Detecting Temporal Logic Predicates on the Happened-Before Model [*]

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

Abstract

Detection of a global predicate is a fundamental problem in distributed computing. In this paper we describe new predicate detection algorithms for certain temporal logic predicates. We use a temporal logic, CTL, for specifying properties of a distributed computation and interpret it on a finite lattice of global states. We present solutions to the predicate detection of linear and observer-independent predicates under $\mathbf{EG}$ *and* $\mathbf{AG}$ *operators of CTL. For linear predicates we develop polynomial-time predicate detection algorithms which exploit the structure of finite distributive lattices. For observer-independent predicates we prove that predicate detection is NP-complete under* $\mathbf{EG}$ *operator and co-NP-complete under* $\mathbf{AG}$ *operator. We also present polynomial-time algorithms for a CTL operator called until , for which such algorithms did not exist. Finally, our work unifies many earlier results in predicate detection in a single framework.*

1. Introduction

Correct distributed programs are difficult to write. These programs often contain bugs which are hard to detect without some kind of automatic verification. Debugging is a process of finding such bugs. A programmer upon observing a certain distributed computation for bugs can check whether the observed computation satisfies some expected property. For example, when debugging a distributed mutual exclusion algorithm, it is useful to monitor the system to detect concurrent accesses to the shared resources. A system that performs leader election may be monitored to ensure that processes agree on the current leader. It is also important to be able to observe distributed systems for fault-tolerance. On detecting a violation of a safety property like a deadlock, one of the processes must be aborted and restarted.

The problem of deciding whether a happened-before model [15] of a distributed computation satisfies a global predicate (property), referred to as the *predicate detection* problem, is the main focus of this paper.

In distributed programs no process can determine the sequence of global states the system passed. This makes it difficult to check whether a global predicate held. Another obstacle in detection of global predicates is the *state explosion problem*—the set of possible global states of a distributed program with n individual processes can be of size exponential in n. A variety of strategies for ameliorating the state explosion problem, including symbolic model checking and partial-order model checking have been explored [17, 12].

Our approach to predicate detection is based on exploiting the structure of the predicate [10]. This approach, instead of building the lattice of global states (global state space) for the distributed computation, directly uses the computation to detect if the predicate is satisfied in a global state. Some examples of the predicates for which the predicate detection can be solved efficiently are: *conjunctive* [10, 13], *disjunctive* [10], *stable* [2], *observer-independent* [3, 4], *linear* [4], and *regular* [9, 18] predicates.

Our work is different from *model checking* [8, 14], which checks that a predicate is satisfied for all computations of a program. We check that a predicate is satisfied for a *single* computation of a program since our purpose is to develop algorithms for fault-tolerance and debugging of distributed programs where a single execution trace of the program is observed. Even if model checking algorithms are used on a single computation with a finite lattice as in our case, the complexity of detecting a predicate would be in general proportional to the size of the lattice which is still exponential in the number of processes (state-explosion problem), whereas our algorithms have polynomial complexity. The exponential complexity in the case of model checking is

[*]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

due to the fact that model checking algorithms use reachability analysis (which performs fixpoint iterations that generates a sequence of global states or formulas corresponding to global states), whereas we work on the computation itself without explicitly generating all global states. Finally, we work on specific predicate classes as mentioned above, whereas model checking in general deals with arbitrary predicates.

The temporal operators under which predicate detection algorithms for a distributed computation have been discussed in the literature are: *possibly* $(\mathbf{EF})$ [6, 10, 3, 9], *definitely* $(\mathbf{AF})$ [6, 11, 3, 9], *controllable* $(\mathbf{EG})$ [20, 9], and *invariant* $(\mathbf{AG})$ [6, 10, 3, 9].

We integrate the above mentioned operators under a well-known temporal logic CTL [8] as it is done in model checking. Our distributed computation model leads to a global state space which is a finite distributive lattice. We interpret CTL on this lattice structure and use the properties of the structure itself which ultimately helps in obtaining efficient predicate detection algorithms.

We present solutions to the predicate detection of linear and observer-independent predicates under *controllable* and *invariant* operators. A linear predicate [4] is such that the set of global states that satisfy a linear predicate forms an inf-semilattice of the lattice of global states. Linear predicates include several useful predicate classes like conjunctive predicates, regular predicates, monotonic channel predicates and some relational predicates. Charron-Bost et al. [3] introduced observer-independent predicates to capture the class of predicates for which the detection of *possibly* and *definitely* are equivalent. Observer-independent predicates include predicate classes like stable predicates and disjunctive predicates.

A temporal operator that we use in CTL is *until* $(\mathbf{U})$. This operator helps in detecting properties where a condition has to hold until another condition eventually holds. We present two polynomial algorithms for detecting conjunctive, disjunctive and linear predicates under CTL temporal operator *until*. Efficient algorithms for this operator did not exist before. A formula $\mathbf{E}[p \ \mathbf{U} \ q]$ (resp. $\mathbf{A}[p \ \mathbf{U} \ q]$) intuitively means that for some sequence (resp. all sequences) of global states starting from the initial state of a computation and ending at the final state, there exists an initial prefix of the sequence such that q holds at the last state of the prefix and p holds at all other global states along the prefix. A mutual exclusion predicate like "processes are in trying state before getting to critical state" can be specified as $\mathbf{A}[try_i \ \mathbf{U} \ critical_i]$.

Table 1 lists the predicate detection algorithms for the classes of predicates mentioned above.

The remainder of this paper is organized as follows: Section 2 discusses the model we use to represent distributed programs. CTL syntax and semantics is given in Section 3. A brief overview of predicate classes is given in Section 4. We present algorithms for detecting linear predicates under *controllable* and *invariant* operators in Section 5. Section 6 has NP-completeness results for observer-independent predicates. The detection of global predicates under *until* operator and our algorithms are described in Section 7. Finally, some concluding remarks are given in Section 8.

Table 1. Predicate Detection Algorithms

Predicate	Detection Algorithm			
p	$\mathbf{EF}(p)$	$\mathbf{AF}(p)$	$\mathbf{EG}(p)$	$\mathbf{AG}(p)$
conjunctive	[10]	[11]	[10]	[11]
disjunctive	[11]	[10]	[11]	[10]
stable	[2]	[3]	trivial	trivial
linear	[4]	open	this paper	this paper
observer-independent	[4, 3]	[4, 3]	this paper	this paper
regular	[9, 18]	open	[9, 18]	[9, 18]
arbitrary	[4]	[20]	[20]	[4]

2. 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, $\rightarrow$ [15]. 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 $(E, \rightarrow)$.

We define a *consistent cut* of a computation $(E, \rightarrow)$ 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 $(E, \rightarrow)$ by $C(E)$. It is well known that the set of consistent cuts of any distributed computation $(E, \rightarrow)$ forms a *distributive lattice*, under the relation $\subseteq$ [16, 9]. We denote this lattice by $L = (C(E), \subseteq)$. 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 denote the set of maximal (with respect to happened-before relation) elements of a consistent cut G by $frontier(G)$. 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 $e \notin G$. A *maximal consistent cut sequence* $G_0, G_1, \ldots, G_l$ of $(C(E), \subseteq)$ *that starts from* G satisfies that $G_0 = G$, $G_l = E$ and for each $0 \leq i < l$, $G_i \triangleright G_{i+1}$. Note that, $\emptyset$ denotes the *initial cut* of a computation and E denotes the *final cut* of a computation.

3. CTL and Predicate Detection

In this section we first give the syntax and the semantics for the subset of the temporal logic CTL that we use. We compare the logic we use with the logics used in related work. Finally we define predicate detection problem in CTL context.

Propositional temporal logics use a finite set of atomic propositions AP, each one of which represents some property of the global state. A labeling function $\lambda : C(E) \rightarrow 2^{AP}$ assigns to each global state the set of predicates from AP that hold in it.

The formal syntax of the subset of CTL that we use in this paper is given below.
• Every atomic proposition $ap \in AP$ is a CTL formula.
• If p and q are CTL formulas, then so are $\neg p, p \wedge q, \mathbf{A}[p\ \mathbf{U}\ q]$, and $\mathbf{E}[p\ \mathbf{U}\ q]$.

The symbols $\wedge$ and $\neg$ have their usual meanings. There are two path quantifiers: $\mathbf{A}$ denotes *for all maximal consistent cut sequences* and $\mathbf{E}$ denotes *for some maximal consistent cut sequence*. $\mathbf{U}$ is the *until* temporal operator.

Given a lattice $L = (C(E), \subseteq)$, the formulas of CTL are interpreted over the consistent cuts in $C(E)$. Let p and q be CTL formulas and G be a consistent cut in $C(E)$. Then, the satisfaction relation, $L, G \models p$ means that predicate p holds at consistent cut G in lattice $L = (C(E), \subseteq)$ and is defined inductively below. We denote $G \models p$ as a short form for $L, G \models p$, when L is clear from the context.
• $G \models ap$ iff $ap \in \lambda(G)$ for an atomic proposition ap.
• $G \models \neg p$ iff $G \not\models p$.
• $G \models p \wedge q$ iff $G \models p$ and $G \models q$.
• $G \models p \vee q$ iff either $G \models p$ or $G \models q$.
• $G \models \mathbf{A}[p\ \mathbf{U}\ q]$ iff for all maximal consistent cut sequences that start from G there exists a finite prefix $G_0, \ldots, G_k$ of consistent cuts such that (i) $G_0 = G$, (ii) $G_i \triangleright G_{i+1}$ for $0 \leq i < k$, (iii) $G_k \models q$, and (iv) $G_i \models p$ for all $0 \leq i < k$.
• $G \models \mathbf{E}[p\ \mathbf{U}\ q]$ iff for some maximal consistent cut sequence that starts from G there exists a finite prefix $G_0, \ldots, G_k$ of consistent cuts such that (i) $G_0 = G$, (ii) $G_i \triangleright G_{i+1}$ for $0 \leq i < k$, (iii) $G_k \models q$, and (iv) $G_i \models p$ for all $0 \leq i < k$.

We define $L \models p$ if and only if $L, \emptyset \models p$. [1] We use the following abbreviations in writing CTL formulas:
• $\mathbf{AF}(p) \equiv \mathbf{A}[true\ \mathbf{U}\ p]$ intuitively means that p is true in some consistent cut along every finite sequence starting from $\emptyset$ and ending at E; that is, *definitely: p*.
• $\mathbf{EF}(p) \equiv \mathbf{E}[true\ \mathbf{U}\ p]$ intuitively means that p is true in some consistent cut along some finite sequence starting from $\emptyset$ and ending at E; that is, *possibly: p*.
• $\mathbf{EG}(p) \equiv \neg\mathbf{AF}(\neg p)$ intuitively means that p is true on every consistent cut along some finite sequence starting from $\emptyset$ and ending at E; that is, *controllable: p*.
• $\mathbf{AG}(p) \equiv \neg\mathbf{EF}(\neg p)$ intuitively means that p is true on every consistent cut along every finite sequence starting from $\emptyset$ and ending at E; that is, *invariant: p*.

There have been other attempts to unify the predicate detection problem, like RCL [21] and ENF [5]. In these frameworks one can specify sequences of weak conjunctive predicates which are easily expressible in CTL; since CTL is more expressive due to temporal operators. Babaoglu et al. [1] have an automata oriented framework for detecting behavioral patterns with exponential-time algorithms, whereas we have a temporal logic based framework with polynomial-time algorithms.

The *predicate detection* problem is to decide whether a happened before model of a distributed computation satisfies a predicate.

4. Predicate Classes

Our approach to predicate detection is based on exploiting the structure of the predicate. Efficient predicate detection algorithms have been designed for CTL operators $\mathbf{EF}(p)$, $\mathbf{AF}(p)$, $\mathbf{EG}(p)$, and $\mathbf{AG}(p)$, when p belongs to a specific predicate class. We first give definitions of such classes and next present their relationship to each other.

We define a predicate to be a *local predicate* if its truth value depends only on the state of a single process. For example, " the value of x on process i is 2" is a local predicate. A predicate p is said to be *conjunctive* (resp. *disjunctive*) if it can be written as a conjunction (resp. disjunction) of local predicates. A *stable* [2] predicate p is such that the predicate remains true once it becomes true. A predicate p is said to

[1]Note that a distributed program is modeled by a set of partial order sets (computations). In that case, a distributed program P satisfies a CTL formula p if and only if $L \models p$ for each L in P.

be *observer-independent* [3, 4] if $\mathbf{AF}\ (p) \iff \mathbf{EF}\ (p)$ that is, if the predicate holds in some observation then it holds in all observations. Note that if predicate p holds initially then it is an observer-independent predicate. We say that a predicate p is *regular* [9, 18] 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 will also satisfy the predicate. A *linear* [4] predicate is such that the set of consistent cuts that satisfy the predicate forms an inf-semilattice of the lattice of consistent cuts.

We use I_p to denote the least (initial) consistent cut that satisfies predicate p, if the least consistent cut exists. Note that, I_p exists if p is a conjunctive, regular, or linear predicate.

In this paper, we use a fragment of CTL in which formulas are in one of the following forms: $\mathbf{AF}(p)$, $\mathbf{EF}(p)$, $\mathbf{EG}(p)$, $\mathbf{AG}(p)$, $\mathbf{E}[p\ \mathbf{U}\ q]$, and $\mathbf{A}[p\ \mathbf{U}\ q]$, where p and q are non-temporal predicates from the classes of predicates; conjunctive, disjunctive, observer-independent, stable, linear, and regular. We say that a predicate is *non-temporal* if it does not contain temporal operators such as $\mathbf{AF}$, $\mathbf{EF}$, $\mathbf{EG}$, $\mathbf{AG}$, $\mathbf{EU}$, and $\mathbf{AU}$. In other words, we do not consider nested temporal predicates in this paper.

Note that linear predicates include regular predicates and regular predicates include conjunctive predicates. Similarly, observer-independent predicates include stable and disjunctive predicates. We present formal proofs of these relations in [19].

5. Detecting Linear Predicates

Linear predicates include several useful predicate classes like conjunctive predicates, regular predicates, monotonic channel predicates and some relational predicates. Garg et al. [4] introduced efficient algorithms for detecting linear predicates under $\mathbf{EF}$ operator. In this section we introduce efficient algorithms for detecting linear predicates under $\mathbf{EG}$ and $\mathbf{AG}$ operators. Algorithms for detecting subclasses of linear predicates under such CTL operators are shown in Table 1.

Our algorithm for detecting linear predicates under $\mathbf{EG}$ operator finds a sequence of consistent cuts starting from the final cut to the initial cut such that every cut in this sequence satisfies the predicate, if such a sequence exists. Algorithm A1 of Fig. 1 displays our algorithm. Starting from the final cut we move towards the initial cut moving one cut at a time. The next consistent cut to explore is determined based on choosing one of the consistent cuts that precedes the current consistent cut in the computation and that satisfies the predicate. We prove that it does not matter which one of these consistent cuts is chosen as long as one exists. If $\mathbf{EG}(p)$ is satisfied in the lattice then the algorithm returns true, and vice-versa.

```
Algorithm A1

Input:    $((E,\rightarrow),p)$ $p$ is a linear predicate
Output:   $(E,\rightarrow)$ satisfies $\mathbf{EG}(p)$ or not
Step 1.   Let $W$ be the final cut of $(E,\rightarrow)$
          If $W$ does not satisfy $p$ then return false
Step 2.   while $W \neq$ initial cut do
Step 3.       Let $Q = \{G \in C(E) \mid G \models p \wedge G \triangleright W\}$
          be the set of predecessors of $W$ that satisfy $p$
Step 4.       If $Q$ is empty then return false
Step 5.       else let $W$ be an arbitrary element from $Q$
Step 6.   endwhile
Step 7.   If the initial cut satisfies $p$ then return true,
          else return false

Algorithm A2
Input:    $((E,\rightarrow),p)$ $p$ is a linear predicate
Output:   $(E,\rightarrow)$ satisfies $\mathbf{AG}(p)$ or not
Step 1.   Let $V = \mathcal{M}(L) \cup \{E\}$
Step 2.   If any consistent cut in $V$ does not satisfy $p$
          then return false, else return true
```

Figure 1. Algorithms to detect $\mathbf{EG}(p)$ and $\mathbf{AG}(p)$ when p is linear

Next we prove the correctness of Algorithm A1 using the following lemma which is a simple observation of the lattice structure of distributed computations.

Lemma 1 ([19]) *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 $(G \sqcap J) \triangleright J$.*

Theorem 2 *Algorithm A1 detects $\mathbf{EG}(p)$ for p linear.*
Proof: If the algorithm returns true then it is clear that $\mathbf{EG}(p)$ holds. Now we prove that if $\mathbf{EG}(p)$ holds then algorithm returns true. Since $\mathbf{EG}(p)$ holds, there exists a sequence from the final cut to the initial cut along which p holds. We prove by induction on the length of this sequence, j that if $\mathbf{EG}(p)$ holds then the algorithm returns true.
• Base case ($j = 1$) There is only a single sequence $\emptyset, E$ therefore trivial.
• Induction step ($j = k + 1$) Assuming that the assertion is true up to $j = k$ we prove that it holds for $j = k + 1$. If $\mathbf{EG}(p)$ is true then there exists a sequence $G_0 = \emptyset, \ldots, G_k, G_{k+1} = E$ that satisfies the predicate. If in Step 5 of the algorithm G_k is chosen then we are done else let the consistent cut chosen by the algorithm in this step be $H \neq G_k$. We know that $H \models p$ from Step 3. $H \sqcap G_k$ exists since L is a finite distributive lattice. From

Lemma 1, $(H \sqcap G_k) = G_k$ (in which case we are done) or $(H \sqcap G_k) \triangleright G_k$ hence the length of the sequence from the initial cut to $H \sqcap G_k$ is $< k$ therefore the induction hypothesis applies. ■

Complexity of the algorithm: The complexity of finding the next consistent cut is $O(n)$ since there can be at most n preceding cuts of a consistent cut. A consistent cut sequence in the lattice can be of at most $|E|$ length since there are $|E|$ events in the computation therefore the complexity of the while loop is $O(|E|)$. The total complexity of the algorithm is $O(n|E|)$, where n is the number of processes and E is the set of events. This result applies to regular predicates as well and improves the $O(n^2|E|)$ complexity in [9].

For detecting linear predicates under $\mathbf{AG}$ operator we make use of Birkhoff's representation theorem for finite distributive lattices. This theorem uses the notion of meet-irreducible elements of a lattice.

Definition 1 (meet-irreducible element [7]) *An element $x \in L$ is meet-irreducible if $x \neq E$ and $\forall\ a, b \in L : x = a \sqcap b \Rightarrow (x = a) \vee (x = b)$. Dually, an element $y \in L$ is join-irreducible if $y \neq \emptyset$ and $\forall\ a, b \in L : y = a \sqcup b \Rightarrow (y = a) \vee (y = b)$.*

In other words, x cannot be further decomposed into the meet of other elements in the lattice, just as prime numbers cannot be further factored into the product of other natural numbers. For example, the meet-irreducible elements of the lattice in Fig. 2(b), are shown in filled circles. Pictorially, in a finite distributive lattice an element is meet-irreducible iff it has exactly one upper cover, that is, it has exactly one outgoing edge. We denote the set of all meet-irreducible elements of a lattice L by $\mathcal{M}(L)$. We use meet-irreducibility to provide us with a kind of prime factorization for finite distributive lattices. We call a subset S of a poset P an *up-set* if $s \in S \wedge s < t \Rightarrow t \in S$ and denote by $\mathcal{U}$ the set of up-sets of a poset.

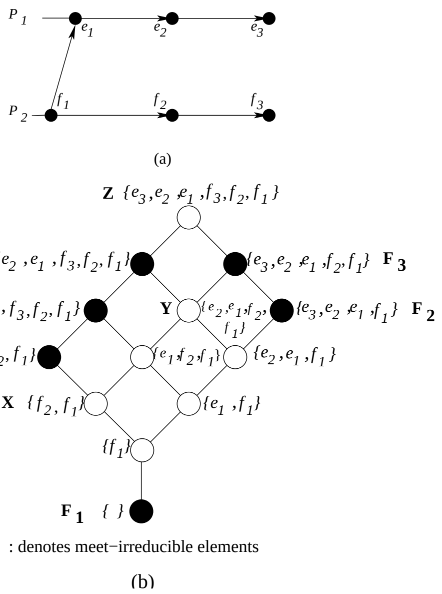
Theorem 3 (Birkhoff [7]) *Let L be a finite distributive lattice. Then the map $f : L \rightarrow \mathcal{U}(\mathcal{M}(L))$ defined by $f(a) = \{x \in \mathcal{M}(L) \mid a \leq x\}$ is an isomorphism of L onto $\mathcal{U}(\mathcal{M}(L))$. Dually, let P be a finite poset. Then the map $g : P \rightarrow \mathcal{M}(\mathcal{U}(P))$ defined by $g(a) = \{x \in P \mid a \leq x\}$ is an isomorphism of P onto $\mathcal{M}(\mathcal{U}(P))$.*

Birkhoff's theorem says that there is a one-to-one correspondence between a finite poset and a finite distributive lattice. Given a finite distributive lattice, we can recover the poset by focusing on its meet-irreducible elements. Given a finite poset, we get the finite distributive lattice by considering its set of up-sets. Informally, every lattice element of a finite distributive lattice (except for the final cut) can be defined as meet of a subset of meet-irreducible elements. The following is a corollary of Theorem 3.

Corollary 4 ([7]) *Let a be any element in a distributive lattice L. Then $a = \bigsqcap \{x \in \mathcal{M}(L) \mid a \subseteq x\}$.*

For example, in the lattice shown in Fig. 2, $X = \bigsqcap\ \{E_1, E_2, E_3, F_3\}, Y = \bigsqcap\ \{E_3, F_3\}$, and similarly for the other elements.

Birkhoff's theorem and the corollary is useful in computational sense as well because the set of meet-irreducible elements of a lattice is generally exponentially smaller than the size of the lattice itself.



P_1
e_1
e_2
e_3
P_2
f_1
f_2
f_3
(a)
Z $\{e_3, e_2, e_1, f_3, f_2, f_1\}$
E₃ $\{e_2, e_1, f_3, f_2, f_1\}$
$\{e_3, e_2, e_1, f_2, f_1\}$ **F₃**
E₂ $\{e_1, f_3, f_2, f_1\}$
Y $\{e_2, e_1, f_2, f_1\}$
$\{e_3, e_2, e_1, f_1\}$ **F₂**
E₁ $\{f_3, f_2, f_1\}$
$\{e_1, f_2, f_1\}$
$\{e_2, e_1, f_1\}$
X $\{f_2, f_1\}$
$\{e_1, f_1\}$
$\{f_1\}$
F₁ $\{\ \}$
● : denotes meet−irreducible elements
(b)

Figure 2. A computation (a) and its lattice (b)

We present an algorithm to detect linear predicates under $\mathbf{AG}$ operator in Fig. 1. Intuitively, algorithm A2 says that, to detect a linear predicate under $\mathbf{AG}$ operator it is sufficient and necessary to check whether the predicate is satisfied at the meet-irreducible elements and the final cut of the lattice. This follows from results above, that is, since p is linear, if p holds at meet-irreducible elements then it holds at their meets. From corollary 4, every element in the lattice (except for the final cut) can be defined as the meet of a subset of meet-irreducible elements.

Complexity of the algorithm: The main complexity comes from finding the set of meet-irreducible elements. Garg et al. [9] has an $O(n^2|E|)$ algorithm to compute the list of join-irreducible elements. The same algorithm can be used for meet-irreducible elements as well, only this time we apply their algorithm backwards on the computation or with all the edges in the computation reversed.

We can use algorithms above for detecting *post-linear* predicates defined in [4]. Post-linear predicates are such that the set of consistent cuts that satisfy a post-linear predicate forms a sup-semilattice of the lattice of consistent cuts. Algorithm A1 can be modified so that it starts from the initial cut and moves towards the final cut. The next consistent cut to move is determined based on finding one of the consistent cuts that follows the current consistent cut in the computation and satisfies the predicate; again it does not matter which one of these consistent cuts is chosen as long as one exists. Algorithm A2 can be modified such that in Step 1 instead of using meet-irreducible elements and the final cut, the new algorithm uses the join-irreducible elements and the initial cut.

6. Detecting observer-independent predicates

It was shown in [4] and [20] that detecting an arbitrary predicate is NP-complete under $\mathbf{EF}$ and $\mathbf{EG}$ operators. In this section we prove that detecting an observer-independent predicate is NP-complete under $\mathbf{EG}$ operator and co-NP-complete under $\mathbf{AG}$ operator.

Theorem 5 *Given a distributed computation, detecting an observer-independent predicate p under $\mathbf{EG}$ operator is NP-complete.*
Proof: The proof of the theorem is very similar to the proof for NP-completeness of an arbitrary predicate under $\mathbf{EG}$ operator [20]. The problem is in NP because it takes polynomial time to check that a candidate global sequence of consistent cuts is valid and that it satisfies the predicate p. To show that it is NP-hard, we reduce Satisfiability to this problem. If p is the boolean expression in Satisfiability, then for each variable $x_1, \ldots, x_m$ in p, we assign a separate process with two states, true and false (Fig. 3 (a)). We define a process for an extra boolean variable x_{m+1} which starts true, goes through a false state, and ends true again. We define $P = p \vee x_{m+1}$. It is clear that P is observer-independent since it is satisfied at the initial state. Then we apply $\mathbf{EG}$ algorithm to detect predicate P. If there exists a global sequence that satisfies $\mathbf{EG}(P)$, then the global state with $x_{m+1} = false$ will have a satisfying assignment for the variables of p. Conversely, if p is satisfiable, then there exists a satisfying global sequence. ■

Theorem 6 *Given a distributed computation, detecting an observer-independent predicate p under $\mathbf{AG}$ operator is co-NP-complete.*
Proof: The problem is in co-NP because it takes polynomial time to check that a candidate consistent cut satisfies the negation of the predicate p. To show that it is co-NP-hard, we reduce Tautology, a co-NP-complete problem, to this problem. If p is the boolean expression in Tautology, then for each variable $x_1, \ldots, x_m$ in p, we assign a separate process with true and false states (Fig. 3 (b)). We define a process for an extra boolean variable x_{m+1} which starts true, and ends at a false state. We define $P = p \vee x_{m+1}$. Similar to the proof of above theorem, it is clear that P is observer-independent since it is satisfied at the initial state. We then apply $\mathbf{AG}$ algorithm to detect invariance of P. If the algorithm returns true, then all global sequences satisfy the predicate and all global states with $x_{m+1} = false$ will have satisfying assignments for the variables of p. Conversely, if p is a tautology, then all global states in all sequences will satisfy p. ■

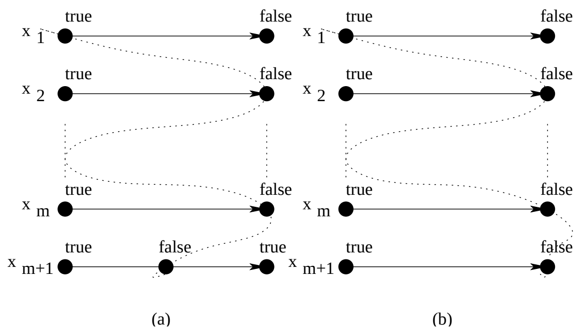


Figure 3. Detecting an observer-independent predicate p under (a) EG is NP-complete (b) AG is co-NP-complete

Although predicate detection of observer-independent predicates under $\mathbf{EG}$ and $\mathbf{AG}$ operators is intractable, subclasses of this predicate class like disjunctive and stable predicates have polynomial detection algorithms (cf. Table 1).

7. Detecting predicates with Until operator

$\mathbf{E}[p\ \mathbf{U}\ q]$ ($\mathbf{EU}$) and $\mathbf{A}[p\ \mathbf{U}\ q]$ ($\mathbf{AU}$) predicates aid in detecting conditions where a condition has to hold until another condition eventually holds. See Fig. 4 for an example computation for detecting $\mathbf{E}[p\ \mathbf{U}\ q]$ where the predicate in question is "there exists an observation such that variable z of process P_3 is less than 6 and variable x of process P_1 is less than 4 *until* channels are empty and variable x of process P_1 is greater than 1". Each event in the computation is labeled with the value of the respective variable immediately after the event is executed. The paths in the lattice starting from the initial cut with patterned circles leading to cuts with filled circles satisfy the predicate. It can also be observed that the first part of the predicate, p, is a conjunctive predicate and the second part, q, is a linear predicate.

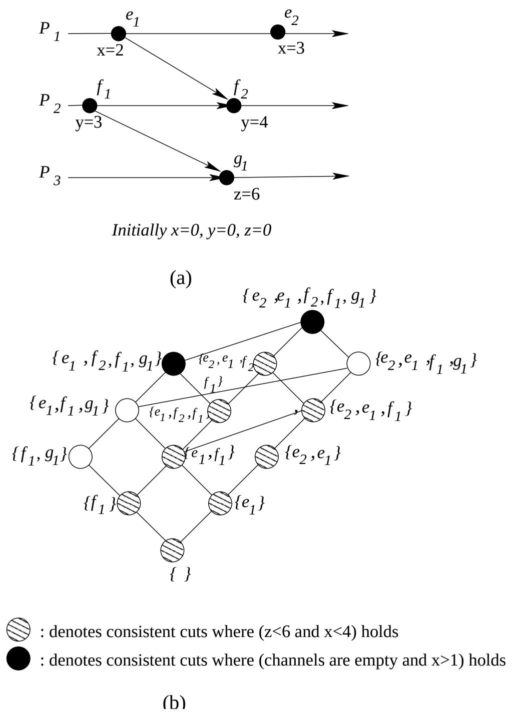


Figure 4. A computation (a) and its lattice (b)

In detecting $\mathbf{E}[p \ \mathbf{U} \ q]$, a lattice construction based approach would, in the worst case, have to check all the consistent cut sequences where p holds until eventually q holds. It is clear that this is very inefficient due to both state explosion and exponential number of consistent cut sequences in the number of events. Theorem 7 shows a way to find such a consistent cut sequence in a more efficient manner.

Theorem 7 ([19]) *Given a finite distributive lattice $L = (C(E), \subseteq)$ for a computation $(E, \rightarrow)$, a conjunctive predicate p, and a linear predicate q; $L \models \mathbf{E}[p \ \mathbf{U} \ q]$ if and only if there exists a finite sequence $K_0 \ldots K_j$ of consistent cuts such that*

(i) $K_0 = \emptyset$, (ii) $K_i \rhd K_{i+1}$ for $0 \leq i < j$,
(iii) $K_j = I_q$ where I_q is the least consistent cut that satisfies predicate q, and (iv) $K_i \models p$ for all $0 \leq i < j$.

Intuitively, Theorem 7 says that when p is conjunctive and q is linear, in detecting $\mathbf{E}[p \ \mathbf{U} \ q]$, it is necessary and sufficient to check for the existence of a sequence of consistent cuts starting from the initial consistent cut to the consistent cut I_q where p holds along the sequence until q holds in I_q. This greatly simplifies the task of detecting $\mathbf{E}[p \ \mathbf{U} \ q]$ because otherwise we would have to check for the existence of a sequence of consistent cuts starting from the initial consistent cut to "some" consistent cut G where p holds along the sequence until q holds in G.

If we apply the theorem to the computation in Fig. 4 it is observed that there is a sequence of consistent cuts which satisfies the theorem, that is $\emptyset$, $\{f_1\}$, $\{e_1, f_1\}$, $\{e_1, f_2, f_1\}$, $\{e_1, f_2, f_1, g_1\}$ where $I_q = \{e_1, f_2, f_1, g_1\}$. Out of a possible 7 paths which start from the initial cut and satisfy the predicate it is enough to consider only the ones that lead to I_q, of which there are only 2 in this case.

We use Theorem 7 to obtain an algorithm to detect $\mathbf{E}[p \ \mathbf{U} \ q]$ as in Fig. 5. [2]

```
Algorithm A3

Input:    $((E, \rightarrow), p, q)$ $p$ is conjunctive, $q$ is linear
Output:   $(E, \rightarrow)$ satisfies $\mathbf{E}[p \ \mathbf{U} \ q]$ or not
Step 1.   Find $I_q$, the least consistent cut that satisfies $q$.
Step 2.   Check if $\mathbf{EG}(p)$ is satisfied in any $(E', \rightarrow)$
          such that $E' = I_q - \{e\}$, $e \in frontier(I_q)$.
          If the result is positive then return true,
          else return false.
```

Figure 5. Detection algorithm for $\mathbf{E}[p \ \mathbf{U} \ q]$

Complexity of the algorithm: When p is a conjunctive predicate, the algorithm presented in Chase et al. [4] for linear predicates is applicable in Step 1. The complexity of the Chase et al. algorithm depends upon the complexity of determining an event to advance a given consistent cut G. In this paper, we assume that the complexity of determining the required event is $O(n)$ and therefore the complexity of Step 1 is $O(n|E|)$. In Step 2, the optimal algorithm presented in [18] for generating a slice of a conjunctive predicate is applicable. The complexity of this is $O(|E|)$ and there are at most n computations generated in Step 2. Therefore the complexity of Step 2 is $O(n|E|)$. The overall complexity of the algorithm is $O(n|E|)$.

We use the equality $\mathbf{A}[p \ \mathbf{U} \ q] \iff \neg(\mathbf{EG}(\neg q) \vee \mathbf{E}[\neg q \ \mathbf{U} \ (\neg p \wedge \neg q)])$ to detect predicates under $\mathbf{AU}$ operator. This equality gives a representation of $\mathbf{AU}$ operator in terms of $\mathbf{EU}$ and $\mathbf{EG}$ operators. When predicates p and q are disjunctive predicates we can use the algorithm developed for $\mathbf{EU}$ since $(\neg q)$ is a conjunctive predicate and $(\neg p \wedge \neg q)$ is a linear predicate. Similarly we can use the algorithm presented in [18] for detecting conjunctive predicates under $\mathbf{EG}$ operator to detect $\mathbf{EG}(\neg q)$. The complexity of this algorithm is $O(|E|)$. The overall complexity of $\mathbf{A}[p \ \mathbf{U} \ q]$ algorithm is therefore $O(n|E|) + O(|E|)$ which is $O(n|E|)$.

[2] Note also that predicate q in Theorem 7 actually could be weaker than a linear predicate where it only has to satisfy the property that
$\exists J : \forall G : (G \models p) \ \Rightarrow \ ((J \subseteq G) \wedge (J \models p))$
which intuitively means that there exists a least consistent cut that satisfies predicate p.

8. Conclusions and Future work

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. Our focus in this paper has been on developing and unifying predicate detection algorithms using a well-known temporal logic CTL on the happened-before model. CTL unifies known predicate detection operators by enabling their expressibility with the operators of the logic. We solved predicate detection of linear and observer-independent predicates under *controllable* and *invariant* operators. We presented polynomial-time algorithms for detecting linear predicates under these operators. These results also improve previous ones for regular predicates. We also proved that detecting observer-independent predicates under *controllable* operator is NP-complete and under *invariant* operator is co-NP-complete. We have used a temporal operator in CTL, called *until* which characterizes a class of predicates for which efficient predicate detection algorithms did not exist. We have determined the necessary and sufficient conditions for solving detection of *until* predicates for conjunctive, disjunctive, and linear predicates. We are planning on developing a debugging environment for the happened-before model making use of the algorithms presented here. Also an open problem is to find polynomial-time algorithms for detecting regular and linear predicates under $\mathbf{AF}$ operator. Another area of future work will be to develop efficient on-line versions of our algorithms.

References

[1] O. Babaoğlu and M. Raynal. Specifi cation and verifi cation of dynamic properties in distributed computations. *Journal of Parallel and Distributed Computing*, 28:173–185, 1995.

[2] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, Feb. 1985.

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

[4] C. Chase and V. K. Garg. Detection of global predicates: Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

[5] H. Chiou and W. Korfhage. ENF event predicate detection in distributed systems. In *Principles of Distributed Computing*, pages 91–100, Los Angeles, California, Aug. 1994.

[6] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, CA, May 1991. ACM/ONR.

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

[8] E. M. Clarke and E. A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In *Proc. of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, Yorktown Heights, New York, May 1981.

[9] V. K. Garg and N. Mittal. On slicing a distributed computation. In *Proc. of the* 15^{th} *International Conference on Distributed Computing Systems (ICDCS)*, pages 322–329, Phoenix, Arizona, 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] V. K. Garg and B. Waldecker. Detection of strong unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 7(12):1323–1333, Dec. 1996.

[12] P. Godefroid and P. Wolper. A partial approach to model checking. In *Proceedings of the 6th IEEE Symposium on Logic in Computer Science*, pages 406–415, 1991.

[13] M. Hurfi n, M. Mizuno, M. Raynal, and M. Singhal. Effi - cient distributed detection of conjunction of local predicates. Technical Report 2731, IRISA, Rennes, France, Nov. 1995.

[14] J. P. Queille and J. Sifakis. Specifi cation and verifi cation of concurrent systems in CESAR. In *Proc. of the Fifth International Symposium in Programming*, volume 137 of *LNCS*, pages 337–351, New York, 1982. Springer-Verlag.

[15] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[16] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proc. of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B.V. (North-Holland), 1989.

[17] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[18] N. Mittal and V. K. Garg. Computation slicing: Techniques and theory. In *In Proc. of the* 15^{th} *International Symposium on Distributed Computing (DISC)*, pages 78–92, Lisbon, Portugal, 2001.

[19] A. Sen and V. K. Garg. Detecting temporal logic predicates on the happened-before model. Technical Report TR-PDS-2001-003, PDSL, ECE Dept. Univ. of Texas at Austin, 2001. Available at http://www.ece.utexas.edu/˜sen/publications/TR-PDS-2001-003.ps.gz.

[20] A. Tarafdar and V. K. Garg. Predicate control for active debugging of distributed programs. In *In Proc. of the 9th Symposium on Parallel and Distributed Processing (SPDP)*, Orlando, 1998.

[21] A. I. Tomlinson and V. K. Garg. Observation of software for distributed systems with RCL. In *Proc. of 15th Conference on the Foundations of Software Technology & Theoretical Computer Science*. Springer Verlag, Dec. 1995. Lecture Notes in Computer Science.