On Detecting Global Predicates in Distributed Computations

Neeraj Mittal
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188, USA
neerajm@cs.utexas.edu

Vijay K. Garg*
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084, USA
garg@ece.utexas.edu

Abstract

Monitoring of global predicates is a fundamental problem in asynchronous distributed systems. This problem arises in various contexts such as design, testing and debugging, and fault-tolerance of distributed programs. In this paper, we establish that the problem of determining whether there exists a consistent cut of a computation that satisfies a predicate in k-CNF, $k > 1$, in which no two clauses contain variables from the same process is NP-complete in general. A polynomial-time algorithm to find the consistent cut, if it exists, that satisfies the predicate for special cases is provided. We also give algorithms albeit exponential that can be used to achieve an exponential reduction in time over existing techniques for solving the general version.

Furthermore, we present an algorithm to determine whether there exists a consistent cut of a computation for which the sum $x_1 + x_2 + \cdots + x_n$ exactly equals some constant k, where each x_i is an integer variable on process p_i such that it is incremented or decremented by at most one at each step. As a corollary, any symmetric global predicate on boolean variables such as absence of simple majority and exclusive-or of local predicates can now be detected. Additionally, the problem is proved to be NP-complete if each x_i can be changed by an arbitrary amount at each step.

Our results solve the previously open problems in predicate detection proposed in [7] and bridge the gap between the known tractability and intractability results.

1. Introduction

Correct non-trivial distributed programs are hard to write. Testing and debugging is an important and feasible way to ensure their reliability and dependability. To that end, predicate detection problem is a useful abstraction for analyzing the executions of distributed programs. For example, when debugging a distributed mutual exclusion algorithm, detecting concurrent accesses to a shared resource is useful. In a leader election protocol, it is necessary to ensure that processes agree on the current leader. Predicate detection is also a natural abstraction for monitoring distributed systems for various reasons such as fault-tolerance. For example, on detecting a deadlock, one of the processes must be aborted and restarted.

An asynchronous distributed system is characterized by lack of global clock, lack of shared memory, and unbounded relative processor speeds and messages delays. Consequently, it is impossible to determine the exact order in which the events on different processes were executed; the events can only be partially ordered [13]. This leads to the combinatorial explosion problem—the number of possible states the system passed through are, in general, exponential, thereby making the predicate detection problem non-trivial. Chase and Garg [3] proved that detecting a predicate in 3-CNF is NP-complete in general. Stoller and Schneider [15] establish the NP-completeness of detecting even a 2-local conjunctive predicate (each conjunct is a function on variables of at most two processes) in general.

Nonetheless, the problem can be solved efficiently for several useful classes of predicates such as stable [2, 1, 14], conjunctive [9, 10], linear and semi-linear [4], and relational [18] predicates. Several fast but exponential algorithms have also been developed for solving the general version of the problem [5, 11, 16]. Stoller and Schneider [15] give an algorithm for detecting a predicate satisfying certain structure by reducing the problem to multiple predicate detection problems each of which is solvable using Garg and Waldecker's algorithm for monitoring a conjunctive predicate [9].

Tarafdar and Garg [17] considered extension of the Lamport's happened-before model [13] for predicate detection that allows events on a process to be partially ordered. They proved that detecting even a conjunctive predicate becomes NP-complete, in general, in this model. However, they solved the problem efficiently for special cases when either

*supported in part by the NSF Grants ECS-9907213, CCR-9520540, TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant.

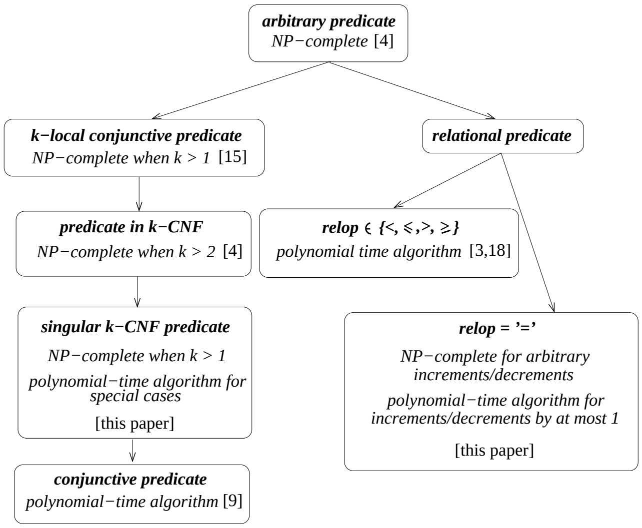

Figure 1. Known results in predicate detection.

all receive events on every process are totally ordered or all send events on every process are totally ordered.

Our contributions in this paper are as follows. We solve the previously open problems in predicate detection proposed in [7]. In Section 3, we establish that the problem of determining whether there exists a consistent cut of a computation that satisfies a predicate in k-CNF such that no two clauses contain variables from the same process, called singular k-CNF predicate, is NP-complete in general when k is at least 2. Our result bridges the gap between the known tractability [9] and intractability [3, 15] results in detecting conjunction of clauses (see Figure 1) and subsumes the two earlier known NP-completeness results. Furthermore, the result can be used to establish the intractability of other related interesting problems. A polynomial-time algorithm to find the consistent cut, if it exists, that satisfies a singular k-CNF predicate for special cases is provided. We also give algorithms albeit exponential that can be used to achieve an exponential reduction in time over existing techniques for solving the general version.

Moreover, in Section 4, we present an algorithm to determine whether there exists a consistent cut of a computation for which the sum $x_1 + x_2 + \cdots + x_n$ exactly equals some constant k, where each x_i is an integer variable on process p_i such that it is incremented or decremented by at most one at each step. As a corollary, any symmetric global predicate on boolean variables can now be observed. Additionally, the problem is proved to be NP-complete if each x_i can be changed by an arbitrary amount at each step. Our results build upon and, in some sense, complete the work described in [3, 18].

2. Model and notation

In this section, we formalize the notion of distributed computation, consistent cut and global predicate.

2.1. Distributed computations

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

A *local computation* of a process is described by a sequence of events that transforms the *initial state* of the process to a *final state*. At each step, the *local state* of a process is captured by the initial state and the sequence of events that have been executed up to that step. We assume that there is a fictitious event for each process, called the *initial event*, that initializes the state of the process. The initial event occurs before any other event on the process. Let $\bot_i$ and $\top_i$ denote the initial and final event, respectively, on process p_i.

Each event is a *send event*, a *receive event* or an *internal event*. An event can be a send event as well as a receive event. An event causes the local state of a process to be updated. Additionally, a send event causes a message or a set of messages to be sent and a receive event causes a message or a set of messages to be received. We assume that all events are distinct. We use lowercase letters e and f to represent events. Let $e.proc$ denote the process on which event e occurs. The previous and next events of e on $e.proc$ are denoted by $e.pred$ and $e.succ$, respectively, if they exist. We denote the order of events on process p_i by $<_{p_i}$ and let $<_P = \bigcup_{1 \leq i \leq n} <_{p_i}$. Further, let $\lhd_M$ be the relation induced by messages, that is, $\lhd_M = \{(s, r) \mid s$ is a send event and r is the corresponding receive event$\}$.

A *distributed computation* is modeled by an irreflexive partial order on the set of events of the underlying program's execution. We use $E_{\prec}$ to denote a distributed computation with the set of events E and the irreflexive partial order $\prec$ (read as "precedes"). We do not assume that the distributed computation is *complete*, that is, every message that was sent has been received. Let $E.\bot$ and $E.\top$ denote the set of initial and final events, respectively. We assume that $\prec$ includes $<_P$ and $\lhd_M$ and an initial event precedes any other event, that is, for each $\bot_i \in E.\bot$ and $e \in E \setminus E.\bot$, $\bot_i \prec e$, where "$\setminus$" denotes the set difference operation. The irreflexive partial order $\prec$ could be (but not restricted to) the *happened-before* relation defined by Lamport [13].

A *run* of a distributed computation $E_{\prec}$ is some total order of events in E consistent with the partial order $\prec$.

Observe that every run is a distributed computation whose events are totally ordered. We use the terms "distributed computation" and "computation" interchangeably.

2.2. Cuts and consistent cuts

Intuitively, a cut represents the global state of a distributed system. A *global state* is a collection of local states, one from each process. Equivalently, a *cut* of a computation $E_{\prec}$ is a set of events C, where $E.\bot \subseteq C$, such that, for each event e in C, $e.pred$ is also in C (if it exists).

Some cuts or global states cannot arise in the execution of the distributed system. Only those cuts that respect causality can possibly occur. A cut C is *consistent* iff, for each event e in C, all its preceding events are also in C. Formally,

$$C \text{ is a consistent cut of } E_{\prec} \;\triangleq\; (E.\bot \subseteq C) \bigwedge \langle \forall\, e, f :: (e \prec f) \wedge (f \in C) \;\Rightarrow\; e \in C \rangle$$

Observe that every consistent cut is a computation and vice versa. A cut C *passes through* an event e on process P iff e is the last event in P to be contained in C. Formally,

$$C \text{ passes through } e \;\triangleq\; (e \in C) \bigwedge (e \notin E.\top \;\Rightarrow\; e.succ \notin C)$$

Two events are *consistent* if there exists a consistent cut that passes through both the events, otherwise they are *inconsistent*. It can be verified that events e and f are inconsistent iff either $e.succ \preccurlyeq f$ or $f.succ \preccurlyeq e$. Finally, two events e and f are *independent* iff they are incomparable with respect to $\prec$. For example, in Figure 2, events f and h are consistent whereas events e and h are not. Also, events f and g are independent whereas events f and h are not.

2.3. Global predicates

A *global predicate* (or simply a predicate) is a boolean-valued function defined on a cut or global state. A global predicate is *local* iff it is a function of variables of a single process. Given a set of local predicates, one for each process, we define *true events* as those events for which the relevant variable evaluates to true. In this paper, whenever it is appropriate, we encircle the true events in our figures.

A conjunction of local predicates is called *conjunctive predicate* [9]. A predicate of boolean variables in CNF is called *singular* iff no two clauses contain variables from the same process. Intuitively, a predicate in CNF is singular if it is possible to rewrite the predicate such that each variable occurs in at most one clause and each process hosts at most one variable. For example, for the computation in Figure 2, the predicate $(x_1 \vee x_3) \wedge (x_2 \vee \neg x_4)$ is singular but the predicate $(x_1 \vee x_2) \wedge (x_2 \vee \neg x_3 \vee x_4)$ is not. For convenience,

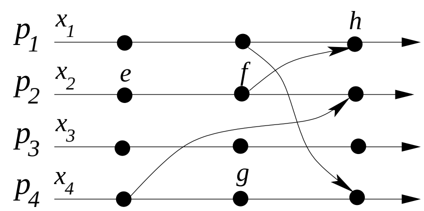

Figure 2. A distributed computation.

we write a singular predicate in k-CNF (exactly k literals per clause) as *singular k-CNF predicate*. Note that a singular k-CNF predicate reduces to a conjunctive predicate when k is 1.

A *relational predicate* [18] is of the form $x_1 + x_2 + \cdots + x_n$ relop k, where each x_i is an integer variable on process p_i and relop $\in \{=, <, >, \leq, \geq\}$. Note that our definition of relational predicates includes equality which was excluded in the definition by Tomlinson and Garg [18].

The predicate detection problem can be defined under two modalities, namely $possibly$ and $definitely$ [5], which roughly correspond to weak and strong predicates [8], respectively. The predicate $possibly: b$ is true in a computation iff there is a consistent cut that satisfies b. The predicate $definitely: b$ holds in a computation iff b eventually becomes true in all runs of the computation. Possibly true predicates are useful for detecting bad conditions such as violation of mutual exclusion and absence of simple majority, whereas definitely true predicates are useful for verifying the occurrence of good conditions such as commit point of a transaction and election of a leader. In this paper, unless otherwise stated, we focus on observing predicates under $possibly$ modality and omit the word "$possibly$" when distinction between the two modalities is not required. For convenience, we abbreviate the predicate $possibly: (x_1 + x_2 + \cdots + x_n$ relop $k)$ by $possibly:$ (relop k). For example, $possibly: (= k)$ is a shorthand for $possibly: (x_1 + x_2 + \cdots + x_n = k)$. Likewise, we obtain $definitely:$ (relop k).

3. Detecting singular k-CNF predicates

First, we prove that the problem of monitoring a singular 2-CNF predicate is NP-complete. Next, we present polynomial-time algorithm for solving special cases of the problem, namely when the computation is either receive-ordered or send-ordered. Finally, we give algorithms albeit exponential that can be used to achieve an exponential reduction in time over existing techniques for solving the general version. Our NP-completeness result solves two of the open problems proposed in [7] and subsumes the earlier known two NP-completeness results [4, 15]. Our proof and algorithms use the following observation:

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

The observation is the consequence of the fact that, given a set of pairwise consistent events—not necessarily from all processes, it is always possible to find a consistent cut that passes through all the events in the set.

3.1. NP-completeness result

The problem is in NP because the general problem of observing an arbitrary boolean expression is in NP [4]. To prove its NP-hardness, we transform an arbitrary instance of a variant of the satisfiability problem, which we call *non-monotone 3-SAT problem*, to an instance of detecting a singular 2-CNF predicate.

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

It is easy to prove that the non-monotone 3-SAT problem is NP-complete in general. This is because, given a formula in 3-CNF, it can be easily transformed into a formula that satisfies the above-mentioned conditions, which we call a *non-monotone 3-CNF formula*. Consider a clause in a formula in 3-CNF containing only positive literals, say $C = y_1 \vee y_2 \vee y_3$. We replace the clause C with clauses $y_1 \vee y_2 \vee \neg z_3$, $y_3 \vee z_3$ and $\neg y_3 \vee \neg z_3$. A similar substitution can be done for a clause containing only negative literals. It is easy to see that resulting formula is a non-monotone 3-CNF formula. Further, the new formula is satisfiable iff the original formula is satisfiable.

We now prove the NP-hardness of detecting a singular 2-CNF predicate. Consider a non-monotone 3-CNF formula with clauses C_i, $1 \leq i \leq m$. We construct a computation and a singular 2-CNF predicate defined on consistent cuts of the computation as follows. Without loss of generality, assume that each clause has at least two literals—a lone literal in a clause has to be assigned value true in any satisfying assignment. For each clause C_i in the formula, there are two processes p_i^1 and p_i^2 with boolean variables x_i^1 and x_i^2, respectively, in the computation. Initially, all variables evaluate to false. We add the clause $x_i^1 \vee x_i^2$ to the predicate. Now, we describe the local computations of processes in the computation. There is one true event for each literal in the formula. There are two cases to consider: either $|C_i| = 2$ or $|C_i| = 3$.

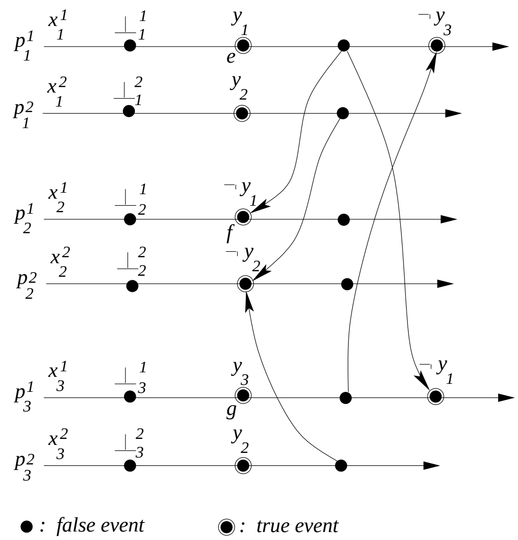

Figure 3. The transformation for $(y_1 \vee y_2 \vee \neg y_3) \wedge (\neg y_1 \vee \neg y_2) \wedge (\neg y_1 \vee y_2 \vee y_3)$.

Case 1 [$|\mathbf{C_i}| = 2$]: Let $C_i = l_i^1 \vee l_i^2$. The local computations of processes p_i^1 and p_i^2 consist of a true event, corresponding to literals l_i^1 and l_i^2, respectively, followed by a false event.

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

Given a consistent cut of the computation that satisfies the singular 2-CNF predicate, a satisfying assignment for the corresponding non-monotone 3-CNF formula is obtained by assigning true value to a literal if the cut passes through the true event corresponding to the literal. To ascertain that the assignment is consistent, that is, no two conflicting literals are assigned value true, we need to ensure that no two true events corresponding to conflicting literals (such as events e and f in Figure 3) are consistent, thereby guaranteeing that no consistent cut passes through both such events. To that effect, we add an arrow from the successor of the true event corresponding to the positive literal (such as event e) to the true event corresponding to the negative literal (such as event f).

We claim that the computation does not have any cycles and two true events are inconsistent iff the corresponding literals are conflicting. The latter equivalence ensures that if two literals can simultaneously be assigned value true (such as y_1 and y_3) then there does exist a consistent cut that passes through both the corresponding true events (such as events e and g in Figure 3). Observe that each arrow is from the successor of the true event corresponding to a positive literal, which is a false event, to the true event corresponding to the conflicting negative literal. Thus each external event is either a send event or a receive event but not both. Further, when a process contains two true events, the true event corresponding to the positive literal precedes the true event corresponding to the negative literal. Therefore if a process contains both send and receive events, the send event precedes the receive event. As a result, there is no outgoing edge after an incoming edge on any process, that is, there are no dependencies between true events due to transitivity. Thus the computation is free of cycles and two true events are inconsistent iff the corresponding literals are conflicting.

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

Theorem 1 *Detecting a singular 2-CNF predicate is NP-complete in general.*

Corollary 2 *Detecting a conjunction of clauses of the form x_i relop x_j, where each x_i is an integer variable and relop $\in \{<, \leq, >, \geq, \neq\}$, such that no two clauses contain variables from the same process is NP-complete in general.*

The above corollary states that even detecting predicates such as $(x_1 < x_2) \wedge (x_3 < x_4) \wedge \cdots \wedge (x_{n-1} < x_n)$, where each x_i is an integer variable on process p_i, is NP-complete in general. The proof involves a simple transformation from a singular 2-CNF predicate. Consider a clause $x_i \vee x_j$. We define integer variables z_i and z_j such that z_i is 0 when x_i is false and is -1 otherwise. Similarly, z_j is 0 when x_j is false and is 1 otherwise. It can be easily shown that $x_i \vee x_j$ iff $z_i < z_j$.

3.2. Efficient algorithm for special cases

In [17], Tarafdar and Garg considered extension of the Lamport's happened before model for predicate detection, called *strong causality model*, that allows events on a process to be partially ordered. For this model, they presented an algorithm for detecting a conjunctive predicate when either all receive events on every process are totally ordered or all send events on every process are totally ordered (CPDSC - Conjunctive Predicate Detection in Strong Causality Model). The observation 1 enables us to view each group as a *meta-process* with events on it as partially ordered. Thus CPDSC algorithm can be applied to solve our problem in a straightforward fashion. However, as in their case, either all receive events on every meta-process should be totally ordered, that is, the computation is *receive-ordered*, or all send events on every meta-process should be totally ordered, that is, the computation is *send-ordered*. We give a brief description of the algorithm in this paper assuming that the computation is receive-ordered. The proof of correctness and other details can be found in [17].

For happened-before model, Garg and Waldecker [9] gave a polynomial time algorithm for detecting a conjunctive predicate (CPDHB - Conjunctive Predicate Detection in Happened-Before Model). Note that given a set of true events, one from each process, either the events in the set are pairwise consistent or there exist events e and f in the set such that $e.succ$ happened before f. Since events on a process are totally ordered in happened-before model, e is also inconsistent with every event on the process that occurs after f. This allows us to eliminate e from consideration in a scan of the computation from left to right, thereby giving an efficient algorithm for the predicate detection.

Since events on a meta-process are, in general, not totally ordered, CPDHB algorithm cannot be applied directly. However, if the computation is receive-ordered then it satisfies property P1 that enables an efficient algorithm to be developed. Consider a computation $E_{\prec}$. We first extend the partial order $\prec$ as follows. For two independent events e and f on a meta-process such that f is a receive event, we add an arrow from e to f. It can be proved that the added arrows do not create any cycle [17]. We then linearize the new partial order thus generated to obtain a total order on all events, say $<$. It can be verified that the computation satisfies the following property:

Property P1 *Given events e, f and g such that events f and g are on the same meta-process but events e and f are on different meta-processes, we have,*

$$(e \prec f) \wedge (f < g) \;\Rightarrow\; e \prec g$$

Thus given events e and f on different meta-processes such that $e.succ \prec f$, by virtue of property P1, e is also inconsistent, with respect to $\prec$, with every event g that occurs after f, with respect to $<$, on the same meta-process (as f). Since events on a meta-process are totally ordered with respect to $<$, we can eliminate e from consideration in a scan of $E_<$ from left to right. This gives us an efficient algorithm to detect the given predicate.

3.3. Algorithms for the general case

For the general case, when the computation is neither receive-ordered nor send-ordered, we construct subsets of processes with one process from each group and apply CPDHB algorithm to each such subset. Since there are k

processes in each group, the number of such subsets is at most $k^{n/k}$. Therefore the complexity of the algorithm is $O((n^2m) \cdot k^{(n/k)-2})$, where m is the maximum number of events on each process and $O((n/k)^2m)$ is the complexity of invoking CPDHB algorithm once. Note that the expression $k^{(n/k)-2}$ attains its maximum value at $k = 2$ for a fixed n. The worst-case time complexity of the existing techniques is $O(m^n)$ making them exponentially worse than our algorithm.

Alternatively, we can divide events in each group into a set of chains (of events) that cover all true events in that group - each true event belongs to at least one chain. We then construct subsets of chains containing one chain from each group. Finally, we apply CPDHB algorithm to each such subset treating each chain in the subset as a separate process for detection purposes. Note that the minimum number of chains needed to cover all true events in a group is upper bounded by k.

4. Detecting $\mathrm{Possibly}:(=\mathbf{k})$

First, we establish the NP-completeness of observing $possibly:(=k)$ in general. Next, we present a polynomial-time algorithm for the special case when each x_i can be incremented or decremented by at most one at each step.

4.1. NP-completeness result

The problem is in NP because the general problem of observing an arbitrary boolean expression is in NP [4]. To prove its NP-hardness, we reduce an arbitrary instance of the subset sum problem [6, problem SP13] to an instance of detecting $possibly:(=k)$. The subset sum problem is defined as follows:

Subset Sum Problem: Given a finite set A, size $s(a_i) \in Z^+$ for each $a_i \in A$ and a positive integer B, does there exist a subset $A' \subseteq A$ such that the sum of the sizes of the elements in A' is exactly B?

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

Theorem 3 *Detecting $possibly:(=k)$ when each x_i can be modified (incremented or decremented) by an arbitrary amount at each step is NP-complete in general.*

4.2. Efficient algorithm for the special case

Our algorithm for the special case is based on monitoring predicates $possibly:(\leq k)$ and $possibly:(\geq k)$. Efficient algorithms to observe these predicates can be found elsewhere [3, 18].

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

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

Observation 2 *Let C and C' be consistent cuts such that C' is obtained from C by executing at most one event. Then* $|sum(C') - sum(C)| \leq 1$.

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

Theorem 4 *Let C and C' be consistent cuts such that there is a path π from C to C' in the computation. Then, for each v,*

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

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

The central idea behind the algorithm for detecting $possibly:(=k)$ is to find a pair of consistent cuts C

and C', if they exist, such that C' is reachable from C and k lies in $range(sum(C), sum(C'))$. Theorem 4 then guarantees the existence of a consistent cut that satisfies $x_1 + x_2 + \cdots + x_n = k$. The consistent cut C is always set to the initial consistent cut $E.\bot$. The advantage is that every consistent cut of a computation is reachable from the initial consistent cut. The next lemma gives sufficient conditions for $possibly:(=k)$ to hold in a computation.

Lemma 5 *Let $E_{\prec}$ be a computation. Then,*

$$(sum(E.\bot) \le k) \wedge (possibly:(\ge k)) \Rightarrow possibly:(=k), \text{ and}$$

$$(sum(E.\bot) \ge k) \wedge (possibly:(\le k)) \Rightarrow possibly:(=k)$$

Proof: Assume that the conjunction $(sum(E.\bot) \le k) \wedge (possibly:(\ge k))$ holds. Using Theorem 4, with $C = E.\bot$ and C' = "some consistent cut with $sum(.)$ at least k", we can deduce that there is a consistent cut D such that $sum(D) = k$. Observe that C' exists because $possibly:(\ge k)$ is true. Further, Theorem 4 is applicable since C' is reachable from C and $sum(C) = sum(E.\bot) \le k \le sum(C')$ implying that $k \in range(sum(C), sum(C'))$. Thus $possibly:(=k)$ holds and, therefore, $(sum(E.\bot) \le k) \wedge (possibly:(\ge k))$ implies $possibly:(=k)$. Likewise, $(sum(E.\bot) \ge k) \wedge (possibly:(\le k))$ implies $possibly:(=k)$. ■

The following lemma presents sufficient conditions for $definitely:(=k)$ to hold in a computation. The proof is similar to the proof of Lemma 5 and therefore has been omitted.

Lemma 6 *Let $E_{\prec}$ be a computation. Then,*

$$(sum(E.\bot) \le k) \wedge (definitely:(\ge k)) \Rightarrow definitely:(=k), \text{ and}$$

$$(sum(E.\bot) \ge k) \wedge (definitely:(\le k)) \Rightarrow definitely:(=k)$$

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

Theorem 7 *Let $E_{\prec}$ be a computation. Then,*

(1) $possibly:(=k) \equiv$
$$(sum(E.\bot) \le k) \wedge (possibly:(\ge k)) \bigvee (sum(E.\bot) \ge k) \wedge (possibly:(\le k)), \text{ and}$$

(2) $definitely:(=k) \equiv$
$$(sum(E.\bot) \le k) \wedge (definitely:(\ge k)) \bigvee (sum(E.\bot) \ge k) \wedge (definitely:(\le k))$$

Proof: *(1)* Follows from the fact that $possibly:(=k)$ implies $possibly:(\le k)$ and $possibly:(\ge k)$, the disjunction $(sum(E.\bot) \le k) \vee (sum(E.\bot) \ge k)$ is a tautology and Lemma 5.

(2) Follows from the fact that $definitely:(=k)$ implies $definitely:(\le k)$ and $definitely:(\ge k)$, the disjunction $(sum(E.\bot) \le k) \vee (sum(E.\bot) \ge k)$ is a tautology and Lemma 6. ■

Observe that the final consistent cut is reachable from every consistent cut of a computation. Thus an alternate set of necessary and sufficient conditions for $possibly:(=k)$ and $definitely:(=k)$ based on final consistent cut can be defined.

4.3. Applications

Recall that *possibly* distributes over disjunction. Some examples of predicates that can be expressed as disjunction of predicates of the form $x_1 + x_2 + \cdots + x_n$ exactly equals k are:

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

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

Symmetric Predicates: A predicate of n boolean variables $p(x_1, x_2, \ldots, x_n)$ is called *symmetric* iff it is invariant under any permutation of its variables. Some examples of symmetric predicates are $x \wedge y$, $x \vee y$ and $(x \wedge y) \vee (\neg x \wedge \neg y)$.

The necessary and sufficient condition for a predicate $p(x_1, x_2, \ldots, x_n)$ to be symmetric is that it may be specified by a set of numbers $\{a_1, a_2, \ldots, a_m\}$, where $0 \le a_i \le n$ and $m \le n + 1$, such that it assumes value true when and only when, for some i, exactly a_i of the variables are true. For example, the symmetric predicate $(x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$ is logically equivalent to the predicate $(x + y + z = 1) \vee (x + y + z = 2)$, where false and true are represented by 0 and 1, respectively, for the purposes of evaluating $x + y + z$. The proof of this result can be found elsewhere [12, page 174]. Since, *possibly* distributes over disjunction, $possibly: b$ when b is a symmetric predicate can be efficiently computed using Theorem 7. Some examples of symmetric predicates that arise in distributed systems are:

- exclusive-or of local predicates: $x_1 \oplus x_2 \oplus \cdots \oplus x_n \equiv \bigvee_{k \text{ is odd}} (x_1 + x_2 + \cdots + x_n = k)$.
- not all x_i's are equal: $(x_1 \vee x_2 \vee \cdots \vee x_n) \wedge (\neg x_1 \vee \neg x_2 \vee \cdots \vee \neg x_n) \equiv \bigvee_{k \in A} (x_1 + x_2 + \cdots + x_n = k)$, where $A = [1 \ldots (n-1)]$.

5. Conclusion

Predicate detection is a fundamental problem in asynchronous distributed systems. This problem arises in various contexts such as design, testing and debugging, and fault-tolerance of distributed programs. In this paper, we solve the previously open problems in predicate detection proposed in [7]. In particular, we establish that the problem of determining whether there exists a consistent cut of a computation that satisfies a singular k-CNF predicate is NP-complete in general when k is at least 2. Our result bridges the gap between the known tractability [9] and intractability [3, 15] results in detecting conjunction of clauses (see Figure 1). Furthermore, the result can be used to establish the intractability of other related interesting problems (see Corollary 2). A polynomial-time algorithm to find the consistent cut, if it exists, that satisfies a singular k-CNF predicate for special cases is provided. We also give algorithms albeit exponential that can be used to achieve an exponential reduction in time over existing techniques for solving the general version.

Furthermore, we present an algorithm to determine whether there exists a consistent cut of a computation for which the sum $x_1 + x_2 + \cdots + x_n$ exactly equals some constant k, where each x_i is an integer variable on process p_i such that it is incremented or decremented by at most one at each step. As a corollary, any symmetric global predicate on boolean variables can now be observed. Additionally, the problem is proved to be NP-complete if each x_i can be changed by an arbitrary amount at each step. Our results build upon and, in some sense, complete the work described in [3, 18].

References

[1] L. Bouge. Repeated Snapshots in Distributed Systems with Synchronous Communication and their Implementation in CSP. *Theoretical Computer Science*, 49:145–169, 1987.

[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] C. Chase and V. K. Garg. On Techniques and their Limitations for the Global Predicate Detection Problem. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 303–317, France, Sept. 1995.

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

[5] R. Cooper and K. Marzullo. Consistent Detection of Global Predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, California, 1991.

[6] M. R. Garey and D. S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1991.

[7] V. K. Garg. Observation and Control for Debugging Distributed Computations. In *Proceedings of the International Workshop on Automated Debugging (AADEBUG)*, pages 1–12, Linköping, Sweden, 1997. Keynote Presentation.

[8] V. K. Garg and B. Waldecker. Detection of Unstable Predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, May 1991.

[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, Mar. 1994.

[10] 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.

[11] R. Jegou, R. Medina, and L. Nourine. Linear Space Algorithm for On-line Detection of Global Predicates. In J. Desel, editor, *Proceedings of the International Workshop on Structures in Concurrency Theory (STRICT)*. Springer-Verlag, 1995.

[12] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw-Hill, 2nd edition, 1978.

[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] M. Spezialetti and P. Kearns. Efficient Distributed Snapshots. In *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS)*, pages 382–388, 1986.

[15] S. D. Stoller and F. Schnieder. Faster Possibility Detection by Combining Two Approaches. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, France, Sept. 1995.

[16] S. D. Stoller, L. Unnikrishnan, and Y. A. Liu. Efficient Detection of Global Properties in Distributed Systems Using Partial-Order Methods. In *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 264–279. Springer-Verlag, July 2000.

[17] A. Tarafdar and V. K. Garg. Addressing False Causality while Detecting Predicates in Distributed Programs. In *Proceedings of the 9th International Conference on Distributed Computing Systems (ICDCS)*, pages 94–101, Amsterdam, The Netherlands, May 1998.

[18] 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.