On Checking Whether a Predicate Definitely Holds

Alper Sen and Vijay K. Garg

TR-PDS-2003-003 **April 2003**

Parallel and Distributed Systems Group
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, Texas 78712

On Checking Whether a Predicate Definitely Holds *

Alper Sen and Vijay K. Garg
Dept. of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX, 78712, USA
{sen, garg}@ece.utexas.edu
http://www.ece.utexas.edu/˜{sen,garg}

Abstract

Predicate detection problem is an important problem in testing and debugging distributed programs. Cooper and Marzullo introduced two modalities *possibly* and *definitely* as a solution to this problem. Given a predicate p, a computation satisfies *possibly*$: p$ if p is true for any global state in the computation. A computation satisfies *definitely*$: p$ if *all* paths from the initial to the final global state go through some global state that satisfies p. In general, *definitely* modality is used to detect *good* conditions such as "a leader is eventually chosen by all processes", or "a commit point is reached by every process", whereas *possibly* modality is used to detect bad conditions such as violation of mutual exclusion. There are several efficient algorithms for *possibly* modality in the literature [9, 11, 1, 2, 21]. However, this is not the case for *definitely* modality. Cooper and Marzullo's *definitely*$: p$ algorithm for arbitrary p has a worst case space and time complexity exponential in the number of processes. This is due to the global state explosion problem. In this paper we present efficient algorithms for detecting *definitely*$: p$. In particular, we give a simple algorithm that uses polynomial space. Then, we present a global state space reduction algorithm that can significantly reduce the global state space. We determine necessary conditions and sufficient conditions under which detecting *definitely*$: p$ may be efficiently solved. We apply our algorithms to an example, achieving a speedup of over 100, compared to partial order reduction based technique of SPIN [10].

*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

1 Introduction

Detection of a global predicate is a fundamental problem in distributed computing. This problem arises in many contexts such as testing and debugging of distributed programs. For example, when debugging a distributed mutual exclusion algorithm, it is useful to monitor the system to detect concurrent accesses to the shared resources.

Cooper and Marzullo introduced two modalities for predicate detection, which are denoted by *possibly* and *definitely*. Given a predicate p, a computation satisfies *possibly*$:p$ if p is true for any global state in the computation. A computation satisfies *definitely*$:p$ if *all* paths from the initial state to the final global state go through some global state that satisfies p. In general, *possibly* modality is used to detect *bad* conditions such as the system reaches a global state where the mutual exclusion predicate is false. In contrast, *definitely* modality is in general used to detect *good* conditions such as "a leader is eventually chosen by all processes", or "a commit point is reached by every process". Cooper and Marzullo's definitions of these modalities established an important conceptual framework for predicate detection, which has been the basis of considerable research. However, most of the research has focused on *possibly* modality [9, 11, 1, 2, 21].

Cooper and Marzullo present an algorithm for detecting *definitely*$:p$ for arbitrary predicate p. The worst case space and time complexities of the their algorithm are exponential in the number of processes. This is due to the *state explosion problem*—in a distributed system of n processes, the number of possible global states (state-space) can be of size $O(m^n)$, where m is the maximum number of events on a process.

This paper presents efficient algorithms for detecting *definitely*$:p$. We first present a simple algorithm for *definitely*$:p$ that uses $O(nm)$ space in Section 4. Then, we present a polynomial time state-space reduction algorithm that enables us to work on a distributed computation that is in general much smaller than the original computation. We prove that the original computation satisfies *definitely*$:p$ if and only if the smaller computation satisfies it. It is in general coNP-complete to detect a predicate under *definitely* modality. In Sections 5 and 6, we determine necessary conditions and sufficient conditions under which detecting *definitely*$:p$ may be efficiently solved. In order to develop these conditions we use lattice theoretic properties of distributed computations and their set of global states. We validate the effectiveness of our algorithms with experimental studies in Section 7. For this purpose, we describe an implementation of our techniques and compare its performance to partial order reduction based tool SPIN [10]. In one case, our algorithms are significantly faster and space efficient. We have measured over 100-fold gain. Finally, some concluding remarks are given in Section 8.

2 Related Work

Our approach exploits the structure of the predicate itself—by imposing restrictions—to evaluate its value efficiently for a given computation. Polynomial-time algorithms for *possibly*$:p$ have been developed when p belongs to conjunctive [9, 11], observer-independent [1], linear [2], and relational predicates [21].

Tarafdar and Garg [19] proved that it is in general NP-complete to detect a predicate under *controllable* modality. Since the problem of detecting a predicate under *definitely* modality is the dual of the problem of detecting a predicate under *controllable* modality, it is in general coNP-complete to detect a predicate under *definitely* modality. Using Tarafdar and Garg's [20] NP-completeness result for controlling a special case of 2-CNF predicates, called *independent mutual exclusion* predicates, we can easily deduce that detecting a special case of 2-DNF predicates, which is the dual of independent mutual exclusion predicates, under *definitely* modality is coNP-complete in general.

Fromentin and Raynal [6] presented a polynomial time algorithm to solve the predicate detection problem for *proper* modality, which is a special case of *definitely*, A computation satisfies *proper* :p if all paths from the initial state to the final global state go through a *unique* global state that satisfies p.

The *definitely* :p problem has efficient solutions when the predicate is 1-CNF or 1-DNF [7]. However, the complexity problem is open for *definitely*: p for regular p. In this paper, we present efficient conditions to solve the problem for both arbitrary and regular predicates.

3 Model

We assume a loosely-coupled message-passing asynchronous system without any shared memory or a global clock. A *distributed program* consists of n sequential processes denoted by $P_1, P_2, \ldots, P_n$ communicating via asynchronous messages. In this paper, we are concerned with a single *computation* (*execution*) of a distributed program. We assume that no messages are altered or spuriously introduced. We do not make any assumptions about FIFO nature of channels.

The execution of a process in a computation can be viewed as a sequence of events with events across processes ordered by Lamport's *happened-before* relation, $\rightarrow$ [13]. We use lowercase letters e and f to represent events. The *happened-before* relation between any two events e and f can be formally stated as the smallest relation such that $e \rightarrow f$ if and only if e occurs before f in the same process, or e is a send of a message and f is a receive of that message, or there exists an event g such that e happened-before g and g happened-before f. We represent the set of events as the union of events from each process, $E = \bigcup E_i$, for each $1 \leq i \leq n$. We define a *distributed computation* as the partially ordered set consisting of the set of events together with the happened-before relation and denote it by $\langle E, \rightarrow \rangle$.

We define a *consistent cut* of a computation $\langle E, \rightarrow \rangle$ as a subset $G \subseteq E$ such that $f \in G \wedge e \rightarrow f \Rightarrow e \in G$. We use uppercase letters G, H, J, and K to represent consistent cuts. A consistent cut captures the notion of a reachable global state. We use consistent cut and global state interchangeably. We denote the set of consistent cuts of any distributed computation $\langle E, \rightarrow \rangle$ by $C(E)$. It is well known that the set of consistent cuts of any distributed computation $\langle E, \rightarrow \rangle$ forms a *distributive lattice*, under the relation $\subseteq$ [14, 8]. We denote this lattice by $L = (C(E), \subseteq)$ and also call this as the *state space* of the distributed computation. For any partially ordered set, we use $\sqcup$ and $\sqcap$ to denote join and meet operators. Note that the join (resp. meet) of two consistent cuts correspond to their union (resp. intersection). We use $\perp$ to denote the initial cut and E to denote the final cut.

We denote the set of maximal (with respect to happened-before relation) elements of a consistent cut G by $frontier(G)$. Figure 2 shows a computation and its lattice of consistent cuts. A consistent cut in the figure is represented by its frontier. For example, the consistent cut $\{e_3, e_2, e_1, f_2, f_1, \perp\}$ is represented by $\{e_3, f_2\}$. A consistent cut H is *reachable* from a consistent cut G iff it is possible to attain H from G by executing zero or more events. It is easy to see that H is reachable from G iff $G \subseteq H$. We define *successor* of a cut by a relation $\triangleright \subseteq C(E) \times C(E)$ such that $G \triangleright H$ if and only if $H = G \cup \{e\}$ for some $e \in E$ such that $e \notin G$. We say that H is a *successor* of G and G is a *predecessor* of H. A *path* $G_0, G_1, \ldots, G_l$ of $(C(E), \subseteq)$ satisfies that for each $0 \leq i < l$, $G_i \triangleright G_{i+1}$.

We say that a predicate is *regular* if the set of consistent cuts that satisfy the predicate forms a sublattice of the lattice of consistent cuts. Equivalently, if two consistent cuts satisfy a regular predicate then the cuts given by their set intersection and set union also satisfies the predicate. Let $inf(p)$ and $sup(p)$ denote the least and the greatest cut that satisfies a given predicate p, respectively. From the definition of a regular predicate we deduce that both $inf(p)$ and $sup(p)$ exist for a regular predicate. Regular predicates contain most of the earlier studied predicates such as monotonic channel predicates [7], conjunction of local predicates [7, 12], and some relational predicates [7]. There are efficient algorithms for detecting regular predicates under *possibly* and *controllable* modalities [8, 17].

4 Polynomial Space Algorithm

The performance of algorithms for detecting *definitely*:p can be improved by considering a smaller state space, that is, a smaller computation. Now we present a polynomial time algorithm for reducing the state space. We show that detecting *definitely*:p on the original computation is the same as detecting it on the smaller computation. For this purpose, we first define an *interval* of a computation $\langle E, \rightarrow\rangle$ with respect to cuts C and D as the computation which is a subset of $\langle E, \rightarrow\rangle$ such that only the cuts between C and D (including C and D) belong to the interval. We denote this subset computation by $interval(C, D)$.

We state informally a Lemma before presenting our state space reduction algorithm. Given three consistent cuts G, H and J, where H is reachable from J and H is a successor of G, the intersection of G and J is either J or it is a predecessor of J. We present the proof of this Lemma in the Appendix.

Theorem 1 (NSC) *Given a computation $\langle E, \rightarrow\rangle$, definitely:$p$ holds in $\langle E, \rightarrow\rangle$ iff definitely:p holds in $interval(C, D)$, where C is the meet of predecessors of $inf(p)$, if they exist, otherwise $inf(p)$, and D is the join of successors of $sup(p)$, if they exist, otherwise $sup(p)$.*

Proof: Without loss of generality, assume that both C and D exist and are different from the initial and final cut of $\langle E, \rightarrow\rangle$. We prove the contrapositives.

$\Rightarrow$:

We obtain a path from the initial cut to the final cut in $\langle E, \rightarrow\rangle$ as follows: Pick an arbitrary path from the initial cut of $\langle E, \rightarrow\rangle$ to C. We know that none of the cuts on this path satisfy p since all cuts that satisfy p belong to $interval(C, D)$. Next, using the assumption, continue this arbitrary path with a path in $interval(C, D)$ where none of the cuts on the path satisfy p. Finally, pick an arbitrary path from D to the final cut of $\langle E, \rightarrow\rangle$.

$\Leftarrow$:

Now we prove that if there exists a path from the initial to the final cut in $\langle E, \rightarrow\rangle$ where all cuts on the path satisfy $\neg p$ then there exists a path from the initial to the final cut in $interval(C, D)$ where all cuts on the path satisfy $\neg p$. We prove the claim in two Steps.

Step 1: We first show that if there exists a path, $\mathcal{P}$, from the initial to the final cut in $\langle E, \rightarrow\rangle$ where all cuts on the path satisfy $\neg p$ then there exists a path from the initial to the final cut in $interval(C, E)$ where all cuts on the path satisfy $\neg p$.

Let β be the first cut on the path $\mathcal{P}$ such that $inf(p) \subseteq \beta$. Let β' be the predecessor of β on the path $\mathcal{P}$. From Lemma 9, we have that the meet of β' and $inf(p)$ is either $inf(p)$ or a predecessor of $inf(p)$, say C'. However, if the meet is $inf(p)$ then $\mathrm{inf}(p) \subseteq \beta'$. Since β' is also on the path $\mathcal{P}$ we have that β' is the first cut on the path $\mathcal{P}$ such that $inf(p) \subseteq \beta$. This is a contradiction since β is the first such cut. Therefore the meet of β' and $inf(p)$ is C'.

There exists a path from C' to β' because C' is the meet of β' and $inf(p)$ and therefore $C' \subseteq \beta'$. Furthermore every cut on this path satisfies $\neg p$. We prove this as follows. From the definition of $interval(C, D)$, only cuts in $interval(inf(p), sup(p))$ satisfy p. Now consider all cuts F such that $C' \subseteq F \subseteq \beta'$. We have that $inf(p) \not\subseteq \beta'$ and therefore $inf(p) \not\subseteq F$. Therefore, β' and all such F do not satisfy p. Since C is the meet of all predecessors of $inf(p)$ and C' is a predecessor of $inf(p)$, $C \subseteq C'$ and therefore $C \subseteq F$. Also, all cuts from C to C' satisfy $\neg p$ since none of them belong to $interval(inf(p), sup(p))$.

We obtain a path in $interval(C, E)$ where all cuts on the path satisfy $\neg p$ as follows. Choose an arbitrary path from C to C', then continue the path from C' to β' and then to β. Continue the path from β to the final cut with the same path from β to the final cut as in path $\mathcal{P}$.

Step 2: Now we show that if there exists a path, $\mathcal{P}$, from the initial to the final cut in $interval(C, E)$ where all cuts on the path satisfy $\neg p$ then there exists a path from the initial to the final cut in $interval(C, D)$ where all cuts on the path satisfy $\neg p$.

The proof is similar to Step 1 with the paths reversed. In this case we choose β as the last cut on the path $\mathcal{P}$ such that $\beta \subseteq sup(p)$ and β' as the successor of β on the path $\mathcal{P}$. Furthermore, we choose D' as a successor of $inf(p)$. We can show in a similar fashion as in Step 1 that there exists a path from β' to D' where all cuts on the path satisfy $\neg p$. Finally, we can construct a path from C to D as the concatenation of the paths from C to β, β to β', β' to D', and D' to D. ■

We can compute $interval(C, D)$ by computing $inf(p)$ and $sup(p)$ in $O(n|E|)$ time for regular p [8]. We can compute the predecessors and successors of a cut in $O(n|E|)$ time as well. Note that the above theorem is not restricted to predicates with a single least and greatest cut only. For example, if the predicate has several least cuts then first we take the intersection of all those cuts, second, we find the predecessors of the intersection, and finally, we compute the intersection of the predecessors to obtain C.

Although the time complexity of computing $interval(C, D)$ is polynomial, the time and space complexity of detecting *definitely*:p on this reduced state space may be exponential since $interval(C, D)$ may contain exponential number of global states. However, it is always better to work on $interval(C, D)$ rather than $\langle E, \rightarrow \rangle$ since $interval(C, D)$ is a subset of $\langle E, \rightarrow \rangle$. In fact, we believe that $interval(C, D)$ is generally much smaller than the original computation $\langle E, \rightarrow \rangle$. Furthermore, Theorem 1 is orthogonal to the conditions we present for detecting *definitely*:p, that is, we can always first compute $interval(C, D)$ and then apply those conditions.

Next, we present a polynomial space algorithm for *definitely*:p. Cooper and Marzullo presented a worst case exponential space and time algorithm when they introduced *definitely*:p. Their algorithm [4] detects *definitely*:p using level sets where a *level set* is the set of successors of a consistent cut. The algorithm starts from the initial consistent cut. If p is true in the initial consistent cut we are done. Otherwise, it constructs the next level set including only those consistent cuts in which $\neg p$ is true. Continuing in this manner, if the algorithm can reach the final consistent cut, then *definitely*:p is false; otherwise, it is true. This algorithm requires space proportional to the size of the largest level set, which is exponential. We obtain a simple space efficient algorithm for detecting *definitely*:p by generating all paths of cuts for the given computation. This algorithm is based on generating linearizations of a partial order [16]. For each such path, we check whether $\neg p$ holds on every cut on the path. If such a path exists then *definitely*:p is not satisfied otherwise it is satisfied. The length of every path is at most $|E|$, the total number of events in the system. A frontier of a consistent cut can be represented by an n-dimensional vector. Therefore, for each consistent cut $O(n)$ space is required giving us the space complexity of $O(n|E|)$. The time complexity is bounded by the number of paths, which may be exponential in the number of processes. We will improve the time complexity using computation slicing technique later in this paper.

Figure 8 shows a polynomial space *definitely*:p algorithm that uses the techniques developed in this section.

5 Polynomial Time Necessary Conditions

Now we present a polynomial time necessary condition to detect *definitely*:p that uses meet-irreducible cuts [5]. We say that a cut is *meet-irreducible* if it has only one successor consistent cut. For example, the predecessors of the final cut of a computation (e.g. predecessors of the final cut $\{e_3, f_3\}$ in Fig. 2(b)) are all meet-irreducible cuts. The number of meet-irreducible cuts of a distributive lattice is generally exponentially smaller than the number of all cuts in the lattice. In fact, for a finite distributive lattice, the number of meet-irreducible cuts is exactly equal to the size of the longest chain in the lattice [5]. In our case, the length of the longest chain is equal to the number of events $|E|$. Hence, if some computation can be done on meet-irreducible cuts, we get a significant computational advantage.

```
Input:   A computation ⟨E, →⟩ and a predicate p
Output:  definitely : p is satisfied or not

compute inf(p) and sup(p);
let C = intersection of predecessors of inf(p);
let D = union of successors of sup(p);
use C and D to obtain interval(C, D);          // reduce the number of global states
for each path in interval(C, D) do             // obtain paths using [16]
    let G be the first cut on the path
    while G satisfies ¬p do
        G := successor of G on the path;
    endwhile;
    if G = D then                              // final cut is reached
        return false;
    endif;
endfor;
return true;
```

Figure 1: A polynomial space algorithm for detecting *definitely*$: p$

Theorem 2 (NC) *Given a computation $\langle E, \rightarrow \rangle$ and a regular predicate p, if $\neg p$ holds at the initial consistent cut and at the successor of every meet-irreducible cut then definitely$: p$ does not hold in $\langle E, \rightarrow \rangle$.*

Proof: We show that there exists a path from the initial to the final consistent cut in the computation $\langle E, \rightarrow \rangle$ where all cuts on the path satisfy $\neg p$. Given an arbitrary cut C that satisfies $\neg p$ and different from the final consistent cut, we first show that there exists a successor of C that satisfies $\neg p$. There are two cases.

Case 1: C has a single successor. In this case C is a meet-irreducible cut and therefore $\neg p$ holds at the successor of C.

Case 2: C has at least two successors. Observe that if more than one successor of C satisfies p then from the regularity of p, the intersection of those successor cuts, which is C, satisfies p. This leads to a contradiction. Therefore, there exists at least one successor of C where $\neg p$ holds.

We construct the path as follows: From the assumption we have that $\neg p$ holds at the initial cut. From above we have that for every cut that satisfies $\neg p$ we can find a successor cut that satisfies $\neg p$. Finally, we reach the final cut which is the successor of a cut that satisfies $\neg p$. ■

The converse of Theorem 2 is false. Figure 2(c) displays the lattice of consistent cuts of the computation in Fig. 2(a). From the lattice we observe that this computation satisfies the right side of Theorem 2. However, the left side of the theorem does not hold because the successor of the meet-irreducible cut $\{f_3\}$ satisfies p. A similar condition can be given for join-irreducible cuts. A *join-irreducible* cut of a distributive lattice is such that it has only one predecessor consistent cut. Meet and join-irreducible cuts are duals of each other.

Theorem 3 *Given a computation $\langle E, \rightarrow \rangle$ and a regular predicate p, if $\neg p$ holds at the final consistent cut and at the predecessor of every join-irreducible cut then definitely$: p$ does not hold in $\langle E, \rightarrow \rangle$.*

We can check Theorem 2 and 3 by finding the meet-irreducible (resp. join-irreducible) cuts of the computation in $O(n^2|E|)$ time for p regular [18].

Next we present another polynomial time necessary condition for detecting *definitely*$: p$ based on the notion of intervals introduced earlier. We say that a predicate is an *interval predicate* if there exists

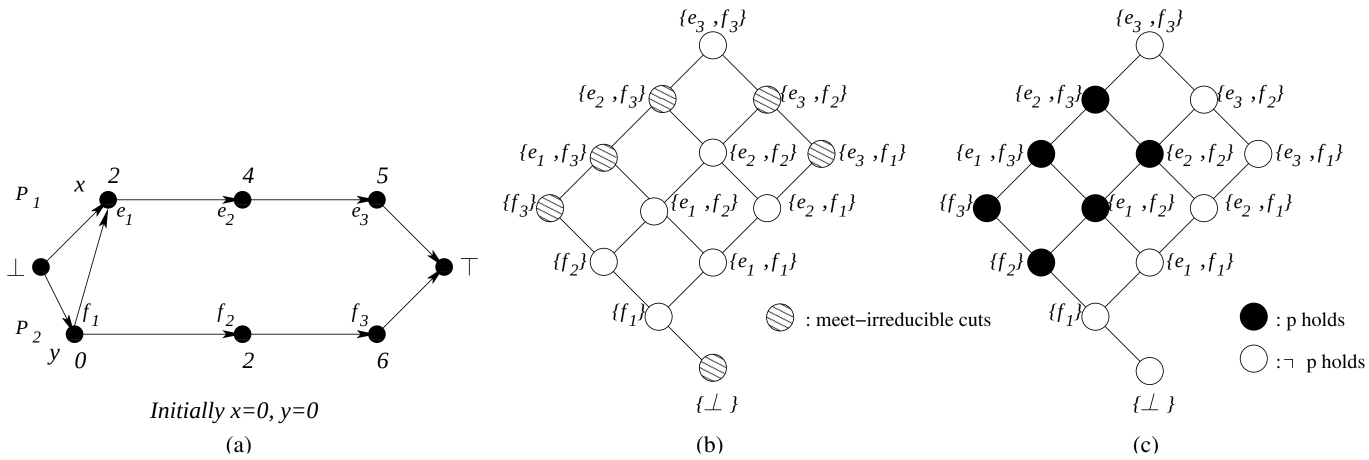

Figure 2: (a) A computation (b) meet-irreducible cuts (c) corresponding lattice of the computation

a unique initial cut, C, and a unique final cut, D, that satisfies the predicate and the predicate holds in *all* cuts between C and D. An interval predicate with the initial and final cuts C and D defines an $interval(C, D)$ in a computation $\langle E, \rightarrow\rangle$. Observe that $interval(C, D)$ may partition the lattice of consistent cuts of $\langle E, \rightarrow\rangle$ as in Figure 3. The patterned region in the figure denotes the cuts that belong to $interval(C, D)$, i.e., the set of cuts that satisfy the interval predicate. A cut F belongs to partition I if $C \not\subseteq F \subseteq D$, partition II if $C \not\subseteq F \not\subseteq D$, partition III if $C \subseteq F \subseteq D$, and partition IV if $C \subseteq F \not\subseteq D$. Given that $interval(C, D)$, that is, partition III exists, other partitions may not exist. For example, if C is the initial cut of $\langle E, \rightarrow\rangle$ and D is the final cut of $\langle E, \rightarrow\rangle$ then only partition III exists.

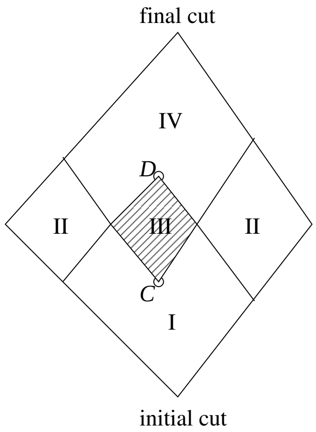

Figure 3: $interval(C, D)$ partitions the lattice of consistent cuts

Theorem 4 *Given a computation $\langle E, \rightarrow\rangle$ and an interval predicate p with $interval(C, D)$, there exists a cut F that belongs to partition II in $\langle E, \rightarrow\rangle$ iff definitely: p does not hold in $\langle E, \rightarrow\rangle$.*

Proof:

$\Rightarrow$:

If there exists a cut F that belongs to partition II then we obtain a path where all cuts on the path satisfies $\neg p$ in $\langle E, \rightarrow\rangle$ as follows. We know that F is reachable from the initial cut. Furthermore, there does not exist a cut H on a path from $\perp$ to F such that H satisfies p since all such cuts belong

to $interval(C, D)$. For the purpose of contradiction, assume that such a cut H exists. For F to be reachable from H, we must have that $H \subseteq F$. However since H satisfies p, $C \subseteq H$ and since F is in partition II, $C \not\subseteq F$, therefore we have a contradiction. C cannot be $\perp$ and D cannot be E because we assume that partition II exists. Therefore, partitions I and IV also exist. Now we obtain a path where all cuts satisfy $\neg p$ by starting from $\perp$ and following an arbitrary path in partition I such that the path reaches F in partition II. Then we follow an argument similar to the argument for cut H to show that we can construct a path from F to the final cut where all the cuts on the path satisfy $\neg p$.
$\Leftarrow$:
We prove by contradiction. Suppose that partition II does not exist and there exists a path in $\langle E, \rightarrow \rangle$ from initial to the final cut where all cuts on the path satisfy $\neg p$. Since there exists such a path, we have that partitions I and IV exist. Otherwise, $C = \perp$ and $D = E$ and we do not have a path from $\perp$ to E where $\neg p$ holds on the path. Since partition II does not exist and a path of cuts satisfying $\neg p$ exists, we must have that there is a path from partition I to partition IV without passing through partition III (since p is an interval predicate). We will show that this is impossible.

Consider two cuts, F and H, on a path from $\perp$ to E where $\neg p$ holds on the path, such that F belongs to partition I and H belongs to partition IV and H is a successor of F. From the definition of partitions, we have that $C \not\subseteq F \subseteq D$ and $C \subseteq H \not\subseteq D$. Furthermore, from the definition of successor of a cut, we know that $H = F \cup \{e\}$, where e is an event in $\langle E, \rightarrow \rangle$ and $e \notin F$. To obtain H from F, there are two cases: On one hand, we should add $e \notin D$ to F (therefore $e \notin C$) so that $H \not\subseteq D$. On the other hand, we should add $e \in C$ to F (therefore in $e \in D$) so that $C \subseteq H$. However, $x \in D$ and $x \notin D$ leads to a contradiction. ■

We present a weaker result for regular predicates. The necessary conditions of Theorem 2 and 3 are not comparable with the condition of Theorem 5 below. Furthermore, observe that the converse of the next condition is false.

Theorem 5 *Given a computation $\langle E, \rightarrow \rangle$, and a regular predicate p with $interval(C, D)$, where $C = inf(p)$ and $D = sup(p)$, if there exists a cut F that belongs to partition II in $\langle E, \rightarrow \rangle$ then definitely:p does not hold in $\langle E, \rightarrow \rangle$.*

We can use a technique called slicing, which we explain next, to detect whether there exists a cut F in partition II. The overall complexity of checking the existence of F using slicing is $O(n^2|E|^2)$ [15].

6 Polynomial Time Sufficient Condition

We have advocated the use of a technique called *computation slicing* for predicate detection in [8, 15, 17]. The notion of computation slice is based on the Birkhoff's Representation Theorem for Finite Distributive Lattices [5]. The readers who are not familiar with earlier papers on slicing [8, 15, 17] are strongly urged to read Appendix A. We also present a directed graph model of a computation in the Appendix to handle both computations and computation slices in a uniform and convenient manner. In this model, a distributed computation $\langle E, \rightarrow \rangle$ is a directed graph with vertices as the set of events and edges as $\rightarrow$. A subset of vertices forms a consistent cut if the subset contains a vertex only if it also contains all its incoming neighbours. Observe that a consistent cut either contains all vertices in a strongly connected component or none of them. 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 [15]) *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 p by $\mathsf{slice}(\langle E, \rightarrow \rangle, p)$. We prove in [15] that the slice exists and is uniquely defined for all predicates. Intuitively, the consistent cuts that belong to the slice are obtained by computing the union and intersection closure of the cuts in the computation that satisfy the predicate. In other words, if two cuts G and H satisfy p, then the slice contains cuts $G \sqcup H$ and $G \sqcap H$.

Given a computation as in Fig. 4(a), and a regular predicate p, such as $(x = 4)$, where x is a local variable defined on process P_1, now we consider the slice of the computation with respect to $\neg p$ as displayed in Fig. 4(b). The consistent cuts that belong to the slice are denoted by white filled circles in Fig. 4(c). Note that in this example the cuts that belong to the slice are already closed under union and intersection. We make the following two observations on the computation and its slice. First, consider the cuts in the computation. On every path from the initial to the final consistent cut there is a consistent cut that contains event e_2 but not e_3. These cuts are $\{e_2, f_1\}, \{e_2, f_2\}, \{e_2, f_3\}$ and they satisfy p. Furthermore, all of these cuts satisfy p. Second, consider the cuts in the slice, when the slice contains a non-trivial strongly connected component, such as $\{e_2, e_3\}$ in Fig. 4 (b), then none of the cuts of the original computation that contain a single element from this component belongs to the slice. For example, cuts that contain only e_2 but not e_3 do not belong to the slice. This is because of the definition of a consistent cut in a directed graph.

From above two observations, we have that if the slice contains a non-trivial strongly connected component then in the computation on every path from the initial to the final consistent cut there exists a consistent cut that satisfies p which does not belong to the slice. Therefore, *definitely*$: p$ holds. We can use these observations to state a sufficient condition for detecting *definitely*$: p$.

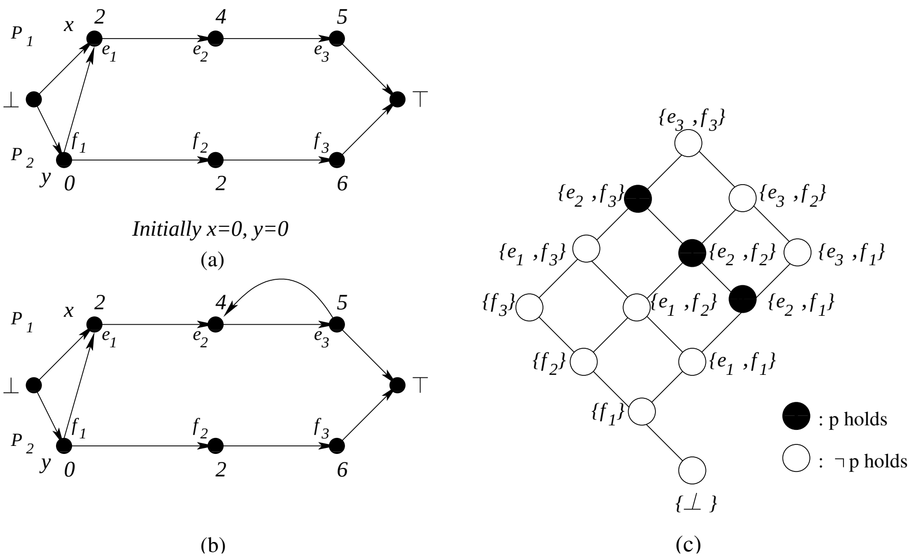

Figure 4: (a) A computation (b) slice wrt $x \neq 4$ (c) lattice of consistent cuts of the computation

Theorem 6 (SC) *Given a computation $\langle E, \rightarrow \rangle$ and a regular predicate p, if slice$(\langle E, \rightarrow \rangle, \neg p)$ contains a non-trivial strongly connected component then definitely$: p$ holds in $\langle E, \rightarrow \rangle$.*

We can check this condition by finding the slice in $O(n^2|E|^2)$ time [15] and then checking the strongly connected components of the slice in $O(n|E|)$ time [15].

The converse of Theorem 6 is false. Figure 5(a) displays a computation that satisfies *definitely*$: p$. When we compute the union and intersection closure of the cuts that satisfy the predicate (the closure of white filled circles), the set of consistent cuts that belongs to the slice and the set of consistent cuts that belongs to the computation are the same, that is, $slice(\langle E, \rightarrow\rangle, \neg p)$ is the same as $\langle E, \rightarrow\rangle$. Therefore, the slice does not contain a non-trivial strongly connected component not in $\langle E, \rightarrow\rangle$.

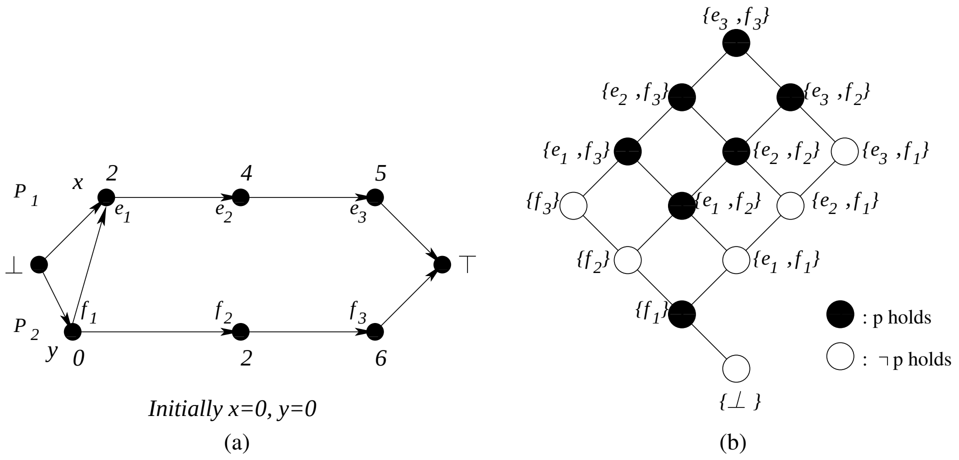

Figure 5: (a) A computation (b) corresponding lattice

We can use the slice with respect to $\neg p$ to obtain a smaller number of linearizations for the first polynomial-space algorithm explained in Section 4.

7 Experimental Results

We implemented the conditions in this paper in Java and applied it to a leader election protocol. The leader election program implements the Chang-Roberts algorithm where processes are arranged in a unidirectional ring. We check $definitely: (done_0 \wedge done_1 \wedge \ldots \wedge done_{n-1})$ which denotes that eventually a leader is chosen by every process. In order to evaluate the effectiveness of our conditions, we compare our approach with a partial order reduction based model checker SPIN [10]. For this purpose, we implemented a translator from distributed computations to Promela (input language of SPIN). We restricted the memory usage to 256MB. We manually instrumented the leader election procotol to generate computations. The computations are obtained by running the program for 20 seconds. Our results are shown in Table 1. The column labeled by NSC+SC+NC denotes experiments performed by applying all three Theorems 1, 2, 6. Observe that our improvement in space and time performance is in the order of magnitude. We are continuing experiments. Due to lack of space and time these results are not reported in the current version of the paper.

8 Conclusion

We presented efficient space and time algorithms for detecting $definitely: p$. Earlier, we developed polynomial time detection algorithms for predicates from a subset of the temporal logic CTL [3] that did not include definitely modality. We can enlarge this subset by the results of this paper.

References

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

[2] C. Chase and V. K. Garg. Detection of Global Predicates: Techniques and their Limitations. *Distributed Computing*, 11(4):191–201, 1998.

Table 1: Leader Election *definitely*: $(done_0 \wedge \ldots \wedge done_{n-1})$

	SPIN			NSC+SC+NC		
n	No. States	Time (sec)	Memory (MB)	No. States	Time (sec)	Memory (MB)
3	191	0.01	1.57	21	0.01	1.57
4	545	0.01	1.57	105	0.01	1.57
5	1,113	0.01	1.57	113	0.01	1.57
6	6,154	0.07	1.77	545	0.01	1.57
7	17,253	0.22	2.29	1,217	0.01	1.57
8	100,888	1.26	6.38	2,689	0.03	1.57
9	549,740	8.91	34.5	5.889	0.06	1.77
10	395,747	6.05	25.33	19,457	0.19	2.29
11	*			53,121	0.58	3.82
12	*			28,673	0.31	2.8
13	*			126,977	1.38	7.1
14	*			505,857	6.47	28.3
15	*			1,458,180	7.11	29.01
16	*			1,835,010	25.9	96.9
17	*			32,111,260	54.55	181.38

*: denotes out of memory

[3] 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.

[4] 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.

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

[6] E. Fromentin and M. Raynal. Inevitable global states: A concept to detect unstable properties of distributed computations in an observer independent way. In *Proceedings of the 6th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 242–248, Los Alamitos, CA, USA, 1994.

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

[8] 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.

[9] 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.

[10] G.J. Holzmann. The model checker spin. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997. Special issue on Formal Methods in Software Practice.

[11] 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.

[12] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal. Efficient detection of conjuntions of local predicates. *IEEE Transactions on Software Engineering*, 24(8):664–677, 1998.

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

[14] 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.

[15] 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.

[16] G. Preusse and F. Ruskey. Generating linear extensions fast. *SIAM J. Comput., 23, 373-386*, 1994.

[17] A. Sen and V. K. Garg. Automatic Generation of Computation Slices for Detecting Temporal Logic Predicates. Technical Report TR-PDS-2003-001, PDSL, ECE Dept. Univ. of Texas at Austin, 2001. Available at http://maple.ece.utexas.edu/TR-PDS-2003-001.ps.gz.

[18] A. Sen and V. K. Garg. Detecting Temporal Logic Predicates on the Happened-Before Model. In *Proc. of the International Parallel and Distributed Processing Symposium (IPDPS)*, Fort Lauderdale, Florida, 2002.

[19] 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.

[20] 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.

[21] 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.

A Notation and Background

Traditionally, a distributed computation is modeled as a partial order on a set of events [13]. 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 [13]. 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 $e \overset{P}{\rightarrow} f$, and let $\overset{P}{\underline{\rightarrow}}$ be the reflexive closure of $\overset{P}{\rightarrow}$. 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. In the rest of the paper, unless otherwise stated, a consistent cut refers to a non-trivial consistent cut. Note that the empty consistent cut, $\emptyset$, in the traditional model corresponds to $\{\bot\}$ in our model and the final consistent cut, E, in the traditional model corresponds to $E - \{\top\}$ in our model and we denote this by E. We use uppercase letters C and D to represent consistent cuts. 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, $frontier(C) \triangleq \{ e \in C \mid e \notin \top \Rightarrow succ(e) \notin C \}$. Figure 2 shows a computation and its lattice of (non-trivial) consistent cuts. A consistent cut in the figure is represented by its frontier. For example, the consistent cut $C = \{e_3, e_2, e_1, f_2, f_1, \bot\}$ is represented by $\{e_3, f_2\}$.

We define *successor* of a cut by a relation $\triangleright \subseteq \mathcal{C}(G) \times \mathcal{C}(G)$ such that $C \triangleright D$ if and only if $D = C \cup e$, where e is the set of vertices in some strongly connected component in $\langle E, \rightarrow \rangle$ and $e \notin C$. We denote the reflexive closure of this relation by $\trianglerighteq$. A *consistent cut sequence* $C_0, C_1, \ldots, C_k$ of $(\mathcal{C}(G), \subseteq)$ satisfies that for each $0 \le i < k$, $C_i \triangleright C_{i+1}$. We say that a cut D is *reachable* from a cut C if $C \subseteq D$.

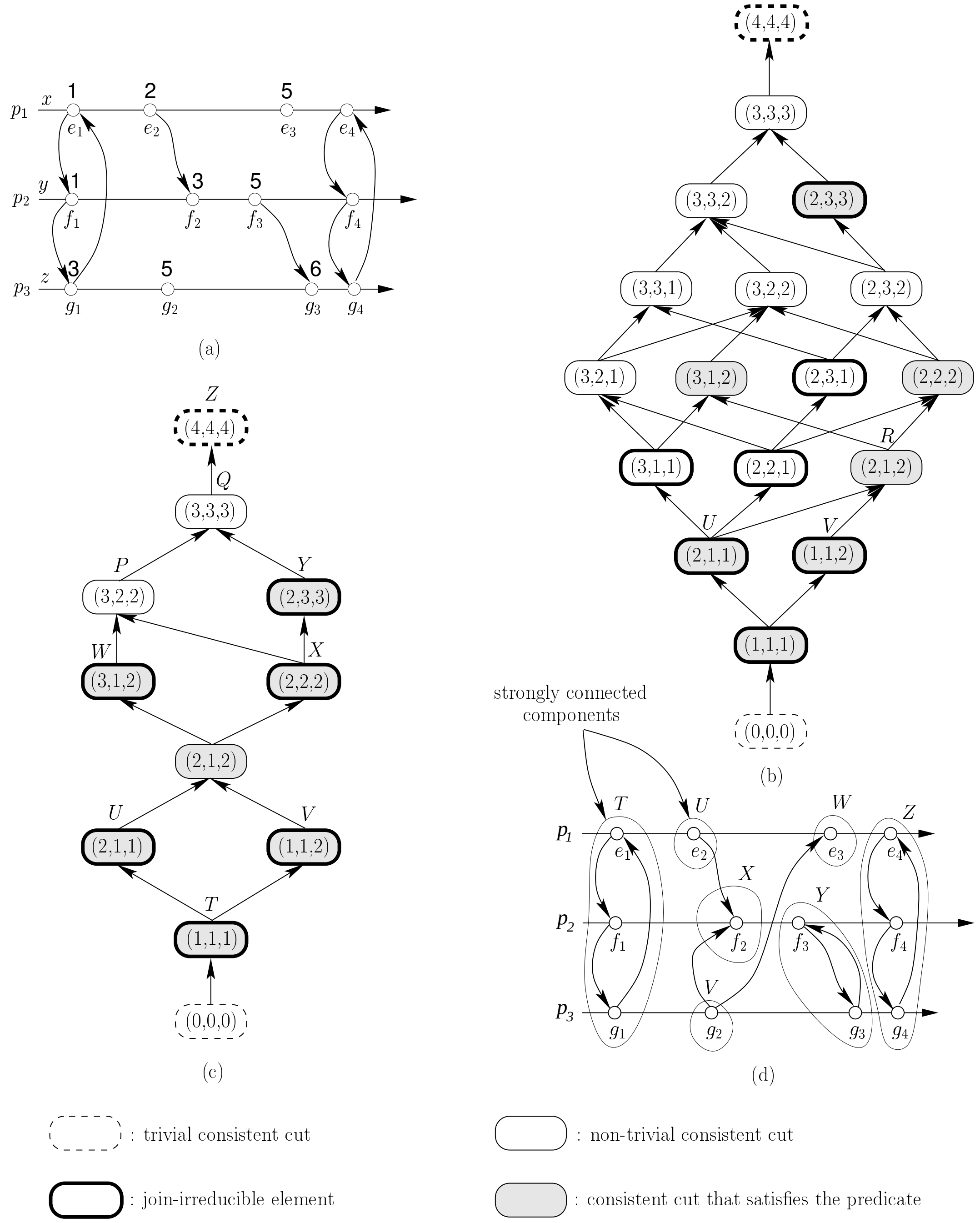

Figure 6: (a) A computation, (b) the lattice of its consistent cuts, (c) the smallest sublattice that contains all consistent cuts satisfying the predicate $x + y - z \leq 1$, and (d) the poset induced on the set of join-irreducible elements of the sublattice.

Example 1 Consider the computation depicted in Figure 6(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 $x \leq 1$ is a local predicate whereas the predicate $x + y \le z$ is not. The consistent cut $\{e_1, f_1, g_1\}$ satisfies the predicate $x + y \le z$ 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 [5]. 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 [5].

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 [5, 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 $\langle E, \rightarrow \rangle$ 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 $\langle E, \rightarrow \rangle$.

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 [5]. 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 6(a). The (distributive) lattice spanned by its set of consistent cuts is shown in Figure 6(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 6(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 $\langle E, \rightarrow \rangle$ and a predicate b. Let $\mathcal{C}(E)$ denote the set of consistent cuts of $\langle E, \rightarrow \rangle$ 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 $\langle E, \rightarrow \rangle$ 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 6(b). The consistent that satisfy the predicate $x + y - z \leq 1$ have been shaded in the figure. Figure 6(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 6(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 6(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.

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 7 ([15]) *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 8 ([15]) *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 [15]. 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.

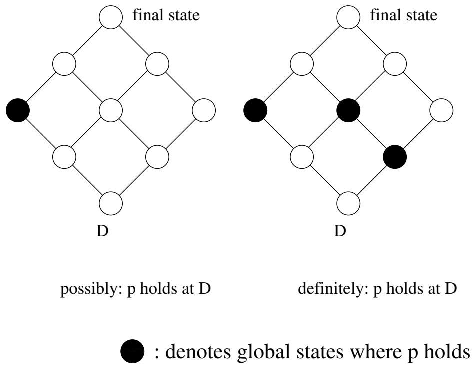

Figure 7: Predicates illustrated on a lattice of global states

B Omitted Proofs

Lemma 9 *Given a finite distributive lattice $L = (C(E), \subseteq)$ for a computation $(E, \rightarrow)$ and two consistent cuts $G, H \in C(E)$ such that $G \triangleright H$ and another consistent cut $J \in C(E)$ such that $J \subseteq H$ then either $(G \sqcap J) = J$ or $(G \sqcap J) \triangleright J$.*

Proof: We have the following cases for J:

- Case 1: $J = G$, $(G \sqcap J) = J$ trivially .
- Case 2: $J = H$, $(G \sqcap J) = G$ and $G \triangleright H$ imply $(G \sqcap J) \triangleright J$.
- Case 3: $J \neq H$ and $J \neq G$,
  1. $J \subseteq H$, given
  2. $J = J \cap H$, from 1
  3. $H = (G \cup \{x\})$, given $G \triangleright H$, $x \in E$
  4. $J = J \cap (G \cup \{x\})$, from 2 and 3
  5. $J = (J \cap G) \cup (J \cap \{x\})$, distributivity of $\cap$ on $\cup$

  There are two cases for $(J \cap \{x\})$.

  - Case 3.1: $x \notin J$.
    6. $J = (J \cap G) \cup \emptyset$, from 5 and $x \notin J$
    7. $J = (J \cap G)$, from 6
    8. $J = (J \sqcap G)$, $\cap$ and $\sqcap$ are same for elements of L
  - Case 3.2: $x \in J$.
    9. $J = (J \cap G) \cup \{x\}$, from 5 and $x \in J$
    10. $J = (J \sqcap G) \cup \{x\}$, $\cap$ and $\sqcap$ are same for elements of L
    11. $(J \sqcap G) \triangleright J$, from 10 and definition of $\triangleright$

■

C A General Algorithm

```
Input:   A computation ⟨E, →⟩ and a regular predicate p
Output:  definitely : p is satisfied or not

        // reduce the state space
    compute inf(p) and sup(p);
    let C := intersection of predecessors of inf(p);
    let D := union of successors of sup(p);
    use C and D to obtain K := interval(C, D);        // reduce the number of global states

        // apply polynomial time necessary condition
    let SMI := the set of Sucessors of all Meet-Irreducible elements in K;
    if initial cut of K and all cuts in SMI satisfy ¬p then
        return false;
    endif;

        // apply polynomial time sufficient condition
    compute slice(K, ¬p);
    if slice(K, ¬p) contains a non-trivial strongly connected component then
        return true;
    endif;

        // apply polynomial space algorithm
    for each path in interval(C, D) do                // obtain paths using [16]
        let G be the first cut on the path
        while G satisfies ¬p do
            G := successor of G on the path;
        endwhile;
        if G = D then                                 // final cut is reached
            return false;
        endif;
    endfor;
    return true;
```

Figure 8: A general algorithm for detecting *definitely* : p for regular p