Efficient Detection of Restricted Classes of Global Predicates

Craig M. Chase* Vijay K. Garg†

Parallel and Distributed Systems Laboratory
email: pdslab@ece.utexas.edu
Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, TX 78712

Abstract

We show that the problem of predicate detection in distributed systems is NP-complete. We introduce a class of predicates, *linear predicates*, such that for any linear predicate B there exists an efficient detection of the least cut satisfying B. The dual of linearity is *post-linearity*. These properties generalize several known properties of distributed systems, such as the set of consistent cuts forms a lattice, and the WCP and GCP predicate dectection results given in earlier work.

We define a more general class of predicates, *semi-linear predicates*, for which efficient algorithms are known to detect whether a predicate has occurred during an execution of a distributed program. However, these methods may not identify the least such cut. Any stable predicate is an example of a semi-linear predicate. In addition, we show that certain unstable predicates can also be semi-linear, such as mutual exclusion violation.

Finally, we show application of max-flow to the predicate detection problem. This result solves a previously open problem in predicate detection, establishing the existence of an efficient algorithm to detect predicates of the form $x_1 + x_2 \ldots + x_N < k$ where x_i are variables on different processes, k is some constant, and N is larger than 2.

keywords: distributed debugging, predicate detection, unstable predicates.

*supported in part by the Texas Instruments/Jack Kilby Faculty Fellowship and by NSF Grant CCR-9409736

†supported in part by the NSF Grant CCR-9110605, a TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant

1 Introduction

Detection of a global predicate is a fundamental problem in distributed computing. This problem arises in many contexts such as designing, testing and debugging of distributed programs. For example, the detection of global predicate arises in implementing the most basic command of a debugging system:"stop the program when the predicate q is true." To stop the program, it is necessary to detect the predicate q; a non-trivial task if q requires access to the global state.

There have been three approaches in solving the detection of global predicates. The first approach is based on the global snapshot algorithm by Chandy and Lamport [CL85, Bou87, SK86]. Their approach requires repeated computation of consistent global snapshots of the computation till the desired predicate becomes true. This approach works only for stable predicates, that is, predicates which do not turn false once they become true. If the desired predicate q were not stable then their approach would fail because q may turn true only between two successive snapshots. Further, their approach does not provide any indication as to when the snapshot needs to be taken. Thus, it may either result in excessive overhead when the snapshots are taken too often, or in significant delay between the occurrence and the detection of the predicate q.

The second approach to global predicate detection is based on the construction of the lattice of global states. This approach, first presented by Cooper and Marzullo [CM91], allows user to detect *definitely*: q and *possibly*: q where q is any predicate defined on a single global state. The predicate *possibly*: q is true if in the lattice of global states there is a path from the initial global state to the final global state in which q is true in some intermediate state. The predicate *definitely*: q is true if q becomes true in all paths from the initial state to the final state. This approach works even for unstable predicates. However, given n processes each with m "relevant" local states, their approach requires exploring $O(m^n)$ possible global states in the worst case.

The third approach is based on exploiting the structure of the predicate q. This approach, instead of building the lattice, directly uses the computation to deduce if q became true. For example, [GW94, GW92] present algorithms to detect *possibly*: q and *definitely*: q of complexity $O(n^2m)$ when q is a conjunction of local predicates. Similarly, [TG93] presents an efficient algorithm to detect $x_1 + x_2 < k$ where x_1 and x_2 are variables on different processes. In a recent paper [SS95], Stoller and Schneider propose combining this approach with that of Cooper and Marzullo. Their method may require exponential complexity to detect some predicates.[1]

In this paper, we study techniques and limits of the third approach. This paper makes the following contributions.

- We show that a detection algorithm for conjunctive predicates cannot be generalized to any arbitrary boolean expression of local predicates. In particular, the problem of detecting whether a boolean expression became true in a distributed computation is an NP-complete problem. The problem stays NP-

[1] Stoller and Schneider's method is exponential in the size of the *fixed set*. The cardinality of the fixed set is at most $n - 1$.

complete even when processes do not communicate with each other and each process executes a single instruction.

- We define a property on the space of boolean predicates that we call *linearity*. We show that there exists a polynomial algorithm to detect the least global cut that satisfies a given linear boolean predicate. We also show that the set of global cuts satisfying a boolean predicate B is an inf-lattice if and only if B is a linear boolean predicate. Thus, linearity captures the class of predicates for which efficient detection of the *least* satisfying cut is possible. For example, the monotonicity condition on channel predicates [GCKM95] is a special case of linearity.

- By considering the dual property of linearity, we get a necessary and sufficient condition for a given set of global cuts to be a lattice. This generalizes many earlier results. For example, the fact that the set of all recoverable cuts form a lattice [JZ90] is an easy consequence of our result.

- When the linearity property does not hold for a predicate B, we show that a weaker property, called *semi-linearity*, is sufficient to permit detection of B with a polynomial algorithm. However, it is not, in general, possible to detect the least satisfying cut when B is only semi-linear. The class of semi-linear predicates subsumes stable predicates [CL85].

- Finally, we show that for a third class of predicates, max-flow technique can be used to detect them. In particular, we give an efficient algorithm to detect predicates of the form $x_1 + x_2 + ... + x_N < C$. This solves an open problem in [TG93] where the problem was solved for $N = 2$. Also see [BR94] where it is remarked that the technique in [TG93] does not appear to be generalizable to more than two processes.

The techniques presented in this paper can be used in distributed debugging systems for implementing breakpoints, in distributed fault monitoring systems for detecting an erroneous state reached by a distributed program, and in the design of distributed algorithms by optimizing the general predicate detection algorithm.

We have restricted ourselves to the global predicates defined on a *single* cut of the distributed computation. Other researchers have also considered predicates that involve multiple cuts. For example, [MC88, GW92] discuss linked predicates, [HPR93, BR94] discuss atomic sequences, and [FRGT94] discuss regular patterns. We refer the reader to [BM94, SM94] for surveys of stable and unstable predicate detection.

This paper is organized as follows. Section 2 describes our model of execution of a distributed program. We use the notion of a *deposet* to model an execution. Section 3 proves the intractability of the general problem. Section 4 describes the linearity property and its applications. Section 5 describes the semi-linearity property. Section 6 describes use of max-flow algorithms for detecting "Bounded Sum" predicates. Finally, Section 7 presents the concluding remarks.

2 Our Model of the Execution of a Distributed Program

We model the execution of a sequential process as a sequence of distinct states. For each state, s, the program prescribes what action will be taken to transition to next state. A distributed system consists of a set of N processes $P \stackrel{\text{def}}{=} \{P_1, \ldots, P_N\}$. Processes do not share any clock or memory; they communicate and synchronize with each other by messages over a set of channels. We assume that messages are not lost, altered, or spuriously introduced into a channel. We do not assume that channels are FIFO.

We limit the type of actions any process P_i may take to:

A1. Compute new values for some subset of the program variables. — We denote the set of program variables for process P_i as X_i.

A2. Send a message on channel C_{ij} for some $j : 1 \leq j \leq N$ — The contents of the message can be any tuple of values of variables in X_i.

A3. Receive a message from channel C_{ji} for some $j : 1 \leq j \leq N$ — we assume that receives are blocking.

We permit the value of program variables to change only during the transitions between states. Thus, any state s from process P_i defines a unique value for all variables in X_i. We use S_i to denote the set of states generated by P_i in one execution of the program. Similarly, we permit the contents of a channel C_{ij} to change only during transitions between states on P_i or P_j. Thus, any two states $s \in S_i$ and $t \in S_j$ uniquely define the set of tuples (messages) in channel C_{ij}. We say that for two states s and t, $s \prec_{im} t$ if and only if s *immediately* precedes t in some process P_i. If $s \prec_{im} t$ then exactly one of the actions, A1, A2 or A3 occurs between s and t. We define the initial and final states on each process as: $Init(i) \stackrel{\text{def}}{=} \min S_i$ and $Final(i) \stackrel{\text{def}}{=} \max S_i$. We use $s \prec t$ to denote that s precedes t (not necessarily immediately).

We say that $s \rightsquigarrow t$ (for states $s \in S_i$ and $t \in S_j$) if and only if process P_i transitions from s to some other state by sending a message to P_j and process P_j transitions from some state to t by receiving that message. Following Lamport, we define the causally-precedes relation, $\rightarrow$, (also known as "happened before") as the transitive closure of $\{\prec_{im}\} \cup \{\rightsquigarrow\}$.

The set of states $S \stackrel{\text{def}}{=} \cup_i S_i$ and the relation $\rightarrow$ form an irreflexive partial order. More specifically, $(S, \rightarrow)$ is a deposet (decomposed partially ordered set) [Gar92, TG93]. The execution rules governing transitions between states lead to the following definition:

Definition 1 *A deposet is a tuple $(S_1, \ldots S_N, \rightsquigarrow)$ such that $(S, \rightarrow)$ is an irreflexive partial order which satisfies:*

D1. $\forall u, \forall i : u \in S, 1 \leq i \leq N : u \not\rightsquigarrow Init(i)$

D2. $\forall u, \forall i : u \in S, 1 \leq i \leq N : Final(i) \not\rightsquigarrow u$

D3. $\forall s, t \in S : s \prec_{im} t \Rightarrow \mid \{u \mid s \rightsquigarrow u \vee u \rightsquigarrow t\} \mid \leq 1$

(D1) says that no state happens before the initial state of any process. Similarly, (D2) says that any final state does not happen before any state. (D3) says that there is at most one message either sent or received between any two consecutive states.

An important concept for deposets is that of a consistent cut. A *cut* is a subset of S containing exactly one state from each sequence S_i. Given two states $x, y \in S$, we say that $x||y$ iff $(x \not\rightarrow y) \wedge (y \not\rightarrow x)$. These two states are then called *concurrent*. A subset $G \subset S$ is *consistent* (denoted by consistent(G)) iff $\forall x, y \in G : x||y$. Since each sequence S_i is totally ordered, it is clear that if $|G| = N$ and consistent(G) then G must include exactly one state from each S_i, i.e., G is a consistent cut.

A consequence of the Definition 1 is the following.

Lemma 2 *For any state s and any process P_i, there exists a non-empty sequence of consecutive states called the "*interval *concurrent to s on P_i" and denoted by $I_i(s)$) such that:*

1. $I_i(s) \subseteq S_i$ *— i.e., the interval consists of only states from process P_i, and*
2. $\forall t \in I_i(s) : t||s$ *— i.e., all states in the interval are concurrent with s.*

Proof: If s is on P_i, then the lemma is trivially true. The interval consists of exactly the set $\{s\}$ (which is concurrent with itself). So we assume that s is not on P_i. Define $I_i(s).lo = \min\{v | v \in S_i \wedge v \not\rightarrow s\}$. This is well-defined since $Final(i) \not\rightarrow s$ due to (D2). Similarly, on account of (D1), we can define $I_i(s).hi = \max\{v | v \in S_i \wedge s \not\rightarrow v\}$.

We show that $I_i(s).lo \preceq I_i(s).hi$. If not, we do a case analysis.

Case 1: There exists $v : I_i(s).hi \prec v \prec I_i(s).lo$. Since $v \prec I_i(s).lo$ implies $v \rightarrow s$ and $I_i(s).hi \prec v$ implies $s \rightarrow v$, we get a contradiction ($v \rightarrow v$).

Case 2: $I_i(s).hi \prec_{im} I_i(s).lo$. From the definition of $I_i(s).lo$, it is easy to see that there must be a message sent from the state previous to $I_i(s).lo$. Similarly, from the definition of $I_i(s).hi$, there exists a message received just after $I_i(s).hi$. However, (D3) prohibits more than one send or receive event between two successive states. Thus, this case is also not possible.

From the above discussion it follows that $I_i(s).lo \preceq I_i(s).hi$. Further, for any state t such that $I_i(s).lo \preceq t \preceq I_i(s).hi$, $t \not\rightarrow s$ and $s \not\rightarrow t$ holds. □

An important property of a deposet is given next.

Theorem 3 *Any consistent subset $G \subset S$ can be extended to a consistent cut. That is, $\forall G : G \subset S :$ consistent$(G) \Rightarrow (\exists H : G \subseteq H$: consistent(H) $\wedge |H| = N)$.*

Proof: It is sufficient to show that when $|G| < N$, there exists a cut $H \supset G$ such that consistent(H) and $|H| = |G| + 1$. Consider any process P_i which does not contribute a state to G. We will show that there exists a state in S_i which is concurrent with all states in G. Let s and t be two distinct states in G. We show that $I_i(s) \cap I_i(t) \neq \emptyset$. If not, w.l.o.g. assume that $I_i(s).hi \prec I_i(t).lo$. As in the proof of Lemma 2, it follows that there exists at least one state, say v, between $I_i(s).hi$ and $I_i(t).lo$ (due to (D3)). This implies that $s \rightarrow v$ (because $I_i(s).hi$ precedes v) and $v \rightarrow t$ (because v precedes $I_i(t).lo$). Thus, $s \rightarrow t$, a contradiction with consistent(G). Thus, $I_i(s) \cap I_i(t) \neq \emptyset$.

Since any interval $I_i(s)$ is a total order, it follows that:

$$\bigcap_{s \in G} I_i(s) \neq \emptyset$$

We now chose any state in $\bigcap_{s \in G} I_i(s)$ to extend G. □

The above property allows the algorithms in later sections to search for consistent subcuts rather than consistent cuts.

3 NP-Completeness of Global Predicate Detection

We define a global predicate as any boolean valued function B of the variables in $X \stackrel{\text{def}}{=} \bigcup_i X_i$ and all tuples (i.e., message) which may be present in the channels. During an execution, each state $s \in S_i$ defines a value for each variable $x \in X_i$. Each pair of states $s \in S_i$ and $t \in S_j$ define a set of tuples for channel C_{ij}. We therefore use the notation $B(G)$ to indicate the value of predicate B in a global state defined by a cut $G = \{s_1, \ldots, s_N\}$. We will ignore channels in the rest of this section. This simplification does not affect the NP-completeness of global predicate detection.

The global predicate detection problem (GLOB) is a decision problem. It takes the form of:

> **Given**: a deposet S of N sequences, a set of variables X partitioned into N subsets $X_1, \ldots, X_N$, and a predicate B defined on X.
>
> **Determine** if there exists a consistent cut $G \in S$ such that $B(G)$ has the value true.

We now show that the predicate detection problem is NP-Complete.

Theorem 3.1 *GLOB is NP-complete.*

Proof : First note that the problem is in NP. A verifier for the problem takes as input a cut G and then determines if that cut is consistent and if the predicate is true. The verification that the cut is consistent can easily be done in polynomial time (for example, using vector clocks [Mat89, Fid89] and examining all pairs of states

from the cut). Therefore, if the predicate itself can be evaluated in polynomial time, then the detection of that predicate belongs to the set NP.

We show NP-completeness of the simplified predicate detection problem where all program variables are restricted to taking the values "true" or "false", and at most one variable from each X_i can appear in B. We reduce the satisfiability problem of a boolean expression to GLOB by constructing an appropriate deposet.

The deposet is constructed as follows. For each variable $u_i \in U$, define a process P_i which hosts variable u_i (i.e., $X_i = \{u_i\}$). Let the sequence S_i consists of exactly two states. In the first state, u_i has the value false. In the second state, u_i has the value true. There are no messages exchanged during the computation (i.e., $\forall s \in S_i, \forall t \in S_j : i \neq j : s||t$).

It is easily verified that the predicate B is true for some cut in S if and only if the expression is satisfiable.

■

The above result shows that detection of a general global predicate is intractable even for simple distributed computation. This implies that the class of predicates must be restricted to allow for efficient detection. The remaining sections discuss three such restricted classes.

4 Linear Predicates

In this section, we describe a class of global predicates for which efficient detection algorithms can be derived. We first define the relation $\leq$ for cuts. Let $\mathcal{G}_S$ (or, simply $\mathcal{G}$) be the set of all cuts for deposet S. For two cuts $G, H \in \mathcal{G}$, we say that $G \leq H$ iff $\forall i : G[i] \leq H[i]$ where $G[i] \in S_i$ and $H[i] \in S_i$ are the states from process P_i in cuts G and H respectively. It is clear that for any deposet S, $(\mathcal{G}, \leq)$ is a lattice.

A key concept in deriving an efficient algorithm is that of a *forbidden* state. Given a deposet S, a predicate B, and a cut $G \subset S$, a state $G[i]$ is called forbidden if its inclusion in any cut H, where $G \leq H$, implies that B is false for H. Formally,

Definition 4 *Given any boolean expression B, we define*

$$\mathit{forbidden}(G, i) \stackrel{\text{def}}{=} \forall H : G \leq H : (G[i] \neq H[i]) \vee \neg B(H)$$

Based on the concept of a forbidden state, we define a predicate B to be linear with respect to deposet S if for any cut G in the deposet, the fact that B is false in G implies that G contains a forbidden state. Formally,

Definition 5 *A boolean predicate B is* linear *with respect to a deposet S iff:*

$$\forall G \in \mathcal{G} : \neg B(G) \Rightarrow \exists i : \mathit{forbidden}(G, i)$$

Observe that the linearity of a boolean predicate also depends on the set $\mathcal{G}$ and, therefore, on the deposet S. We would typically be interested in predicates which are linear for all deposets consistent with a program.

The following is an easy consequence of the definition of linearity.

Lemma 6 *The following are properties of linear predicates:*

1. *If B_1 and B_2 are linear, then so is $B_1 \wedge B_2$.*
2. *If B is defined using variables of a single process, then B is linear.*
3. *The predicate that a cut is consistent is linear. That is, Let $B(G) \equiv \forall i, j : G(i)||G(j)$. Then, B is a linear predicate.*

Proof: We just show the third part. $\neg B(G)$ implies $\exists i, j : G[i] \rightarrow G[j]$. This implies that for all $H \geq G$, $G[i] \rightarrow H[j]$. Thus, we get $\neg(G[i] = H[i]) \vee \neg B(H)$. Thus, B is a linear predicate. □

Observe that as a consequence of Lemma 6, weak conjunctive predicates [GW92] are linear.

4.1 The Least Satisfying Cut Exists for Linear Predicates

Note that any global predicate, B, defines a (possibly empty) subset of cuts $\mathcal{G}_B \subseteq \mathcal{G}$ where B holds for all cuts in $\mathcal{G}_B$. We now show that if B is linear then $\mathcal{G}_B$ is an inf-semilattice. An implication of this result is that the *least* cut satisfying B is well-defined.

Lemma 7 *Let $\mathcal{G}_B \subseteq \mathcal{G}$.*
$\mathcal{G}_B$ is an inf-semilattice iff B is linear with respect to $\mathcal{G}$.

Proof: ($\Leftarrow$) We prove the contrapositive. Assume that B is not linear. This implies that there exists a cut G such that $\neg B(G)$, and $\forall i : \exists H_i \geq G : (G[i] = H_i[i])$ and $B(H_i)$. Consider $Y = \cup_i \{H_i\}$. Note that all elements of $Y \in \mathcal{G}_B$. However, $inf Y$ which is G is not an element of $\mathcal{G}_B$. This implies that $\mathcal{G}_B$ is not an inf-semilattice.

($\Rightarrow$) We again show the contrapositive. Let $Y = \{H_1, H_2, ..H_k\}$ be any subset of $\mathcal{G}_B$ such that its infimum G does not belong to $\mathcal{G}_B$. Since G is infimum of Y, for any i, there exists $j \in 1..k$ such that $G[i] = H_j[i]$. Since $B(H_j)$ is true for all j, it follows that there exists a G for which linearity does not hold. □

Some earlier results can be shown to be special cases of Lemma 7. For example, consider channel predicates as described in [**?**]. Let C denote the state of any channel and M denote any set of messages.

Definition 8 *A channel predicate, $c(C)$, is said to be* monotonic *iff:*

$$\forall C :: \neg c(C) \Rightarrow (\forall M :: \neg c(C \cup M)) \vee (\forall M :: \neg c(S - M))$$

That is, given any channel state, *C*, in which the predicate is false, then either sending more messages is guaranteed to leave the predicate false, or receiving more messages is guaranteed to leave the predicate false. An example of a monotonic predicate is "channel C_{ij} is empty". If this predicate is false (i.e. the channel is not empty), then sending more messages is guaranteed to leave the predicate false. A boolean predicate is called a Generalized Conjunctive Predicate (GCP) iff it can be written as a conjunction of local predicates and monotonic channel predicates. That is,

$$GCP = (l_1 \wedge l_2 \wedge ...l_n \wedge c_1 \wedge c_2 \wedge ...c_e)$$

Note that many classical detection problems in distributed systems, such as termination detection, buffer overflow, and bounding global virtual time, are examples of GCPs.

The following is an easy application of Lemmas 6 and 7.

Theorem 4.1 *Let B be a GCP be such that all of its channel predicates are monotone. Let $(\mathcal{G}, \leq)$ be the set of all global consistent cuts in which the GCP is true. If $G, H \in \mathcal{G}$, then their greatest lower bound is also in $\mathcal{G}$.*

Proof: Note that GCP(G) is true iff

1. G is a consistent cut, and
2. all local predicates are true in G, and
3. all channel predicates are true in G.

Each of the above clauses is linear. □

Example 9 As another example, consider the predicate $x + y \geq k$ where x and y are variables on processes P_1 and P_2, and k is some constant. In general, this predicate is not linear. Figure 1 illustrates this. However, assume that x is known to be monotonically decreasing. In this case, $x + y \geq k$ is linear. Given any cut, if $x + y < k$, then we throw away the state with y variable.

We now discuss detection of linear global predicates. We will assume that given a cut, G, it is efficient to determine whether B is true for G or not. On account of linearity of B, if B is evaluated to be false in some cut G, then we know that there exists a forbidden state in G. We will also assume that there exists an efficient algorithm to determine the forbidden state. With these assumptions, we get:

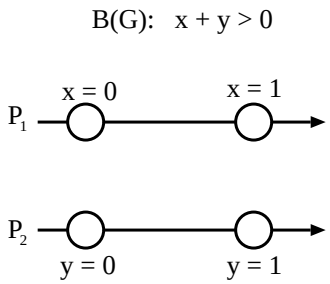

Figure 1: Example of a Non-Linear Predicate

```
(1)   ∀i : G[i] := Init(i);
(2)   while ¬B(G) do
(3)         find i such that forbidden(G, i);
(4)         if (G[i] = Final(i)) then return false
(5)         else G[i] := G[i].next;
(6)   end while;
(7)   return true;
```

Figure 2: An efficient algorithm to detect a linear predicate

Theorem 10 *If B is a linear predicate then there exists an efficient algorithm to determine the least cut that satisfies B (if any).*

Proof: An efficient algorithm to find the *least* cut in which B is true is given in Fig. 2. We search for least cut starting from the initial state. If the predicate is false in the current state (line 2), then we find the process with the forbidden state (line 3). If this is the last state on the process, then we return false else we advance along the process which has the forbidden state (line 5). □

The efficient algorithm can be visualized as searching for the first satisfying cut in the lattice of all cuts by advancing with the help of the forbidden state. Thus, even though there are an exponential number of cuts in the lattice, we explore at most mN cuts where m is the maximum number of states along any process in the deposet.

4.2 Dual Properties

Just as existence of the least cut requires that the predicate B be linear, the existence of the *largest* satisfying cut requires a property that is dual of linearity.

Definition 11 *A predicate B is post-linear iff*

$$\forall G \in \mathcal{G} : \neg B(G) \Rightarrow \exists i : \forall H \leq G : \neg(G[i] = H[i]) \vee \neg B(H)$$

In example 9, if x is known to be monotonically increasing, then the predicate is post-linear.

All the results in the previous section have dual results for post-linear predicates. Thus, B is a post-linear predicate iff $\mathcal{G}_B$ is a sup-semilattice.

Further, there exists an efficient algorithm to find the largest cut for any post-linear predicate. The algorithm in this case starts from the last cut and works its way backwards until it finds a cut which satisfies B. Combining the results from the previous section and their duals, we get:

Theorem 12 *$\mathcal{G}_B$ is a lattice iff B is linear w.r.t. $\mathcal{G}$ and B is also post linear w.r.t. $\mathcal{G}$.*

As an application of Theorem 12, we consider the problem of recovery in a distributed systems. We call a local state *recoverable* if after a failure, the state can be recovered from the disk using a checkpoint and the message log. A cut is called recoverable if all states belonging to that cut are recoverable and the cut is consistent [2]

The following is an easy corollary of the Theorem 12.

Corollary 4.2 *The set of all recoverable cuts is a lattice.*

Proof sketch: Recoverability of a state is local to a process. Any local property is both linear and post-linear. Similarly, the consistency property is linear as well as post-linear.

■

5 Semi-Linear Predicates

Now assume that the given predicate is not linear. The first implication is that we cannot insist on getting the least cut anymore (Lemma 7 states that such a cut may not exist). It is still useful to find *any* cut that satisfies B. We now give a property *semi-linearity*, which is weaker than linearity, such that for every semi-linear predicate there exists an efficient algorithm to determine if there exists at least one cut that satisfies B.

Definition 13 *Given any boolean expression B, we define*

$$\begin{aligned} \textit{semi-forbidden}(G,i) \stackrel{\text{def}}{=} \forall H : \quad G \leq H : \quad & G[i] \neq H[i] \vee \\ & \neg B(H) \vee \\ & \exists K \geq G : B(K) \wedge G[i] \prec K[i] \end{aligned}$$

[2] Note that the notion of consistency in [JZ90] is slightly different from the one discussed in this paper.

Definition 14 *A boolean predicate B is* semi-linear *with respect to a deposet S if:*

$$\forall G \in \mathcal{G} : \neg B(G) \Rightarrow \exists i : \textit{semi-forbidden}(G, i)$$

For an example of a semi-linear predicate, consider the execution of a mutual exclusion algorithm. To ensure that the given execution is proper, we are interested in determining existence of a consistent cut G such that $B(G) \stackrel{\text{def}}{=} \exists i, j : CS(G[i]) \wedge CS(G[j]) \wedge \text{consistent}(G)$. We first use Theorem 3 to reduce the problem to detecting $B(G) = \exists i, j : CS(G[i]) \wedge CS(G[j]) \wedge G[i]||G[j]$. Theorem 3, implies that the subcut $\{G[i], G[j]\}$ can be extended to a consistent cut. Now note that if B is false in G, then:

$$\forall i, j : \neg CS(G[i]) \vee \neg CS(G[j]) \vee \neg(G[i]||G[j])$$

If $\neg CS(G[i])$ holds and there exists a state after $G[i]$, then $G[i]$ is semi-forbidden. Now assume that $CS(G[i])$ is true for all i or $G[i]$ is the last state on process i. Without loss of generality assume the $G[i]$ for which $CS(G[i])$ holds can be sorted (if not, then at least two are concurrent and the algorithm halts). After sorting, the least $G[i]$ is semi-forbidden.

Remark 15 Another property that has been exploited in past is the following. A boolean predicate B satisfies property (STABLE) if $B(G) \wedge \text{consistent}(G) \Rightarrow \forall H : G \leq H \wedge \text{consistent}(H) : B(H)$. Any stable property satisfies semi-linearity.

6 Bounded Sum Predicates

In this section, we describe a technique which can be used to compute predicates defined as a lower bound on the sum of variables in a distributed program. The technique can be generalized for both upper and lower bounds. The predicate $(x_1 + ... + x_N < k)$ belongs to this class of predicates, where x_i's are integers in different processes and k is a constant. The predicate becomes true when the sum of the x_i's falls below the lower bound k. Bounded sum predicates were introduced in [TG93], where they were called "Relation Predicates" and algorithms were presented for the special case when $N = 2$.

Bounded sum predicates are useful for detecting global conditions such as loss of tokens and violations of a limited resource. For example, consider a system in which there are k tokens indicating availability of k resources. If x_i denotes the number of tokens at process P_i, then $\sum_i x_i < k$ indicates loss of one or more tokens. As another example, consider a server which can handle at most k connections at a time. Client processes P_i have variables x_i which indicates the number connections it has with the server. The predicate $(\sum_i x_i > k)$ indicates a potential error.

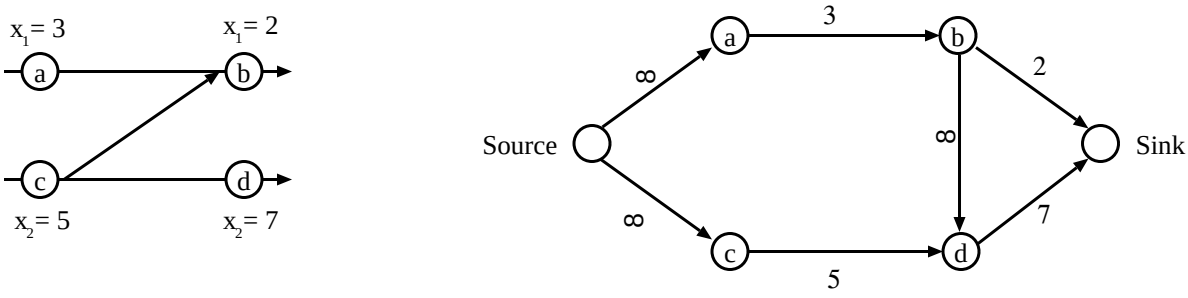

Figure 3: Converting Deposets into Flow Graphs

The predicate to be detected, previously expressed as $(x_1 + x_2 + ... + x_N < k)$, can be stated formally as:

$$\exists G : \text{consistent}(G) : \sum_{s_i \in G} s_i.x_i < k$$

We detect this predicate by computing

$$\min G : \text{consistent}(G) : \sum_{s_i \in G} s_i.x_i$$

and then comparing this value to the constant k.

We transform the deposet into a flow graph such that the max-flow in the graph is equal to the min-value of the deposet. The resulting flow graph G is obtained as follows. $G = (V, E)$, where $V = \bigcup_i S_i \cup \{source, sink\}$ The edge set E is given below.

- First, we add edges from the *source* to all initial states s with the capacity ∞.
- For any two states s and t such that $s \prec_{im} t$, we add an edge between them with capacity $s.x$.
- We add edges from all final states s to the *sink* with the capacity $s.x$.
- For any two states s and t such that $s \rightsquigarrow t$, we first identify the successor to s, $s \prec_{im} s'$. Note that the successor must exist as a consequence of Definition 1. We then add an edge from t to s' with capacity ∞.

Figure 3 shows an example. The original deposet is on the left, and contains two states from each of two processes. The flow graph that we construct is shown on the right. It can easily be seen that the minimum value of $x_1 + x_2$ along any consistent cut is eight. Eight is also the maximum flow of the corresponding flow graph.

The following result gives us a method for computing *min-value* of a deposet.

Theorem 6.1 *The min-value of a deposet S is equal to the min cut of its flow graph G.*

proof sketch: We relate a cut in the flow graph to a cut in the deposet as follows: If edge e connects vertices s and t in G, and if e is part of the cut of flow graph G, then the state corresponding to s is part of the cut in deposet S.

- We first observe that any consistent cut of the deposet partitions the flow graph such that the source and sink are isolated.
- We also note that cut of G has finite value if and only if the cut is a consistent cut of S.

■

Based on the above result, a checker based algorithm can be devised as follows. First, the sequence of states from each process is reduced by replacing the subsequence of states between any two message events with a single state. The value of x_i for this new state is defined as the minimum of x_i over the original states. Second, each process locally maintains the direct dependence relation ($\rightsquigarrow$) for each state. Each process creates a local snapshot for every state, consisting of the value of x_i and the direct dependence information. The local snapshots are sent to a checker process which forms the flow-graph. The checker then runs a max-flow algorithm to find the min cut. If this value is less than k, then the bounded sum predicate is detected.

Max-flow can be solved in $O(|V||E|\log(|V|^2/|E|))$ time by the algorithm due to Goldberg and Tarjan[GT86], In our case, $|V| = |E| = O(mN)$. Therefore, the checker process requires $O(m^2N^2\log(mN))$ time to calculate this cut. The message complexity is only $O(mN)$ messages.

7 Conclusions

We have shown that the general problem of detecting a global predicate is NP-complete. We have defined three classes of predicates — linear, semi-linear and bounded sum. We have given efficient algorithms to detect predicates in these classes.

Acknowledgements

The authors gratefully acknowledge the contributions to this work by Alex Tomlinson. We thank him for many valuable discussions and his insight into the importance of the linearity propery.

References

[BM94] O. Babaoğlu and K. Marzullo. Consistent global states of distributed systems: Fundamental concepts and mechanisms. In Sape Mullender, editor, *Distributed Systems*, pages 55–96. Addison Wesley, New York, NY, 2nd edition, 1994.

[Bou87] L. Bouge. Repeated snapshots in distributed systems with synchronous communication and their implementation in CSP. *Theoretical Computer Science*, 49:145–169, 1987.

[BR94] ¨O. Babaoğlu and M. Raynal. Specification and detection of behavioral patterns in distributed computations. In *Proc. of 4th IFIP WG 10.4 Int. Conference on Dependable Computing for Critical Applications*, San Diego, CA, January 1994. Springer Verlag Series in Dependable Computing.

[CL85] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CM91] 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.

[Fid89] C. J. Fidge. Partial orders for parallel debugging. In *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, volume 24 of *SIGPLAN Notices*, pages 183–194, January 1989.

[FRGT94] E. Fromentin, M. Raynal, V. K. Garg, and A. I. Tomlinson. On the fly testing of regular patterns in distributed computations. In *Proc. of the 23rd Intl. Conf. on Parallel Processing*, St. Charles, IL, August 1994.

[Gar92] V. K. Garg. Some optimal algorithms for decomposed partially ordered sets. *Information Processing Letters*, 44:39–43, November 1992.

[GCKM95] V. K. Garg, C. Chase, R. Kilgore, and J. R. Mitchell. Detecting conjunctive channel predicates in a distribute programming environment. In *Proc. of the International Conference on System Sciences*, volume 2, pages 232–241, Maui, Hawaii, January 1995.

[GT86] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. In *Proc. of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 136–146, 1986.

[GW92] V. K. Garg and B. Waldecker. Detection of unstable predicates in distributed programs. In *Proc. of 12th Conference on the Foundations of Software Technology & Theoretical Computer Science*, pages 253–264. Springer Verlag, December 1992. Lecture Notes in Computer Science 652.

[GW94] V. K. Garg and B. Waldecker. Detection of weak unstable predicates in distributed programs. *IEEE Transactions on Parallel and Distributed Systems*, 5(3):299–307, March 1994.

[HPR93] M. Hurfin, N. Plouzeau, and M. Raynal. Detecting atomic sequences of predicates in distributed computations. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 32–42, San Diego, CA, May 1993. ACM/ONR. (Reprinted in SIGPLAN Notices, Dec. 1993).

[JZ90] D. B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11(3):462–491, September 1990.

[Mat89] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. Elsevier Science Publishers B. V, 1989.

[MC88] B. P. Miller and J. Choi. Breakpoints and halting in distributed programs. In *Proc. of the* 8^{th} *International Conference on Distributed Computing Systems*, pages 316–323, San Jose, CA, July 1988. IEEE.

[SK86] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *Proc. of the* 6^{th} *International Conference on Distributed Computing Systems*, pages 382–388, 1986.

[SM94] R. Schwartz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[SS95] Scott D. Stoller and Fred B. Schneider. Faster possibility detection by combining two approaches. In *Proceedings of the Workshop on Distributed Algorithms, Le Mont Saint Michel, France*, September 1995. available as Cornell University Computer Science Technical Report 95-1511.

[TG93] A. I. Tomlinson and V. K. Garg. Detecting relational global predicates in distributed systems. In *Proc. of the Workshop on Parallel and Distributed Debugging*, pages 21–31, San Diego, CA, May 1993. ACM/ONR.