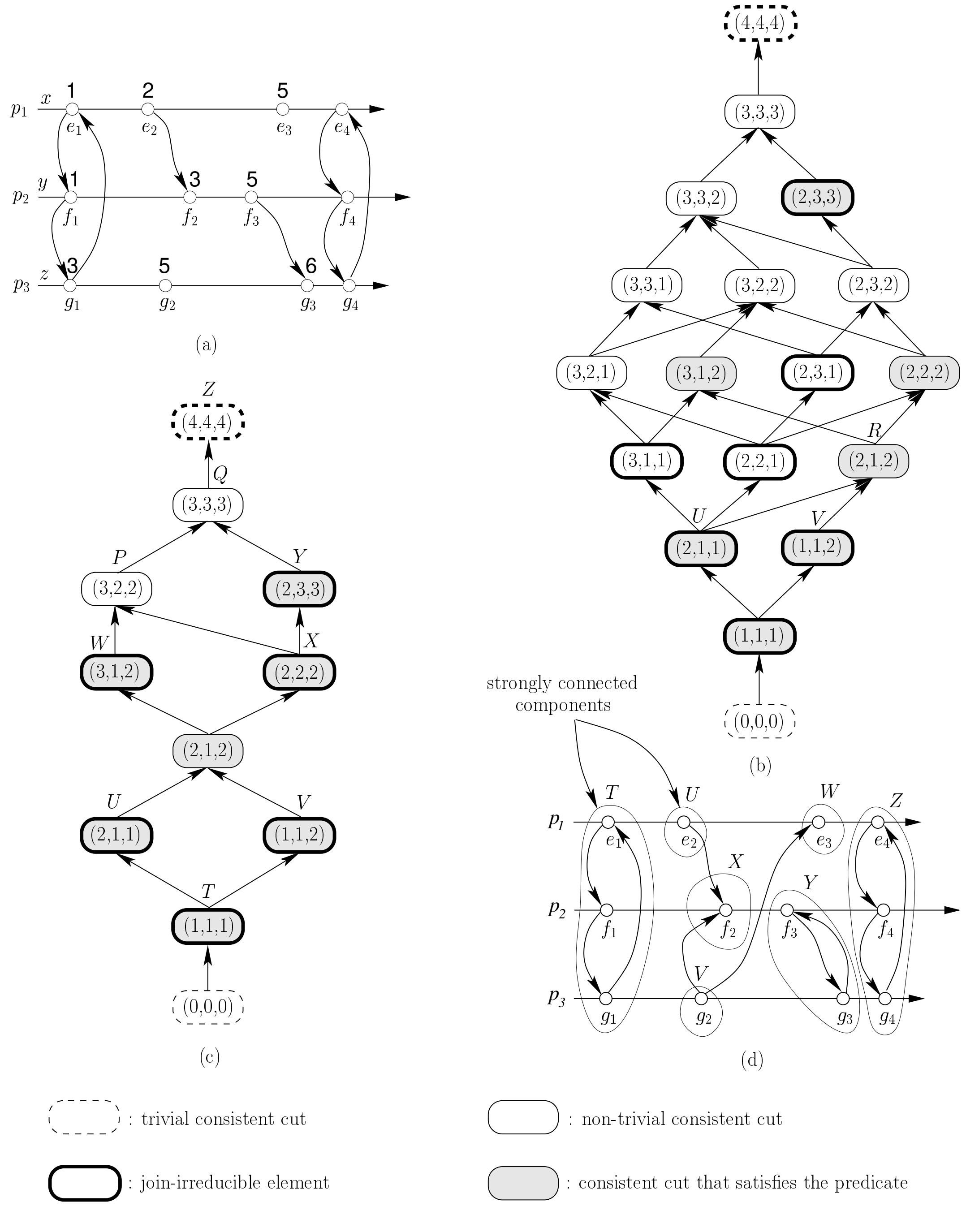

Figure 5: (a) A computation, (b) the lattice of its consistent cuts, (c) the smallest sublattice that contains all consistent cuts satisfying the predicate $x + y - z \leqslant 1$, and (d) the poset induced on the set of join-irreducible elements of the sublattice.

Example 1 Consider the computation depicted in Figure 5(a). It has three processes, namely p_1, p_2 and p_3. The events e_1, f_1 and g_1 are the initial events, and the events e_4, f_4 and g_4 are the final events of the computation. The cut $A = \{e_1, e_2, e_3, e_4, f_1, g_1\}$ is not consistent because $g_4 \rightarrow e_4$ and $e_4 \in A$ but $g_4 \notin A$. On the other hand, the cut $\{e_1, e_2, f_1, f_2, g_1\}$ is a consistent cut. The events e_1,

f_1 and g_1 belong to the same strongly connected component or meta-event. Processes p_1, p_2 and p_3 host integer variables x, y and z, respectively. The predicate $x \leqslant 1$ is a local predicate whereas the predicate $x + y \leqslant z$ is not. The consistent cut $\{e_1, f_1, g_1\}$ satisfies the predicate $x + y \leqslant z$ but the consistent cut $\{e_1, e_2, f_1, f_2, g_1\}$ does not. □

A.1 Birkhoff's Theorem

We first describe some concepts needed to understand the theorem. Given a lattice, its meet (infimum) and join (supremum) operators are denoted by $\sqcap$ and $\sqcup$, respectively. A lattice is *distributive* if meet distributes over join [6]. Formally, $a \sqcap (b \sqcup c) \equiv (a \sqcap b) \sqcup (a \sqcap c)$. It can be proved that meet distributes over join if and only if join distributes over meet. We call an element of a lattice *join-irreducible* if it cannot be expressed as join of two distinct elements (of the lattice), both different from itself [6].

Let L be a lattice and $\mathcal{JI}(L)$ be the set of its join-irreducible elements. In case L is a distributive lattice, it satisfies an important property. Specifically, every element in L can be expressed as join of some subset of elements in $\mathcal{JI}(L)$ and vice versa [6, Birkhoff's Theorem]. In other words, $\mathcal{JI}(L)$ completely characterizes L. This is significant because $|\mathcal{JI}(L)|$ is generally much smaller—exponentially in many cases—than $|L|$. Hence if some computation on L can instead be performed on $\mathcal{JI}(L)$, we obtain a significant computational advantage.

Consider a computation $\langle E, \to \rangle$ and let $\mathcal{C}(E)$ denote the set of its consistent cuts. In [8], we show that $\mathcal{C}(E)$ forms a distributive lattice under the relation $\subseteq$; its join and meet operators correspond to set union ($\cup$) and set intersection ($\cap$), respectively. Furthermore, no additional structural property is satisfied by $\mathcal{C}(E)$. The set of join-irreducible elements of $\mathcal{C}(E)$ is isomorphic to the set of strongly connected components of $\langle E, \to \rangle$.

Now, consider a subset $\mathcal{D} \subseteq \mathcal{C}(E)$. We say that $\mathcal{D}$ forms a *sublattice* of $\mathcal{C}(E)$ if $\mathcal{D}$ is closed under set union and set intersection. That is, given two consistent cuts from $\mathcal{D}$, the consistent cuts obtained by their set union and set intersection also belong to $\mathcal{D}$. It can be proved that any sublattice of a distributive lattice is also a distributive lattice [6]. Thus if $\mathcal{D}$ is a sublattice of $\mathcal{C}(E)$, then, using Birkhoff's Theorem, $\mathcal{JI}(\mathcal{D})$ completely characterizes $\mathcal{D}$. This forms the basis for the notion of computation slice.

Example 2 Consider the computation shown in Figure 5(a). The (distributive) lattice spanned by its set of consistent cuts is shown in Figure 5(b). In the figure, each consistent cut is labeled with the number of events that have to be executed on each process to reach the cut. The join-irreducible elements of the lattice have been drawn with thick boundaries. (They have exactly one incoming edge.) The lattice has eight join-irreducible elements which is same as the number of strongly connected components of the computation. It can be verified that every consistent cut of the computation can be obtained as the join of some subset of these eight join-irreducible elements and vice versa. For instance, the consistent cut R (in Figure 5(b)) can be expressed as the join of the consistent cuts U and V. □

A.2 Establishing the Existence and Uniqueness of Slice

The main idea is as follows. Consider a computation $\langle E, \to \rangle$ and a predicate b. Let $\mathcal{C}(E)$ denote the set of consistent cuts of $\langle E, \to \rangle$ and, further, let $\mathcal{C}_b(E) \subseteq \mathcal{C}(E)$ be the subset of those consistent cuts that satisfy b. We show that there exists a subset $\mathcal{D} \subseteq \mathcal{C}(E)$ satisfying the following conditions. First, $\mathcal{D}$ contains $\mathcal{C}_b(E)$, that is, $\mathcal{C}_b(E) \subseteq \mathcal{D}$. Second, D forms a sublattice of $\mathcal{C}(E)$. Last, among all sublattices that fulfill the first two conditions, it is the *smallest* one. From Birkhoff's Theorem, $\mathcal{JI}(\mathcal{D})$, the set of join-irreducible elements of $\mathcal{D}$, completely characterizes $\mathcal{D}$. We call the poset (partially ordered set) induced on the consistent cuts of $\mathcal{JI}(\mathcal{D})$ by the relation $\subseteq$ as the slice of

$\langle E, \to \rangle$ with respect to b. The slice can also be represented by a directed graph drawn on the set of events such that the set of consistent cuts of the graph is exactly $\mathcal{D}$. Whereas the *poset representation* of a slice is better for presentation purposes, the *graph representation* is more suited for slicing algorithms.

Example 3 Consider the lattice of consistent cuts depicted in Figure 5(b). The consistent that satisfy the predicate $x + y - z \leqslant 1$ have been shaded in the figure. Figure 5(c) depicts the smallest sublattice that contains these consistent cuts. The consistent cuts P and Q do not satisfy the predicate but have been included to complete the sublattice. The join-irreducible elements of the sublattice have been drawn with thick boundaries. There are, in total, seven join-irreducible elements, namely T, U, V, W, X, Y and Z. Figure 5(d) portrays the partial order induced on the set $\mathcal{J} = \{T, U, V, W, X, Y, Z\}$. There is a one-to-one correspondence between the set of join-irreducible elements and the set of strongly connected components of the graph shown in Figure 5(d). It can be verified that every consistent cut in the sublattice can be expressed as join of some subset of $\mathcal{J}$ and, furthermore, the join of every subset of $\mathcal{J}$ is a consistent cut of the sublattice. $\square$

A.3 Skeletal Representation of a Slice

In general, there can be multiple directed graphs with the same set of consistent cuts. Therefore more than one graph may constitute a valid representation of the given slice. The following lemma states that all such graphs are in fact related.

Lemma 12 ([12]) *Consider directed graphs G and H on the same set of vertices. Then,*

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

which in turn implies that:

Lemma 13 ([12]) *Consider directed graphs G and H on the same set of vertices. Then,*

$$\mathcal{P}(G) = \mathcal{P}(H) \equiv \mathcal{C}(G) = \mathcal{C}(H)$$

In other words, two directed graphs G and H, on identical sets of vertices, are *cut-equivalent* (that is, $\mathcal{C}(G) = \mathcal{C}(H)$) if and only if they are *path-equivalent* (that is, $\mathcal{P}(G) = \mathcal{P}(H)$). We consider a special directed graph to capture a slice, called the *skeletal representation* of a slice [12]. Let $F_b(e)$ be a vector of events, where the i^{th} entry in the vector denotes the earliest event on process p_i reachable from e in the slice. The skeletal representation of a slice has 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 $F_b(e)[i]$.

An advantage of the skeletal representation is that it has $O(|E|)$ vertices and only $O(n|E|)$ edges, where n is the number of processes and E is the set of events, and hence generally leads to more efficient algorithms involving slices.

B Omitted Proofs

Proof for Lemma 1: We have,

$$
\begin{aligned}
& \text{there is a path from } e \text{ to } f \text{ in } \mathsf{slice}(G,b) \\
\equiv\; & \{ \text{ definition of } \mathsf{slice}(G,b) \text{ } \} \\
& (\text{there is a path from } e \text{ to } f \text{ in } \mathsf{slice}(G,b)) \wedge \Big(\mathcal{C}(\mathsf{slice}(G,b)) \subseteq \mathcal{C}(G)\Big) \\
\equiv\; & \{ \text{ from Lemma 12, } \mathcal{C}(\mathsf{slice}(G,b)) \subseteq \mathcal{C}(G) \equiv \mathcal{P}(G) \subseteq \mathcal{P}(\mathsf{slice}(G,b)) \text{ } \} \\
& (\text{there is a path from } e \text{ to } f \text{ in } \mathsf{slice}(G,b)) \wedge \Big(\mathcal{P}(G) \subseteq \mathcal{P}(\mathsf{slice}(G,b))\Big) \\
\equiv\; & \{ \text{ definition of } G[e,f] \text{ } \} \\
& \mathcal{P}(G[e,f]) \subseteq \mathcal{P}(\mathsf{slice}(G,b)) \\
\equiv\; & \{ \text{ from Lemma 12 } \} \\
& \mathcal{C}(\mathsf{slice}(G,b)) \subseteq \mathcal{C}(G[e,f]) \\
\equiv\; & \{ \text{ } \mathcal{C}(\mathsf{slice}(G,b)) \text{ contains all consistent cuts of } \mathcal{C}(G) \text{ satisfying } b \text{ } \} \\
& \text{no consistent cut in } \ \mathcal{C}(G) \setminus \mathcal{C}(G[e,f]) \ \text{ satisfies } b
\end{aligned}
$$

This establishes the lemma. $\Box$

Proof for Lemma 2: Consider a *non-trivial* consistent cut C in $\mathcal{C}(G)$. It suffices to show that $C \in \mathcal{C}(\widehat{G}[e,f]) \equiv C \notin \mathcal{C}(G[e,f])$.

(⇒) We need to prove that $C \in \mathcal{C}(\widehat{G}[e,f]) \Rightarrow C \notin \mathcal{C}(G[e,f])$. Intuitively, it means that $G[e,f]$ and $\widehat{G}[e,f]$ do not have any common non-trivial consistent cut. Equivalently, $\mathcal{C}(G[e,f]) \cap \mathcal{C}(\widehat{G}[e,f]) = \{\emptyset, E\}$. We have,

$$
\begin{aligned}
& C \in \mathcal{C}(\widehat{G}[e,f]) \\
\Rightarrow\; & \{ \text{ definition of } \widehat{G}[e,f] \text{ } \} \\
& (f \in C) \wedge (e \notin C) \\
\Rightarrow\; & \{ \text{ definition of } G[e,f] \text{ } \} \\
& C \notin \mathcal{C}(G[e,f])
\end{aligned}
$$

(⇐) We need to prove that $C \notin \mathcal{C}(G[e,f]) \ \Rightarrow\ C \in \mathcal{C}(\widehat{G}[e,f])$. Intuitively, it means that every non-trivial consistent cut of G is either a consistent cut of $G[e,f]$ or a consistent cut of $\widehat{G}[e,f]$. Equivalently, $\mathcal{C}(G[e,f]) \cup \mathcal{C}(\widehat{G}[e,f]) = \mathcal{C}(G)$. The proof consists of two steps. In the first step, we show that if C is not a consistent cut of $G[e,f]$, then it is the case that $e \notin C$ and $f \in C$. Assume that $C \notin \mathcal{C}(G[e,f])$. Therefore there exist events u and v such that there is a path from u to v in $G[e,f]$, $u \notin C$ and $v \in C$. Since C is a consistent cut of G, there is no path from u to v in G. That is, $(u,v) \in \mathcal{P}(G[e,f])$ but $(u,v) \notin \mathcal{P}(G)$. Thus every path from u to v in $G[e,f]$ involves the edge (e,f). This implies that $(u,e) \in \mathcal{P}(G)$ and $(f,v) \in \mathcal{P}(G)$. Since $(u,e) \in \mathcal{P}(G)$ and $u \notin C$, $e \notin C$. Similarly, since $(f,v) \in \mathcal{P}(G)$ and $v \in C$, $f \in C$. Therefore $e \notin C$ and $f \in C$.

In the second step, we show that if C is not a consistent cut of $\widehat{G}[e,f]$, then it is the case that $(e \in C) \vee (f \notin C)$. Assume that $C \notin \mathcal{C}(\widehat{G}[e,f])$. Hence there exist events x and y such that there is a path from x to y in $\widehat{G}[e,f]$, $x \notin C$ and $y \in C$. Since C is a consistent cut of G, there is no path from x to y in G. That is, $(x,y) \in \mathcal{P}(\widehat{G}[e,f])$ but $(x,y) \notin \mathcal{P}(G)$. Thus every path from x to y in $\widehat{G}[e,f]$ either involves the edge $(f, \bot_1)$ or the edge $(\top_1, e)$. In the first case, there is a path from x to f in G which implies that $f \notin C$. In the second case, there is path from e to y in G which implies that $e \in C$. Consequently, either $e \in C$ or $f \notin C$—a logical negation of what we obtained in the first step.

Combining the two steps, it can be inferred that it cannot be the case that C is neither a consistent cut of $G[e,f]$ nor a consistent cut of $\widehat{G}[e,f]$. This establishes the lemma. □

Proof for Theorem 4: The initialization at line 1 requires $O(|E|^2)$ time, where E is the set of events, because G has $|E|$ vertices and therefore $O(|E|^2)$ edges. The for loop at line 2 executes $|E|^2$ times. Each iteration of the for loop requires solving an instance of CONTC. The construction of the particular instance of CONTC involves adding two edges to G, and therefore can be done in $O(1)$ time. Depending on the result of the if statement at line 3, an edge may be required to be added to K at line 4, which can be done in $O(1)$ time. At the end of the iteration, the two edges that were added to G have to be deleted. The deletion can be accomplished in $O(1)$ time by maintaining pointers to the two edges if using adjacency list representation. The overall time-complexity of the for loop is $O(|E|^2 T)$, which is also the time-complexity of the algorithm. □

Proof for Theorem 5: Note that the while loop at line 4 terminates in at most $|E_i| + |E_j|$ iterations, where E_i and E_j denote the set of events on processes p_i and p_j, respectively. This is because the directed graph $\widehat{G}[e,f]$ when $f = \top_j$ has an edge from the final event $\top_j$ to the initial event $\bot_1$ implying that $\widehat{G}[e,\top_j]$ has no non-trivial consistent cut. Therefore $\mathsf{CONTC}(\widehat{G}[e,f], b)$ when $f = \top_j$ will, trivially, evaluate to false. This gives the time-complexity of $O((|E_i| + |E_j|)\,T)$ for the inner for loop at line 3. Hence the time-complexity of the outer for loop at line 1 is $O((n|E_i| + |E|)\,T)$. This implies that the overall time-complexity of computing $F_b(e)$ for each event e is $O(n|E|T)$. □

Proof for Theorem 6: The algorithm first checks whether some consistent cut of G satisfies b. In case no such consistent cut exists, it simply outputs the appropriate message at line 1 and terminates. Now, assume that G does contain a consistent cut that satisfies b. In that case, the for loop at line 3 executes exactly $|E|$ times, where E is the set of events. Let K_i, where $0 \leqslant i \leqslant |E|$ and $K_0 = G$, be the value of K when the i^{th} iteration of the for loop ends. The algorithm maintains the invariant that each K_i contains at least one consistent cut that satisfies b. The proof is by induction on i. When $i = 0$, by definition, $K_0 = G$ and the proposition holds. Assume that the proposition holds for K_i. We need to show that it also holds for K_{i+1}. Let e be the event in consideration at the $(i+1)^{st}$ iteration of the for loop. At line 4 the algorithm tests whether $K_i[e, \bot_1]$ contains a consistent cut that satisfies b. If the answer is yes, then K_{i+1} is set to $K_i[e, \bot_1]$ at line 5. Clearly, in this case, K_{i+1} contains a consistent cut that satisfies b. On the other hand, if the test fails, then K_{i+1} is set to $K_i[\top_1, e]$. From Lemma 2, $\mathcal{C}(K_i) = \mathcal{C}(K_i[e, \bot_1]) \cup \mathcal{C}(\widehat{K}_i[e, \bot_1])$. From induction hypothesis, K_i contains a consistent cut that satisfies b. However, $K_i[e, \bot_1]$ does not contain any such consistent cut. Therefore $\widehat{K}_i[e, \bot_1]$ should contain a consistent cut that satisfies b. Note that $\widehat{K}_i[e, \bot_1]$ is obtained from K_i by adding edges $(\bot_1, \bot_1)$ and $(\top_1, e)$. Clearly, the presence or absence of the edge $(\bot_1, \bot_1)$ does not affect the set of consistent cuts. Therefore K_{i+1} contains a consistent cut that satisfies b.

We next prove that $K_{|E|}$ contains exactly one non-trivial consistent cut. Assume the contrary. Therefore $K_{|E|}$ contains two *non-trivial* consistent cuts C and D with $C \neq D$. Evidently, either $C \setminus D \neq \emptyset$ or $D \setminus C \neq \emptyset$. Without loss of generality assume that it is the former, and let e be an event in $C \setminus D$. The for loop at line 3 considers events in E one by one. Let the iteration corresponding to e be i. After the i^{th} iteration ends, K_i contains either $(e, \perp_1)$ or $(\top_1, e)$. Since no edge is ever deleted, $K_{|E|}$ also contains either $(e, \perp_1)$ or $(\top_1, e)$. In the first case, since $e \notin D$, $\perp_1 \notin D$—a contradiction. In the second case, since $e \in C$, $\top_1 \in C$—a contradiction. $\Box$

Proof for Theorem 7: Clearly, the initialization at line 2 has $O(n|E|)$ time-complexity, where n is the number of processes and E is the set of events, assuming that the skeletal representation is used. Also, each iteration of the for loop at line 3 has $O(T)$ time-complexity. Therefore the overall time-complexity of the algorithm is $O(n|E| + |E|\,T)$. It is reasonable to assume that $T = \Omega(n)$. This gives the time-complexity of $O(|E|\,T)$. $\Box$

Proof for Theorem 8: Assume the antecedent. Since $\langle E^{(k)}, \rightarrow\rangle$ is a "subcomputation" of $\langle E^{(k+1)}, \rightarrow\rangle$ and $J_b^{(k)}(e)$ does not contain any final event, $J_b^{(k)}(e)$ is a consistent cut of $\langle E^{(k+1)}, \rightarrow\rangle$ as well. Therefore both $J_b^{(k)}(e)$ and $J_b^{(k+1)}(e)$ are consistent cuts of $\langle E^{(k+1)}, \rightarrow\rangle$, contain e and satisfy b. Since $J_b^{(k+1)}(e)$ is the *least* such cut, $J_b^{(k+1)}(e) \subseteq J_b^{(k)}(e)$.

Now, $J_b^{(k+1)}(e) \subseteq J_b^{(k)}(e)$ implies that $J_b^{(k+1)}(e)$ is a consistent cut of $\langle E^{(k)}, \rightarrow\rangle$ as well. Thus both $J_b^{(k)}(e)$ and $J_b^{(k+1)}(e)$ are consistent cuts of $\langle E^{(k)}, \rightarrow\rangle$, contain e and satisfy b. However, $J_b^{(k)}(e)$ is the *least* such cut. As a result, $J_b^{(k)}(e) \subseteq J_b^{(k+1)}(e)$.

Combining the two, we obtain $J_b^{(k)}(e) = J_b^{(k+1)}(e)$. $\Box$

Proof for Theorem 9: Suppose m events are executed in total. Let $add_i^{(k)}$ denote the set of events added to $current_i$ when the algorithm RecomputeJ is executed on arrival of the k^{th} event. It can be verified that the time-complexity of the k^{th} invocation of the algorithm is given by $O(\sum_{i=1}^{n}(n + n|add_i^{(k)}|))$, which could be as large as $O(n^2 m)$ because $|add_i^{(k)}|$ in the worst-case could be $O(m)$. The initialization of $recompute_i$ and $current_i$ for each i takes $O(n^2)$ time in total. The time-complexity of the algorithm *summed* over all the m invocations is therefore given by:

$$
\begin{aligned}
& O\Big(n^2 + \sum_{k=1}^{m}\sum_{i=1}^{n}(n + n|add_i^{(k)}|)\Big) \\
= \;& \{ \text{ changing the order of summations } \} \\
& O\Big(n^2 + \sum_{i=1}^{n}\sum_{k=1}^{m}(n + n|add_i^{(k)}|)\Big) \\
= \;& \{ \; |add_i^{(1)}| + |add_i^{(2)}| + \cdots + |add_i^{(m)}| \leqslant m \; \} \\
& O\Big(n^2 + \sum_{i=1}^{n}(nm + nm)\Big) \\
= \;& \{ \text{ simplifying } \} \\
& O(n^2 m)
\end{aligned}
$$

Therefore the amortized time-complexity for each iteration is $O(n^2)$, which is much less than the worst-case time-complexity of $O(n^2 m)$. $\square$

Proof for Lemma 10: For convenience, let $f = F_b^{(k)}(e)[i]$. In case $f = \perp_i$, the lemma holds trivially. Therefore assume that $f \neq \perp_i$ and let $g = pred(f)$. There are two cases to consider:

Case 1 [$J_b^{(k)}(e)$ is a non-trivial consistent cut]: Clearly, $J_b^{(k)}(g)$ is also a non-trivial consistent cut and $J_b^{(k)}(e) \not\subseteq J_b^{(k)}(g)$; otherwise, $J_b^{(k)}(e) \subseteq J_b^{(k)}(g)$ which contradicts the choice of f. Using Theorem 8, $J_b^{(k+1)}(e) = J_b^{(k)}(e)$ and $J_b^{(k+1)}(g) = J_b^{(k)}(g)$. Consequently, $J_b^{(k+1)}(e) \not\subseteq J_b^{(k+1)}(g)$ implying that $g \stackrel{P}{\rightarrow} F_b^{(k+1)}(e)[i]$.

Case 2 [$J_b^{(k)}(e)$ is the default cut]: This implies that $J_b^{(k)}(f)$ is the default cut. Moreover, $J_b^{(k)}(g)$ is a non-trivial consistent cut; otherwise, $J_b^{(k)}(e) \subseteq J_b^{(k)}(g)$ which contradicts the choice of f. Using Theorem 8, $J_b^{(k+1)}(g) = J_b^{(k)}(g)$. If $J_b^{(k+1)}(e) \subseteq J_b^{(k+1)}(g)$, then $J_b^{(k+1)}(e) \subseteq J_b^{(k)}(g)$, which implies that $J_b^{(k+1)}(e)$ is a non-trivial consistent cut of $\langle E^{(k)}, \rightarrow \rangle$. As a result, there is a consistent cut of $\langle E^{(k)}, \rightarrow \rangle$ that contains e and satisfies b, which contradicts the fact that $J_b^{(k)}(e)$ is the default cut. Consequently, $J_b^{(k+1)}(e) \not\subseteq J_b^{(k+1)}(g)$ implying that $g \stackrel{P}{\rightarrow} F_b^{(k+1)}(e)$.

This establishes the lemma.

$\square$

C An Illustration of Lemma 1 and Lemma 2

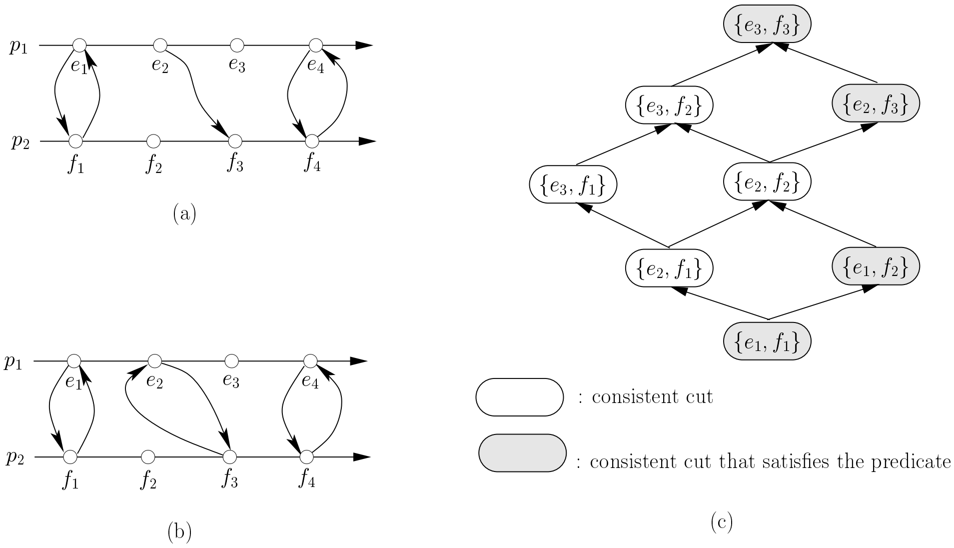

Figure 6: (a) A directed graph G, (b) $\mathsf{slice}(G, b)$, where b is the predicate "all channels are empty", and (c) the set of non-trivial consistent cuts of H.

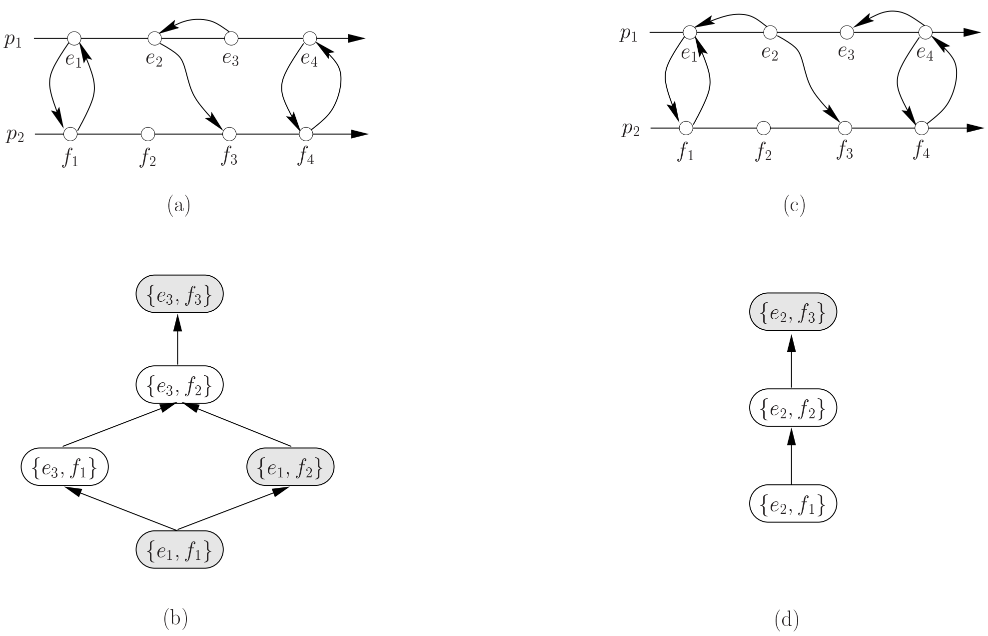

Figure 7: (a) The directed graph $G[e_3, e_2]$, where there is no path from e_3 to e_2 in $\mathsf{slice}(G, b)$, (b) the set of non-trivial consistent cuts of $G[e_3, e_2]$, (c) the directed graph $\widehat{G}[e_3, e_2]$, and (d) the set of non-trivial consistent cuts of $\widehat{G}[e_3, e_2]$.

Example 4 Consider the directed graph G shown in Figure 6(a). In the graph, e_1 and f_1 are the initial events, whereas e_4 and f_4 are the final events. Figure 6(b) depicts $\mathsf{slice}(G, b)$, where b is the predicate "all channels are empty". The slice is obtained by adding the edge from f_3 to e_2 because, for all channels to be empty, send and receive events of the same message have to be

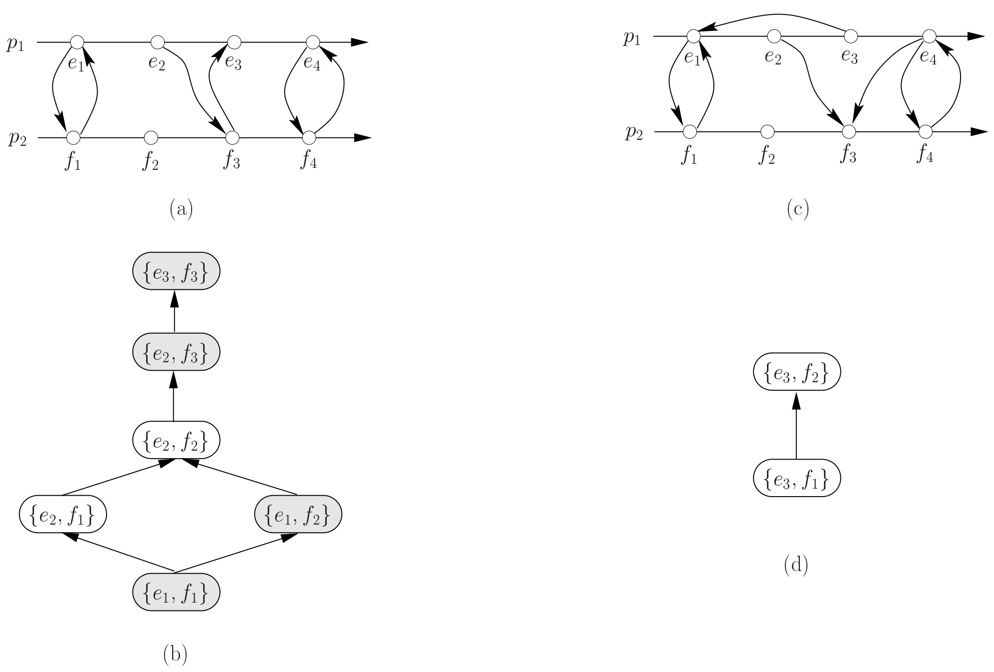

Figure 8: (a) The directed graph $G[f_3, e_3]$, where there is a path from f_3 to e_3 in $\mathsf{slice}(G, b)$, (b) the set of non-trivial consistent cuts of $G[f_3, e_3]$, (c) the directed graph $\widehat{G}[f_3, e_3]$, and (d) the set of non-trivial consistent cuts of $\widehat{G}[f_3, e_3]$.

executed atomically. The set of non-trivial consistent cuts of G are shown in Figure 6(c). The cuts for which all channels are empty have been shaded. Now, consider directed graphs $G[e_3, e_2]$ and $\widehat{G}[e_3, e_2]$ shown in Figure 7(a) and Figure 7(c), respectively. Their sets of consistent cuts (excluding trivial consistent cuts) are shown in Figure 7(b) and Figure 7(d), respectively. As expected, the two sets satisfy Lemma 2. Also, since there is no path from e_3 to e_2 in $\mathsf{slice}(G, b)$, $\widehat{G}[e_3, e_2]$ contains a consistent cut that satisfies b. Figure 8 illustrates the case when the edge (f_3, e_3) is such that $\mathsf{slice}(G, b)$ contains a path from f_3 to e_3. $\square$

D An Illustration of the Algorithm in Figure 4 for Solving COMPC

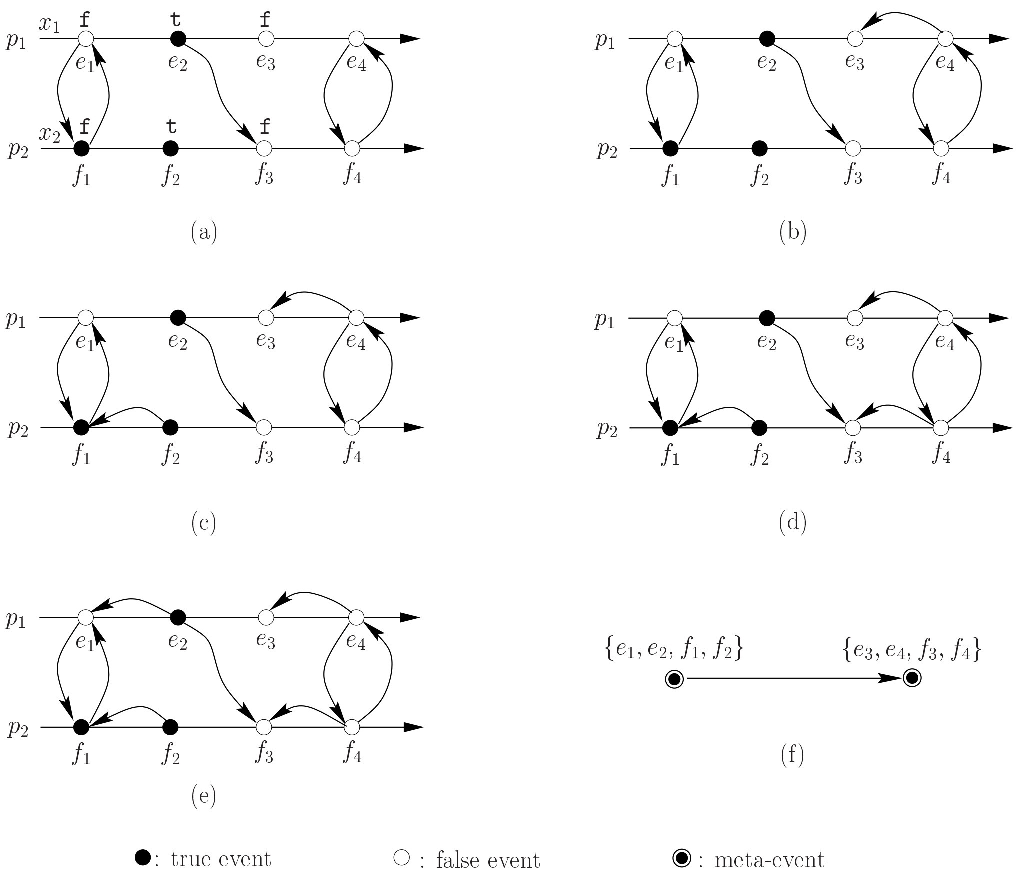

Figure 9: All illustration of the algorithm in Figure 4.

Example 5 Consider the computation shown in Figure 9(a). In the figure, e_1 and f_1 are the initial events and e_4 and f_4 are the final events. Suppose we want to find a consistent cut for which both x_1 and x_2 evaluate to true. In the figure, we represent an event for which the corresponding local predicate evaluates to true—referred to as a *true event*—using a solid circle. Clearly, there are two consistent cuts which satisfy $x_1 \wedge x_2$, namely $\{e_1, e_2, f_1\}$ and $\{e_1, e_2, f_1, f_2\}$. Suppose the events are considered in the order e_3 f_2 f_3 e_2. (There is no need to consider the initial and final events.) Figure 9(b) to Figure 9(e) depict the sequence of iterations of the for loop. In the first iteration, an edge is added from $\top_1$ to e_3. In the second iteration, an edge is added from f_2 to $\bot_2$, which is equivalent to adding an edge from f_2 to $\bot_1$. In the third iteration, an edge is added from $\top_2$ to f_3, which is same as adding an edge from $\top_1$ to f_3. Finally, in the fourth iteration, an edge is added from e_2 to $\bot_1$. Figure 9(f) shows the resultant slice as a poset on meta-events. Clearly, the slice has exactly one non-trivial consistent cut, namely $\{e_1, e_2, f_1, f_2\}$. □

E Reducing the Time-Complexity of the Algorithm for Solving COMPC

The main idea is as follows. Consider an event e on process p_i. Suppose the if statement at line 4 evaluates to true for e. In that case, we will add an edge from e to $\perp_1$. The edge will also create a path from every event that occurred before e on p_i to $\perp_1$. These events do not need to be considered at all by the for loop and can be simply ignored. On the other hand, in case the if statement evaluates to false, we will add an edge from $\top_1$ to e. Similar to before, the edge will also create a path from $\top_1$ to every event that occurred after e on p_i. Again, these events do not need to be considered by the for loop. Therefore, by selecting e appropriately, it is possible to eliminate close to half of the remaining events on p_i from consideration in a similar fashion as binary search.

F Details of the Algorithm RecomputeJ

<u>Input</u>: (1) a regular predicate b, (2) the computation $\langle E^{(k+1)}, \to \rangle$, and
(3) for each event $e \in E^{(k)}$, $J_b(e)$ currently set to $J_b^{(k)}(e)$

<u>Output</u>: for each event $e \in E^{(k+1)}$, $J_b(e)$ now set to $J_b^{(k+1)}(e)$

```
J_b(e^(k+1)) := the default cut;
// add e^(k+1) to the set of events whose J_b has to be recomputed
recompute_new := min{recompute_new, e^(k+1)} where p_new = proc(e^(k+1));

for each process p_i do
    // resume the scan from current_i
    done := (recompute_i = ⊤_i);

    // as soon as done becomes true, move to the next process
    while not(done) do
        // update J_b(recompute_i), if required
        found := false;
        while not(found) and not(done) do
            if there exist events g and h in current_i's frontier such that succ(g) → h then
                // current_i is not a consistent cut
                if succ(g) ∉ ⊤ then
                    current_i := current_i ∪ {succ(g)};
                else  done := true;                          // terminate the scan
                endif;
            else                                             // current_i is a consistent cut
                if current_i satisfies b_{recompute_i} then found := true;
                else
                    // invoke the linearity property
                    g := the forbidden event of current_i with respect to b_{recompute_i};
                    if succ(g) ∉ ⊤ then
                        current_i := current_i ∪ {succ(g)};
                    else  done := true;                      // terminate the scan
                    endif;
                endif;
            endif;
        endwhile;
        if (found) then
            // found a consistent cut that contains recompute_i and satisfies b
            J_b(recompute_i) := current_i;
            // move to the next event on p_i
            recompute_i := succ(recompute_i);
        endif;
        if not(found) or (recompute_i = ⊤_i) then
            done := true;                                    // terminate the scan
        endif;
    endwhile;
endfor;
```

Figure 10: Algorithm RecomputeJ to update J_b on arrival of the new event $e^{(k+1)}$.

We now give implementation details of the algorithm. First, an event is represented by an integer corresponding to the number of events that have been executed so far on the process not including the event itself. For example, the event $\bot_i$ will be represented by 0. Second, so that the representation of a final event does not change on arrival of new events, we choose a sufficiently large integer, which is bigger than the number of events that can be executed on any process. Third, we represent a cut using its frontier, which can be done using a vector of n integers, one for each process.