# 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 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 some 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 [10, 14, 1, 2, 29]. 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 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 an 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 [13].*

## 1. Introduction

A fundamental problem in distributed computing is *predicate detection*—deciding whether an execution trace of a distributed program satisfies a given predicate. 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 some 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 [10, 14, 1, 2, 29].

Cooper and Marzullo present an algorithm for detecting $definitely : p$ for arbitrary predicate $p$. The worst-case space and time complexity of the their algorithm is 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 sat-

*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

isfies it. It is, in general, coNP-complete to detect a predicate under $definitely$ modality [27]. 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. We validate the effectiveness of our algorithms with experimental studies in Section 7. For this purpose, we implement our algorithms in the Partial Order Trace Analyzer (POTA) tool [26] and compare performance to partial order reduction based algorithms of model checker SPIN [13]. In one case, our algorithms are significantly faster and space efficient. We have measured over 100-fold gain.

Our work constitutes part of the POTA tool [26, 22] for testing distributed program execution traces using temporal logic predicates. Figure 1 displays an overview of POTA architecture. POTA consists of an instrumentation module, a translator module that translates execution traces into Promela [13] (SPIN model checker input language) and an analyzer module. The use of partial order model for execution traces and the use of an effective abstraction technique for temporal logic verification called computation slicing are significant aspects of POTA and constitutes the analyzer module. POTA implements polynomial-time temporal logic predicate detection algorithms. The temporal logic used in POTA is a subset of CTL [3]. With the results of this paper, we extend efficient predicate detection algorithms in POTA for $definitely$ operator. Atomic propositions of the logic used in POTA are regular predicates, which widely occur in practice during verification. Some examples of regular predicates are conjunction of local predicates [8, 15] such as "all processes are in *red* state", certain channel predicates [8] such as "at most $k$ messages are in transit from process $P_i$ to $P_j$", and some relational predicates [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 [10, 14], observer-independent [1], linear [2], and relational predicates [29]. Also in [21] there is an extensive survey on predicate detection techniques.

Tarafdar and Garg [27] proved that it is, in general, NP-complete to detect a predicate under $controllable$ modality. A computation satisfies $controllable: p$ if every state on some path from the initial global state to the final global state satisfies $p$. 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 [28] 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 [7] 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 [8]. 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.

The idea of using temporal logic in program testing has been applied in several tools such as the commercial Temporal Rover tool (TR) [6], the MaC tool [16], and the JPaX tool [12]. TR allows the user to specify the temporal formula in programs. These temporal formula are translated into Java code before compilation. The MaC and JPaX tools consider a totally ordered view of an execution trace and therefore can potentially miss bugs that can be deduced from a partial order view of the trace. Hallal et al. in [11] uses a partial order view of an execution trace as in POTA. They translate execution traces into SDL and use commercial SDL tools for testing translated traces. POTA incorporates several polynomial-time (polynomial in the number of processes) predicate detection algorithms whereas the complexity is exponential-time in [11].

## 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, $\rightarrow$ [17]. 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

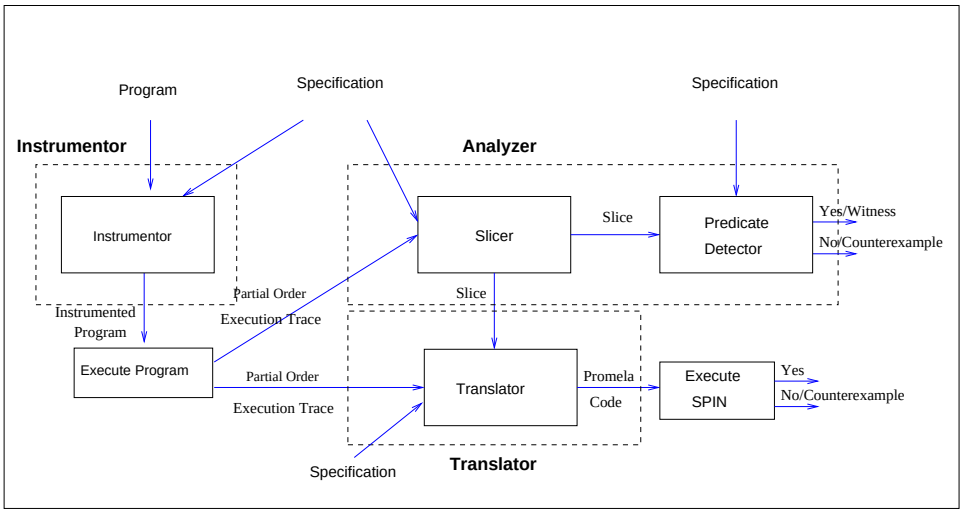


**Figure 1. Overview of POTA Architecture**

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 $\subseteq$ [18, 9]. 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 consistent cut, $E$ to denote the final consistent cut of all processes, and $\top$ to denote a fictitious final cut occurring after $E$.

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 $0 \leq i < l$, $G_i \triangleright G_{i+1}$.

A predicate is defined as a boolean-valued function on variables of processes. Given a consistent cut, a predicate is evaluated with respect to the values of variables resulting after executing all events in the cut. If a predicate $p$ evaluates to true for a consistent cut $C$, we say that "$C$ satisfies $p$". We leave the predicate undefined for $\top$. A global predicate is *local* if it depends on variables of a single process.

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 consistent 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. There are efficient algorithms for detecting regular predicates under $possibly$ and $controllable$ modalities [9, 24].

## 4. Polynomial-Space Algorithm

The performance of algorithms for detecting $definitely\colon p$ can be improved by considering a smaller state-space, that is, a smaller computation than the original computation. In this section, we present a polynomial-time algorithm for reducing the size of the computation. We show that detecting $definitely\colon p$ on the original computation is the same as detecting $definitely\colon p$ on the smaller computation. For this purpose, we define an $interval$ of a computation $\langle E, \rightarrow \rangle$ with respect to consistent cuts $C$ and $D$ as the computation $interval(C, D)$, which is a subset of $\langle E, \rightarrow \rangle$ and only the cuts between

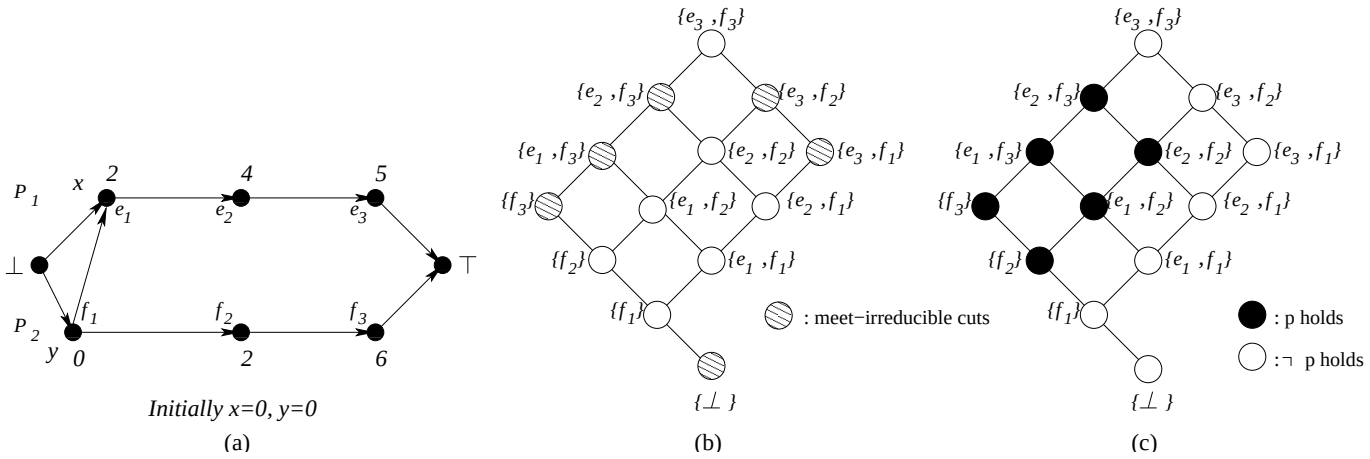


**Figure 2. (a) A computation (b) meet-irreducible cuts (c) corresponding lattice of the computation**

$C$ and $D$ (including $C$ and $D$) of $\langle E, \rightarrow \rangle$ belong to $interval(C, D)$.

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 extended version of this paper [25].

**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 the predecessors exist, otherwise $inf(p)$, and $D$ is the join of successors of $sup(p)$, if the successors exist, otherwise $sup(p)$.*

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

$\Rightarrow$:

We obtain a path from the initial consistent cut to the final consistent cut in $\langle E, \rightarrow \rangle$ as follows: Pick an arbitrary path from the initial consistent 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 consistent 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 consistent 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 consistent 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 $\beta$ be the first cut on the path $\mathcal{P}$ such that $inf(p) \subseteq \beta$. Let $\beta'$ be the predecessor of $\beta$ on the path $\mathcal{P}$. From the lemma stated above, the meet of $\beta'$ and $inf(p)$ is either $inf(p)$ or a predecessor of $inf(p)$, say $C'$. However, if the meet is $inf(p)$ then $\inf(p) \subseteq \beta'$. Since $\beta'$ is also on the path $\mathcal{P}$ we have that $\beta'$ is the first cut on the path $\mathcal{P}$ such that $inf(p) \subseteq \beta$. This is a contradiction since $\beta$ is the first such cut. Therefore the meet of $\beta'$ and $inf(p)$ is $C'$.

There exists a path from $C'$ to $\beta'$ because $C' \subseteq \beta'$. 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, $\beta'$ 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 the required path as follows. Choose an arbitrary path from $C$ to $C'$, then continue the path from $C'$ to $\beta'$ and then to $\beta$. Continue the path from $\beta$ to the final cut with the same path from $\beta$ 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 consistent 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 consistent 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 $\beta$ as the last cut on the path $\mathcal{P}$ such that $\beta \subseteq sup(p)$ and $\beta'$ as the successor of $\beta$ 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 $\beta'$ 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 $\beta$, $\beta$ to $\beta'$, $\beta'$ 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 $p$ [9]. Similarly, we can compute the predecessors and successors of a cut in $O(n)$ time. 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 $\langle E, \rightarrow \rangle$ and we validate this belief with experimental work. Furthermore, Theorem 1 is orthogonal to the conditions we will 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 [4] presented a worst case exponential-space and time algorithm when they introduced $definitely: p$. Their algorithm 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 [20]. 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 can improve the time complexity using computation slicing technique explained later in this paper.

Figure 3 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 consistent cut of a computation (e.g. predecessors of the final cut $\{e_3, f_3\}$ in Figure 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.

**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 consistent 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 from the assumption $\neg p$ 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, $\neg p$ holds at the initial cut. From above we have that for every consistent cut that satisfies $\neg p$ we can find a successor consistent cut that satisfies $\neg p$. Finally, we reach the final consistent 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 Figure 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*

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

1.  compute inf(p) and sup(p);
2.  let C be the intersection of predecessors of inf(p);
3.  let D be the union of successors of sup(p);
4.  use C and D to obtain interval(C, D);          // reduce the number of global states
5.  for each path in interval(C, D) do             // obtain paths using [20]
6.      let G be the first cut on the path
7.      while G satisfies ¬p do
8.          G := successor of G on the path;
9.      endwhile;
10.     if G = D then                              // final cut is reached
11.         return false;
12.     endif;
13. endfor;
14. return true;
```

**Figure 3. A polynomial-space algorithm for detecting $definitely: p$**

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 (resp. Theorem 3) by finding the meet-irreducible (resp. join-irreducible) cuts of the computation in $O(n^2|E|)$ time for regular $p$ [23].

Next we present another polynomial-time 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 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 a computation as in Figure 4. 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 $interval(C, D)$ exists, that is, partition III exists, other partitions may not exist. For example, if $C$ is the initial consistent cut of $\langle E, \rightarrow\rangle$ and $D$ is the final consistent cut of $\langle E, \rightarrow\rangle$ then only partition III exists.

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

*Proof:*

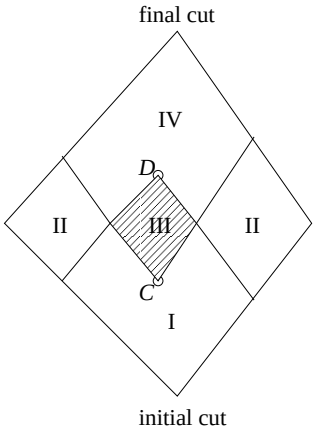


**Figure 4. $interval(C, D)$ partitions the lattice of consistent cuts**

$\Rightarrow$:
We know that $F$ is reachable from the initial cut. For the purpose of contradiction, assume that there exists a cut $H$ on a path from $\perp$ to $F$ such that $H$ satisfies $p$ . 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. Similarly, we can show that there does not exist a cut $H'$ on a path from $F$ to $E$ such that $H'$ satisfies $p$. $C$ 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 arbitrary path from $F$ to the final consistent cut.

$\Leftarrow$:

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 consistent 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, 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, $e \in D$ and $e \notin D$ 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 consistent 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 consistent cut $F$ in partition II. The overall complexity of checking the existence of $F$ using slicing is $O(n^2|E|^2)$ [19].

## 6. Polynomial-Time Sufficient Condition

We have advocated the use of a technique called *computation slicing* for predicate detection in [9, 19, 24]. The notion of computation slice is based on Birkhoff's Representation Theorem for Finite Distributive Lattices [5]. The readers who are not familiar with earlier papers on slicing [9, 19, 24] are strongly urged to read the extended version of this paper in [25]. We also use a directed graph model of a computation 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 [19])** *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 $\mathsf{slice}(\langle E, \rightarrow \rangle, p)$. It was shown in [19] 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 $G \sqcap H$ too.

Given a computation as in Figure 5(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 Figure 5(b). The consistent cuts that belong to the slice are denoted by white filled circles in Figure 5(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 $\{e_2, f_1\}, \{e_2, f_2\}, \{e_2, f_3\}$. 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 Figure 5 (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.

From the above two observations, if the slice for $\neg p$ 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 $definitely: p$.

**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 [19] and then checking the strongly connected components of the slice in $O(n|E|)$ time [19].

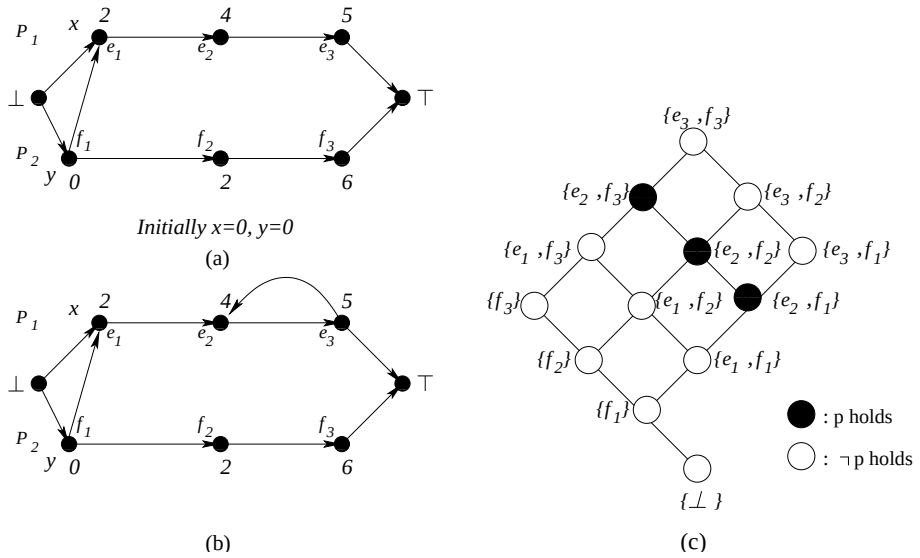


**Figure 5. (a) A computation (b) slice wrt $x \neq 4$ (c) lattice of consistent cuts of the computation**

The converse of Theorem 6 is false. Figure 6(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), we obtain the set of consistent cuts that belongs to the computation, that is, $slice(\langle E, \rightarrow \rangle, \neg p)$ has the same set of cuts as $\langle E, \rightarrow \rangle$. Therefore, the slice does not contain a non-trivial strongly connected component not in $\langle E, \rightarrow \rangle$.

Another advantage of slicing is that, We can use the slice with respect to $\neg p$ instead of the computation 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 POTA and applied it to a leader election protocol. The leader election protocol 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 [13]. For this purpose, we used the translator from execution traces to Promela (input language of SPIN). We restricted the memory usage to 256MB. We manually instrumented the program. 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. Due to lack of space further experimental results are not reported in the current version of the paper but these results and their detailed explanations are available at POTA website [22].

## 8. Conclusion

We presented space and time efficient algorithms for detecting $definitely: p$. Earlier, we developed polynomial time detection algorithms for predicates from a subset of the temporal logic CTL [3] in POTA 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.

[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.

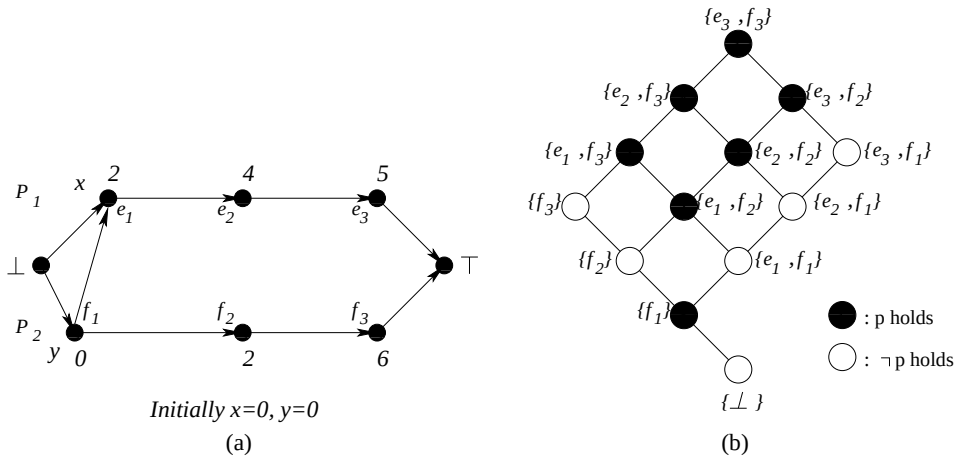


**Figure 6. (a) A computation (b) corresponding lattice**

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

| | 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 | 34.50 | 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 | 2.80 |
| 13 | * | | | 126,977 | 1.38 | 7.10 |
| 14 | * | | | 505,857 | 6.47 | 28.30 |
| 15 | * | | | 1,458,180 | 7.11 | 29.01 |
| 16 | * | | | 1,835,010 | 25.90 | 96.90 |
| 17 | * | | | 32,111,260 | 54.55 | 181.38 |

*: denotes out of memory


[6] D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Verification*, volume 1885 of *LNCS*, pages 323–330, 2000.

[7] 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.

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

[9] 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, Apr. 2001.

[10] 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, Mar. 1994.

[11] H. Hallal, A. Petrenko, A. Ulrich, and S. Boroday. Using SDL Tools to Test Properties of Distributed Systems. In *Proc. FATES'01, Workshop on Formal Approaches to Testing of Software*, Aug. 2001.

[12] K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. In *Runtime Verification 2001*, volume 55 of *ENTCS*, 2001.

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

[14] 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, Oct. 1996.

[15] 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.

[16] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: a Run-time Assurance Tool for Java Programs. In *Runtime Verification 2001*, volume 55 of *ENTCS*, 2001.

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

[18] 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.

[19] 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, Oct. 2001.

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

[21] R. Schwartz and F. Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, 7(3):149–174, 1994.

[22] A. Sen and V. K. Garg. Partial Order Trace Analzyer (POTA). http://maple.ece.utexas.edu/~sen/POTA.html.

[23] 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.

[24] 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, 2003.

[25] A. Sen and V. K. Garg. On Checking Whether a Predicate Definitely Holds. Technical Report TR-PDS-2003-003, PDSL, ECE Dept. Univ. of Texas at Austin, 2003.

[26] A. Sen and V. K. Garg. Partial Order Trace Analyzer (POTA) for Distributed Programs. In *Runtime Verification 2003*, volume 89 of *ENTCS*, 2003.

[27] 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.

[28] 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, Sept. 1999.

[29] 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, Mar. 1997.