Detecting Global Predicates in Distributed Computations

Vijay K. Garg*
Department of Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712-1084, USA
garg@ece.utexas.edu
and
Visiting Professor
Computer Science and Engineering Department
Indian Institute of Technology Kanpur
Kanpur-208016, India

Abstract

Global predicate detection is a fundamental problem in distributed computing in the areas of distributed debugging and software fault-tolerance. It requires searching the global state lattice of a computation to determine if any consistent global state satisfies the given predicate. Alagar and Venkatesan's algorithm performs the depth-first-search (DFS) traversal of the lattice and requires $O(nM)$ time and $O(nE)$ space where where n is the number of processes, M is the number of consistent global states and E is the number of events in the computation. In many distributed computing applications, the breadth-first-search (BFS) traversal and the lexical *traversal of the lattice are more useful. We propose a new algorithm that performs the* lexical *traversal of the lattice with $O(nE)$ space and $O(n^2M)$ time complexity. Cooper and Marzullo's algorithm performs a BFS traversal and requires exponential* space. *We show that the time complexity of the BFS traversal can be reduced by exploiting the distributive lattice structure of the global state graph. Further, we show that the BFS can be performed in polynomial space by increasing the time complexity.*

Enumeration of the CGS lattice in the lexical *and the BFS order has applications in combinatorial enumeration besides distributed computing. We show that the algorithms discussed in the paper can be used to efficiently enumerate all subsets of $[n]$ (the set $\{1..n\}$), all subsets of $[n]$ of size k, all permutations, all integer partitions less than a given partition, all integer partitions of a given number, all n-tuples of a product space, and many other families of combinatorial objects.*

Finally, we discuss detection of global predicate under definitely *modality. Cooper and Marzullo's algorithm for this problem requires space exponential in size of the computation. We give a simple algorithm that requires only polynomial space.*

Keywords: Global Predicate Detection, Combinatorial Enumeration, Lattices, Ideals

---

*supported in part by the NSF Grants ECS-9907213, CCR-9988225, Texas Education Board Grant ARP-320, an Engineering Foundation Fellowship, and an IBM grant.

1 Introduction

Global predicate detection is a fundamental problem in distributed computing. It is useful in distributed debugging [GW91, CM91] and software fault-tolerance[TG99]. For debugging a distributed program, it is useful to monitor and stop the execution when the user specified condition, a global predicate, becomes true. For example, the user may specify that the execution should be stopped when $x_1 + x_2 > x_3$ where x_i is a variable on process P_i. Here $(x_1 + x_2 > x_3)$ is a global predicate and the debugger needs to detect the condition and stop the program in a consistent global state that satisfies the condition. In the software fault-tolerance application, an execution of a distributed program may be monitored for a software fault, modeled as a global predicate, and a corrective action taken on detecting the fault. For example, specification of correctness of a program typically includes invariants as safety properties. Violation of a safety property constitutes a software fault and can be detected by global predicate detection algorithms.

Given a distributed computation, the global predicate detection problem asks whether there exists a consistent global state (CGS) [CL85] in which the predicate is true. Informally, a global state is consistent if for any message whose receive event is included in the global state, its send event is also included. For example, consider the computation in Figure 1(a). Its CGS lattice is shown in Figure 1(b). The global state $(a_i a_j)$ denotes that the first process has executed i events and the second process has executed j events in that state. Thus, the global state (21) signifies that P_1 has executed e_1 and e_2 and P_2 has executed f_1. It can alternatively be also viewed as the subset $\{e_1, e_2, f_1\}$. The global state 02 is not consistent because it includes the receive event f_2 but not the send event e_1 which happened before f_2.

Global predicate detection is a hard problem because of combinatorial state explosion. If there are n processes, each with at most k events, then the total number of consistent global states can be as large as $O(k^n)$. Detecting a simple global predicate such as predicates in 2-CNF form even when no two clauses contain variables from the same process is NP-complete in general [MG01a].

The CGS lattice can be traversed in multiple ways as shown in Figure 1(c). Cooper and Marzullo's algorithm[CM91] performs a breadth-first-search (BFS) traversal and requires space proportional to the size of the biggest level of the CGS lattice which, in general, is *exponential* in the size of the computation. Alagar and Venkatesan's algorithm[AV01] performs a depth-first-search (DFS) traversal of the lattice and requires $O(nM)$ time and $O(nE)$ space where n is the number of processes, M is the number of consistent global states and E is the number of events in the computation. Their technique achieves polynomial space by avoiding storage of the intermediate global states during traversal. The main idea is that all predecessors of a global state are ordered and a CGS is visited only from the predecessor that has the maximum value among all predecessors. Using this idea, their algorithm ensures that every CGS is visited exactly once without explicit storage of consistent global states traversed. We note here that if the poset is such that its covering graph [DP90] has maximum in-degree of c, then Alagar and Venkatesan's algorithm can be implemented in only $O(cE)$ space. The largest possible value of c is n.

In this paper, we propose a new algorithm that performs the *lexical* traversal of the lattice with $O(nE)$ space and $O(n^2M)$ time complexity. Lexical traversal is the natural dictionary order used in many applications. It is especially important in distributed computing applications because the user is generally interested in the CGS that satisfies the predicate to be minimal with respect to some order. The lexical order gives a total order on the set of consistent global states based on priorities given to processes and the CGS detected by this traversal gives the lexically smallest CGS.

For some other distributed computing applications, the BFS traversal is more appropriate. The component-wise order (or the vector clock order as shown in Figure 1(b)) imposes a partial order on all global states and the BFS traversal returns a CGS with the minimum number of events executed. We give two algorithms for the BFS traversal. The first algorithm has similar space requirements as the algorithm by Cooper and Marzullo but speeds up enumeration of consistent global states by exploiting the fact that the global state graph is a distributive lattice. The second algorithm shows that the BFS traversal can be performed in space proportional to the size of the computation which may be significantly (exponentially) smaller than the size of a level of a lattice. It is based on efficient enumeration of a level set by enumerating all *integer compositions*.

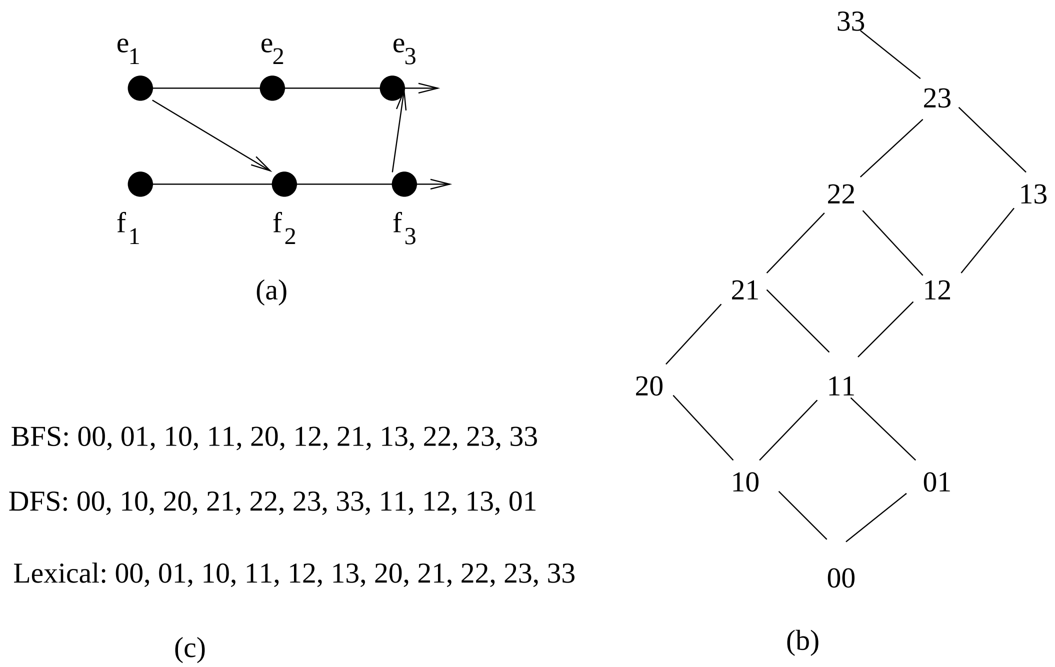

Figure 1: (a) A computation (b) Its lattice of consistent global states (c) Traversals of the lattice.

We note that all the traversals discussed in the paper are straightforward if one explicitly generates the graph of the CGS lattice. Since this graph is exponential in size, the challenge is to traverse the graph without storing either the complete graph or a major part of it.

Enumerating ideals in the lexical and the BFS order is also useful in combinatorial applications. In [Gar02] we have shown that many families of combinatorial objects can be mapped either to the CGS lattices or to the level sets of the CGS lattices of appropriate computations. In enumerating combinatorial objects, one is usually interested in the lexical order. When the mapping is to the CGS lattice, one can use the lexical traversal to enumerate the family of combinatorial objects in the lexical order. When the mapping is to a level set of the CGS lattice, one can use the BFS traversal of the CGS lattice to enumerate the family of combinatorial objects. Thus, we show that algorithms for lexical and BFS traversal discussed in the paper can also be used to efficiently enumerate all subsets of $[n]$, all subsets of $[n]$ of size k, all permutations, all integer partitions less than a given partition, all integer partitions of a given number, and all n-tuples of a product space. Note that [SW86] gives different algorithms for these enumerations. Our algorithm is generic and by instantiating it with different posets all the above combinatorial lexical enumeration can be achieved.

Finally, we discuss detection of global predicate under *definitely* modality. A predicate B is definitely true in the CGS lattice if all paths from the initial CGS to the final CGS go through a CGS in which B is true. Cooper and Marzullo's algorithm for detecting $definitely : B$ requires exponential space. We give a simple algorithm that requires only polynomial space.

2 Model and Background

We assume a loosely-coupled message-passing asynchronous system without any shared memory or a global clock. A distributed program consists of n processes denoted by $\{P_1, P_2, \ldots, P_n\}$ communicating via asynchronous messages. In this paper, we will be concerned with a single computation of a distributed program. We do not make any assumptions about FIFO nature of the channels.

The execution of a single process in a computation results in a sequence of events totally ordered by the relation *occurred before*. We use $e < f$ to denote that e occurred before f on some process. To impose an order relation on events across processes, we use Lamport's happened-before relation [Lam78]. The happened-before relation between two events e and f can be formally stated as: $e \rightarrow f$ iff $e < f$ 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 define a distributed computation as the partially ordered set (poset) consisting of the set of events together with the happened before relation and denote it by $(P, \rightarrow)$. Given a poset $(P, \rightarrow)$, we say that f *covers* e if $e \rightarrow f$ and there is no g such that $e \rightarrow g$ and $g \rightarrow f$. Two events e and f are concurrent in $(P, \rightarrow)$ if $e \not\rightarrow f$ and $f \not\rightarrow e$. This is denoted by $e||f$.

A global state (or, a cut) is a subset $G \subseteq P$ such that

$$f \in G \wedge e < f \Rightarrow e \in G$$

A consistent global state (CGS) of a computation $(P, \rightarrow)$ is a subset $G \subseteq P$ such that

$$f \in G \wedge e \rightarrow f \Rightarrow e \in G$$

For a global state G, $G[i]$ denotes the maximal event of P_i in G (i.e. there is no event e in G such that $G[i]$ occurred before e). Although we have defined global states as subsets, they can equivalently be defined using vectors of local states as shown in Figure 1. In this case $G[i]$ equals the number of events executed by P_i in G. A CGS captures the notion of a possible global state and is identical to an *order ideal (or simply an ideal)* used in the theory of partial orders [DP90]. From now on, we use *ideals* and *consistent global states* interchangeably.

A *global predicate* (or simply a *predicate*) is a boolean-valued function defined on the set of consistent global states. We say that $B(G)$ (B holds in the CGS G) if the function evaluates to true in G.

A *lattice* is a poset L such that for all $x, y \in L$, the least upper bound of x and y exists, called the *join* of x and y (denoted by $x \sqcup y$); and the greatest lower bound of x and y exists, called the *meet* of x and y (denoted by $x \sqcap y$). A lattice L is *distributive* if for all $x, y, z \in X$: $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$.

Given a computation P, we impose an order on the set of global states as follows. Given two consistent global states, G and H, we say that G is less than H iff $G \subseteq H$. It is well known in the lattice theory that the set of all ideals form a distributive lattice under $\subseteq$ relation.

For some applications, instead of enumerating all ideals of a poset we may be interested in enumerating ideals in a certain *level set*. To define *level sets*, first define a poset to be *ranked* if for each element $x \in P$, one can assign a non-negative integer, $rank(x)$, such that if y covers x, then $rank(y) = rank(x) + 1$. The set of all elements in P with rank i are called the level set with rank i. Every distributive lattice is a ranked poset [Sta86].

The notation used in this paper is summarized in Figure 2.

n	The number of processes
P_1, P_2, P_n	Processes
e, f, g	Events
$\rightarrow$	The happened before relation on events
$(P, \rightarrow)$	Poset
E	The number of events in the computation
L	Lattice
G, H, K	Consistent global states (or ideals)
$G[i]$	Maximum event on P_i in G
m_i	The number of events in P_i
B	A global predicate
M	The number of consistent global states of the computation
$\subseteq$	The containment order on global states
$\leq_l$	The lexical order on global states

Figure 2: Notation

3 An Algorithm for Enumeration of Ideals in Lexical order

It is useful to impose on the set of global states the *lexical* or the dictionary order. We define the lexical order ($<_l$) as follows.

$$G <_l H \equiv \exists k : (\forall i : 1 \leq i \leq k-1 : G[i] = H[i]) \wedge (G[k] < H[k])$$

This imposes a total order on all global states by assigning higher priority to small numbered processes. For example, in Figure 1 $01 <_l 10$ because P_1 has executed more events in the global state 10 than in 01.

We use $\leq_l$ for the reflexive closure of the $<_l$ relation. Recall that we have earlier used the order $\subseteq$ on the set of global states which is a partial order. The $\subseteq$ order shown in Figure 1(b) is equivalent to

$$G \subseteq H \equiv \forall i : G[i] \leq H[i])$$

Note that $01 \not\subseteq 10$ although $01 \leq_l 10$.

The relationship between the two orders defined is given by the following lemma.

Lemma 1 *$\forall G, H : G \subseteq H \Rightarrow G \leq_l H$.*

Proof: $G \subseteq H$ implies that $\forall i : G[i] \leq H[i]$. The lemma follows from the definition of the lexical order.

■

Since there are two orders defined on the set of global states, to avoid confusion we use the term *least* for infimum over $\subseteq$ order, and the term *lexically minimum* for infimum over the $\leq_l$ order.

For ease in understanding our approach, we describe two algorithms. The first algorithm has the same space complexity but higher time complexity than the second algorithm. It is based on first enumerating all possible global states in the lexical order and then checking for each one of them whether it is consistent. The second algorithm shows how to avoid generating inconsistent global states.

The first algorithm is a generalization of the "add one" algorithm to generate all binary strings of size n. To generate all binary strings of size n, we start with the number 0 (the string will all 0's) and keep adding 1 till we reach the string with all 1's. Generating all binary strings of size n can be viewed as generating ideals of the poset in which each process has exactly one event. We generalize "add one" algorithm in two ways.

(1) Each process P_i has m_i events where m_i is greater than or equal to 1. Therefore, we view the ideal as a string in the mixed-radix notation rather than binary string. For example, in a system with three processes such that P_1 has 2 events, P_2 has 5 events, and P_3 has 4 events, the tuple $(1, 3, 4)$ represents the ideal in which P_1 has executed 1 event, P_2 has executed 3 events and P_3 has executed 4 events. The next ideal in the lexical order would be $(1, 4, 0)$.

(2) In case of binary strings, we simply added 1 to obtain the next binary string. This is because all binary strings were consistent. In general, a simple addition of 1 may not result in a CGS. For example, in Figure 1, we get $(0, 2)$ on adding one to $(0, 1)$ but $(0, 2)$ is not a CGS. Therefore, instead of adding one, we need to add the least number that takes us to the next CGS.

Based on the above considerations, we get the algorithm in Figure 3. This algorithm is essentially equivalent to Algorithm M in [Knu02] for the purpose of generating all n-tuples except for the consistency check. The array G denotes the vector of the current ideal and the array m is used for the final ideal. To simplify programming, we use $G[0]$ and $m[0]$ as sentinel values with $m[0]$ equal to 1. This ensures that the nested *while* loop always terminates in a valid state.

```
Input: a distributed computation, a predicate B
Output: the lexically minimum CGS that satisfies B
        null if none exists;
var
 G:array[0 ... n] of integer initially ∀i : G[i] = 0;
 m:array[0 ... n] of integer initially m[i] = the number of events at P_i;

 if (B(G)) then return G;// visit the initial CGS

 while (true) // generate the next global state
     i := n;
     while (G[i] = m[i]) // carry over
         G[i] := 0;
         i := i − 1;
     endwhile;
     if (i = 0) then return null;
     else G[i] := G[i] + 1;

     if (∀j, k : dep(G[j], k) ≤ G[k]) then // if G is consistent
         if (B(G)) then return G; // visit this global state
 endwhile;

 return null;
```

Figure 3: An Algorithm for traversal in lexical order

The consistency check is performed by using *dep* function which encodes the poset as follows. $dep(G[j], k)$ returns the largest index of the event on P_k on which the event $G[j]$ depends. If vector clocks are available, then $dep(G[j], k)$ is simply $G[j].v[k]$ where v is the vector clock at $G[j]$.

The space complexity is $O(nE)$ required for storing the vector clock for each event in the poset for consistency check. The time required is for going through the product space and checking consistency of each of the global state generated. The number of tuples in the product space is $O(k^n)$ assuming that k is the maximum number of events on any one process. Generating each tuple requires $O(n^2)$ amortized time giving us the time complexity of $O(k^n n^2)$.

The above algorithm has the disadvantage that it generates all possible global states and then checks for consistency for each one of them. We now modify the lexical algorithm to avoid generating all global states. We assume that vector clocks are available with each event.

Let $nextLex(G)$ denote the CGS that is the successor of G in the lexical order. For example, in Figure 1, $nextLex(01) = 10$ and $nextLex(13) = 20$. It is sufficient to implement $nextLex$ function efficiently for enumeration of ideals in the lexical order. One can set G to the initial CGS $\langle 0, 0, ..., 0\rangle$ and then call the function $nextLex(G)$ repeatedly. We implement the function $nextLex(G)$ using two secondary functions $succ$ and $leastConsistent$ as described next.

We define $succ(G, k)$ to be the global state obtained by advancing along P_k and resetting components for all processes greater than P_k to 0. Thus, $succ(\langle 7, 5, 8, 4\rangle, 2)$ is $\langle 7, 6, 0, 0\rangle$ and $succ\langle 7, 5, 8, 4\rangle, 3)$ is $\langle 7, 5, 9, 0\rangle$. Note that $succ(G, k)$ may not exist when there is no event along P_k, and even when it exists it may not be consistent.

The second function is $leastConsistent(G)$ which returns the least consistent global state greater than or equal to G in the $\subseteq$ order. This is well defined as shown by the following lemma.

Lemma 2 *The set of all consistent global states that are greater than or equal to G in the CGS lattice is a sublattice. Therefore, there exists the least CGS H that is greater than or equal to G.*

Proof: It is easy to verify that if two consistent global states K and L are both greater than or equal to G, then so is their union and intersection.

■

We now show how $nextLex(G)$ can be computed.

Theorem 1 *Assume that G is a CGS such that it is not the greatest CGS. Then,*

$$nextLex(G) = leastConsistent(succ(G, k))$$

where k is the index of the process with the smallest priority which has an event enabled in G.

Proof: Let e be the event in the smallest priority process enabled in G. First observe that $leastConsistent(succ(G, k)$ does not include any additional events from any process with higher priority than k. This is because $G \cup \{e\}$ is a CGS and $succ(G, k) \subseteq G \cup \{e\}$ and definition of $leastConsistent()$

$$leastConsistent(succ(G, k)) \subseteq G \cup \{e\}$$

Pick any CGS H that is lexically greater than G. It must include at least one event that is not in G; otherwise it would be lexically smaller. Choose the additional event e from the highest priority process. The event e cannot be from a process with higher priority than k as shown before. It cannot be from a process with lower priority than k because no event in a process with a lower priority than k was enabled at G. Therefore, e is from P_k. But then we have that $succ(G, k) \subseteq H$. We also know that H is consistent and $leastConsistent(succ(G, k))$ is the least CGS with this property. Therefore, $nextLex(G) \subseteq H$ which implies that $nextLex(G) \leq_l H$ from Lemma 1.

■

The only task left in the design of the algorithm is to determine k and implement $leastConsistent$ function. The following lemma follows from properties of vector clocks[Fid89, Mat89]. First, we determine that an event e in enabled in a CGS G if all the components of the vector clock for other processes in e are less than or equal to the components of the vector clock in G. Secondly, to compute $leastConsistent(G)$ it is sufficient to take the component-wise maximum of the vector clock of all maximal events in G. Formally,

Lemma 3

(1) An event e on P_k is enabled in a CGS G iff $e.v[k] = G[k] + 1$ and

$$\forall j : j \neq k : e.v[j] \leq G[j]$$

(2) Let $K = leastConsistent(G)$. Then,

$$\forall j : K[j] = max\{G[i].v[j] \mid 1 \leq i \leq n\}$$

Incorporating these observations, we get the algorithm in Figure 4. The outer *while* loop at line (1) iterates till all consistent global states are visited. If the current CGS G satisfies the given predicate B, then we are done and G is returned as the lexically minimum CGS. Lines (4)-(22) generate $nextLex(G)$. Lines (4)-(15) determine the lowest priority process k which has an event enabled in G. The *while* loop on line (6) terminates only when k is found. We are guaranteed to get an event enabled because G is not the final CGS. Lines (8)-(12) check if the next event on P_k is enabled. Lines (16) to (18) compute $succ(G, k)$. Finally, lines (19)-(22) compute $leastConsistent(succ(G, k))$.

Let us now analyze the time and space complexity of the above algorithm. The *while* loop iterates once per CGS of the computation. Each iteration takes $O(n^2)$ time due to nested *for*. Thus the total time taken is $O(n^2 M)$. The dominant space requirement is for storing the vector clock for all events which is $O(nE)$.

4 Algorithms for BFS generation of Ideals

For many applications we may be interested in generating consistent global states in the BFS order, for example, when we want to generate elements in a single level of the CGS lattice. The lexical algorithm is not useful for that purpose.

Cooper and Marzullo [CM91] have given an algorithm to detect $possibly : B$ based on the level set enumeration. They keep two lists of consistent global states: *last* and *current*. To generate the next level of consistent global states, they set *last* to *current* and *current* to the set of global states that can be reached from *last* in one transition. Since a CGS can be reached from multiple global states, an implementation of their algorithm will result in either *current* holding multiple copies of a CGS or added complexity in the algorithm to ensure that a CGS is inserted in *current* only when it is not present. This problem occurs because their algorithm does not exploit the fact that the set of global states form a distributive lattice.

```
lexTraverse(P, B)

Input: a distributed computation P, a predicate B
Output: the smallest CGS that satisfies B, null if none exists;
var
    G:array[0 ... n] of integer initially ∀i : G[i] = 0; // current CGS
    K:array[0 ... n] of integer; // K = leastConsistent(succ(G, k))
    m:array[0 ... n] of integer; // m[i] equals the number of events at P_i

while (G ≤ m) do
    if (B(G)) then return G;
    if (G = m) then return null;

    boolean found := false;
    int k := n;
    while ¬found do
        if (G[k] ≠ m[k]) then // next event on P_k exists
            e := next event on P_k after G[k]
            boolean enabled := true;
            for j := 1 to n, j ≠ k do // check if it is enabled
                if e.v[j] > G[j] then
                    enabled := false;
            found := enabled; // next event on P_k enabled
        else k := k − 1; // try the smaller process;
    endwhile;

    G[k] := G[k] + 1; // advance on P_i
    for j := k + 1 to n do // reset higher numbered processes to 0
        G[j] := 0;

    K := G; // initialize K to G
    for i := 1 to n do // compute K := leastConsistent(G);
        for j := 1 to n do
            K[j] := max(K[j], G[i].v[j]);

endwhile;
```

Figure 4: An Algorithm for Traversal in Lexical Order

We now show an algorithm which will ensure that a CGS is enumerated exactly once and that there is no overhead of checking that the CGS has already been enumerated (or inserted in the list). Our algorithm exploits the following observation.

Lemma 4 *If H is reachable from G by executing an event e and there exists an event $f \in G$ such that f is maximal in G and concurrent with e, then there exists a CGS G' at the same level as G such that $G' = G - \{f\} + \{e\}$ and H is reachable from G'.*

Proof: Since G is consistent and f is a maximal event in G, it follows that $G - \{f\}$ is a CGS. If e is enabled at G and f is concurrent with e, then e is also enabled at $G - \{f\}$. Therefore, $G' = G - \{f\} + \{e\}$ is a CGS. H is reachable from G' on executing f.

■

In the above scenario, our algorithm will avoid enumerating H from both G and G'. To achieve this it is sufficient to have a total order on all events and explore execution of an event e from a global state G iff e is smaller than all maximal events in G which are concurrent with e. Let σ be a topological sort of all events which maps every event e to $\sigma(e)$ a number from $1..E$. Now the rule to decide which events to explore from a CGS G is simple. Let e be any enabled event in G. We explore execution of e on G iff

$$\forall f \in maximal(G) : f || e \Rightarrow \sigma(e) < \sigma(f)$$

With this observation, our algorithm shown in Figure 5 keeps a queue Q of the CGS. At every iteration, it removes the head of the queue G, checks if the global predicate is true on G. If it is, we are done. Otherwise, the algorithm inserts those successors of G that satisfy the above rule.

```
var
  Q:set of consistent global states at the current level initially {(0,0,...,0)};
  σ:a topological sort of the poset P;

  while (Q ≠) do
      G := remove_first(Q);
      if B(G) then return G;
      for all events e enabled in G do // generate CGS at the next level
          if (∀f ∈ maximal(G) : f||e ⇒ σ(e) < σ(f))then
              H := G ∪ {e};
              append(Q, H);
      endfor;
  endwhile;
```

Figure 5: Time Efficient Algorithm for BFS enumeration of CGS

We now analyze the time complexity of above algorithm. There are $O(M)$ iterations of the *while* loop. In each iteration, there may be at most n events enabled per CGS. Determining whether an event is enabled takes at most $O(n)$ time. Determining whether it is smaller than all the maximal events in G that are concurrent to it also takes $O(n)$ time. Thus the overall time complexity is $O(n^2 M)$.

The main disadvantage of this algorithm is that it requires space at least as large as the number of consistent global states in the largest level set. Note that the largest level set is exponential in n and therefore when using this algorithm for a large system we may run out of memory.

We now give two algorithms that use polynomial space to list the global states in the BFS order. The first algorithm is based in integer compositions and consistency checks and the second algorithm is based on using the DFS (or the lexical) traversal multiple number of times to enumerate consistent global states in the BFS order.

The first algorithm for BFS traversal uses consistency check to avoid storing the consistent global states. The main idea is to generate all the global states in a level rather than storing them. Assume that we are interested in enumerating level set r. Any global state in level set r corresponds to the total number of events executed by n processes to be r. A *composition* of r into n parts corresponds to a representation of the form $a_1 + a_2 + \ldots + a_n = r$ where each a_i is a natural number and the order of the summands is important. In our application, this corresponds to a global state $(a_1, a_2, ..., a_n)$ such that $\sum a_i = r$. There are many algorithms that enumerate all the compositions of an integer r into n parts (for example, the algorithm due to Nijenhuis and Wilf[NW78] (pp. 40-46) runs through the compositions in lexicographic order reading from right to left).

We now arrive at the algorithm in Figure 6

```
var
 G:array[1 ... n] of integer initially G[1] = levelnum ∧ ∀i ≥ 2 : G[i] = 0;

 while (G ≠ null) do
     if isConsistent(G) then
         if B(G) then return G;
     // generate the next global state
     G := nextComposition(G) //use Neijenhuis and Wilf's algorithm;

 endwhile;
```

Figure 6: A Space Efficient algorithm to generate the level set of rank *levelnum*

The above algorithm has polynomial space complexity and can therefore generate level sets even for reasonably large systems.

The second algorithm exploits the fact that the DFS and the lexical traversal can be done in polynomial space. We perform the DFS traversal for each level number l. During the DFS traversal we explore a global state only if its level number is less than or equal to l and visit it (evaluate the predicate or print it depending upon the application) only if its level number is exactly equal to l. The algorithm shown in Figure 7 generates one level at a time. In line (2) it reduces the computation to include only those events whose sum of vector clock values is less than or equal to *levelnum*. All other events can never be in a global state at level less than or equal to *levelnum*. In line (3) it performs space efficient lexical traversal of the CGS lattice using the algorithm in Figure 4. The computation used is the reduced one and the global predicate that is evaluated is modified to include a clause that the CGS should be at level equal to *levelnum*. If no CGS is found, then we try the next level. Since the total number of levels is $O(E)$, we can enumerate the consistent global states in the BFS order in $O(En^2M)$ time and $O(nE)$ space. Note that our algorithm enumerates each level itself in the lexical order.

var
    G:array[$1 \ldots n$] of integer;

(1) **for** $levelnum := 0$ to E **do**
(2) $Q := \{e \in P \mid \sum_i e.v[i] \leq levelnum\}$
(3) $G := lexTraverse(Q, B \wedge (lvl = levelnum))$; // Figure 4
(4)   **if** $(G \neq null)$ **then**
(5)   **return** G;
(6) **endfor**;

(7) **return** $null$;

Figure 7: A Space Efficient algorithm for BFS Enumeration

5 Application to Combinatorial Enumeration

We first give the most straightforward application to enumeration of combinatorial objects. We consider the distributed computation shown in Figure 8(a). Each process P_i executes a single event e_i. It is easy to verify that there is a bijection between every CGS of the computation and a subset of X. For example, when $n = 4$, the CGS $(0, 1, 0, 1)$ corresponds to the set $\{e_2, e_4\}$. The lexical enumeration would correspond to enumeration of sets based on binary string representation. Figure 8(b) gives another computation which can also be used to enumerate all the subsets of $\{1..4\}$. In this example, the enumeration would result in the lexical order (reversed). Each global state (or a cut) corresponds to a subset. For example, G corresponds to the subset $\{4, 2, 1\}$. We say that P_i has chosen element k if the cut (global state) goes through the label k for process P_i. In this computation P_1 chooses the biggest element in the set. By the design of the computation, P_2 can only choose an element that is strictly smaller than the element chosen by P_1 for the global state to be consistent. For example, the global state H is not consistent because P_2 has also chosen 1 which is same as the element chosen by P_1. The choice of 0 by any P_i denotes that the process does not choose any element. It can be verified that if P_i chooses 0, then all higher numbered processes can only choose 0.

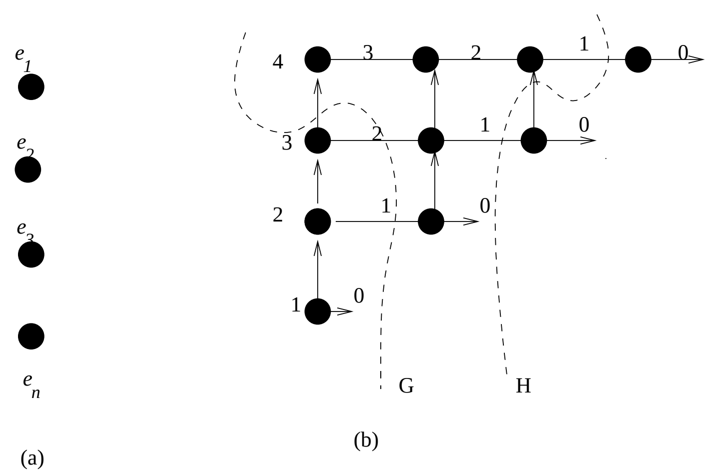

Figure 8: Computation for generating subsets of X

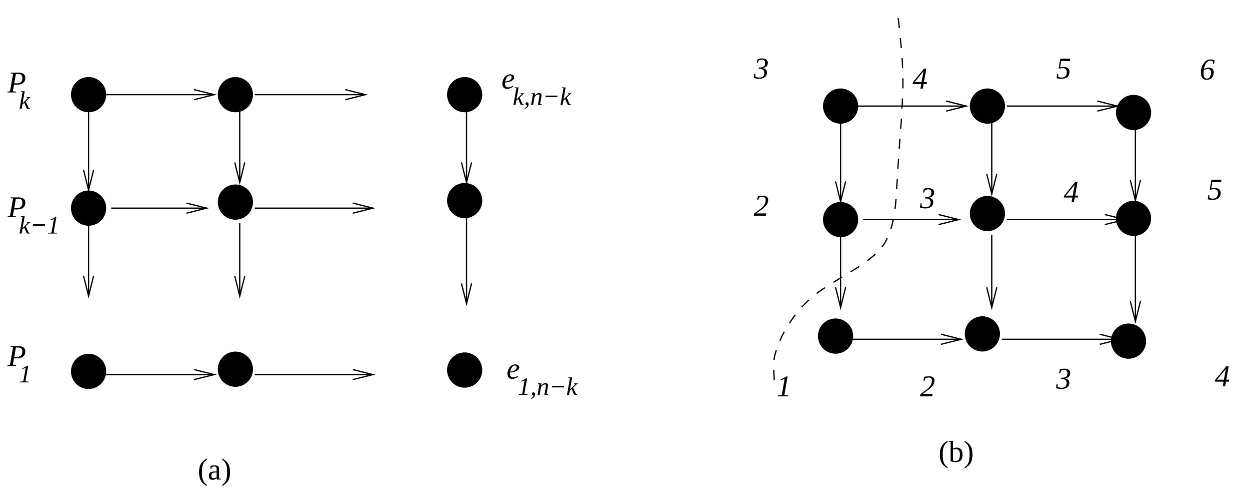

Figure 9: Computation for generating subsets of X of size k

Figure 9 shows a computation such that all the subsets of X of size k are its consistent global states. There are k processes in this computation and each process executes $n-k$ events. By the structure of the computation, if in a CGS P_i has executed j events, then P_{i+1} must have also executed at least j events. The correspondence between subsets of X and global states can be understood as follows. If process P_i has executed t events in the CGS, then the element $t+i$ is in the set Y. Thus process P_1 chooses a number from $1 \ldots n-k+1$ (because there are $n-k$ events); process P_2 chooses the next larger number and so on. Figure 9(b) gives an example of the computation for subsets of size 3 of the set [6]. The CGS, or the ideal, shown corresponds to the subset $\{1, 3, 4\}$.

A k-tuple of positive integers $\lambda = (\lambda_1, \ldots, \lambda_k)$ is an integer partition of N if $\lambda_1 + \ldots + \lambda_k = N$ and for all i, $\lambda_i \geq \lambda_{i+1}$. The number of *parts* of λ is k. An example of partition of 10 into 3 parts is $(4, 3, 3)$. An integer partition can be visualized as a *Ferrers diagram* or an array of squares in decreasing order with λ_i squares in row i. The Ferrers diagram of the partition $(4, 3, 3)$ of 10 is shown in Figure 10(a). A partition λ is contained in another partition δ if the number of parts of λ is at most that of δ and λ_i is less than or equal to δ_i for any i between 1 and the number of parts of λ. For example, $(3, 3, 1)$ is less than $(4, 3, 3)$. Fix any partition λ. The set of all partitions that are less than or equal to λ form the *Young's lattice* denoted by Y_λ. Figure 10(b) shows the computation for generating Young's lattice. By enumerating all consistent global states of the computation in lexical order we can enumerate all partitions that are less than or equal to λ. Now consider the Young Lattice for $\lambda = (N, N, \ldots N\ times)$. The N^{th} level set of this lattice corresponds to all integer partitions of N. By enumerating the N^{th} level set we can enumerate all integer partitions of N.

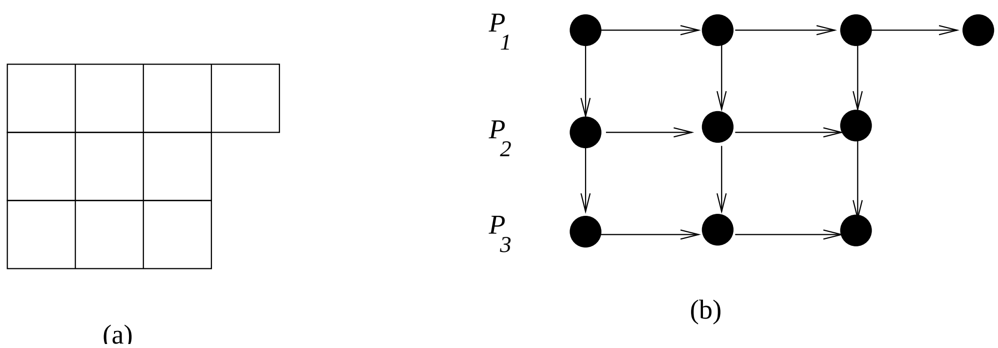

Figure 10: (a) A Ferrer diagram (b) A computation for generating Young's lattice

Finally, we show that our algorithms can also be used to enumerate permutations. We first show a small computation that generates all permutations of n symbols. The computation consists of $n-1$ processes. Process P_i executes $i-1$ events. We use the inversion table[Knu98] to interpret a CGS. The number of inversions of i in a permutation π is the number of symbols less than i that appear to the right of i in π. The way a permutation is generated from a CGS is as follows. We begin the permutation by writing 1. P_1 decides where to insert the symbol 2. There are two choices. These choices correspond to number of *inversions* introduced by 2. If we place 2 after 1, then we introduce zero inversions; otherwise we introduce one inversion. Proceeding in this manner we get that there is a bijection between the set of permutations and the consistent global states. The resulting algorithm is similar to Johnson-Trotter method of enumerating all permutations or showing bijection between inversion table and permutations[SW86].

In many applications we are interested in subsets of combinatorial families. For example, we may be interested in enumerating all k subsets of $[n]$ which do not have any consecutive numbers. By using methods of slicing one can compute posets such that its ideals correspond to precisely these subsets. Now we can use the algorithm for lexical order traversal to enumerate all such subsets. The interested reader is referred to [MG01b] and [Gar02].

6 Detecting $definitely : B$ in polynomial space

A predicate B is definitely true in a computation if all paths from the initial CGS to the final CGS go through some global state that satisfies B. Cooper and Marzullo's algorithm [CM91] detects $definitely : B$ as follows. It starts from the initial CGS. If B is true in the initial CGS we are done. Otherwise, it constructs the next level set including only those global states in which $\neg B$ is true. Continuing in this manner, if the algorithm can reach the final CGS, then $definitely : B$ is false; otherwise, it is true. This algorithm requires space proportional to the size of the largest level set.

We now present a simple and space-efficient algorithm for $definitely : B$ (shown in Figure 11). The algorithm is based on determining a path from the initial CGS to the final CGS in which $\neg B$ holds along all states. The function $nonBpath(G)$ returns true iff there exists a path from G to the final CGS in which $\neg B$ holds along all CGS. To compute $definitely : B$ it is sufficient to check that $\neg B(G)$ holds and then call $nonBpath()$.

This function is recursive and uses ideas in Alagar and Venkatesan's algorithm for DFS traversal for detecting $possibly : B$. In line (1), the function returns false, because if B holds in the starting state then there cannot be a path along which $\neg B$ holds. In line (2), we take care of the base case. If the starting state is same as the final state, then we have found a path along which $\neg B$ holds and we return true. Lines (3)-(8) extend the path from G by one more transition. We assume that processes are numbered from 1 to n and the list of events enabled in G are sorted according to process id's. By choosing the event in the process with the smallest processor id first, we ensure that within a level consistent global states which are lexically bigger are visited first.

Line (5) ensures that $\neg B$ holds in the next state. Lines (6)-(7) ensure that H has not been visited before. To check this we first determine all consistent global states that can lead to H in the variable $prevH$. Of all consistent global states in $prevH$ that satisfy $\neg B$, we choose K to be the one that is lexically the biggest. If $K \neq G$, then H has been visited before from the state K and is not visited again. Otherwise, we visit the state H in line (9). If there is no $nonBpath$ from any of the successor of G, then we return false in line (10).

The space requirement of the algorithm is due to recursive calls of $nonBpath()$. The depth of the recursion is at most E, the total number of events in the system. The global state can be represented using a n-dimensional vector. Therefore, at each recursion level $O(n)$ space is required giving us the space complexity of $O(nE)$. The time complexity of the algorithm is $O(n^3 M)$.

7 Discussion

The paper makes the following contributions.

- We give an algorithm that does lexical traversal of the CGS lattice in $O(nE)$ space and $O(n^2 M)$ time.

```
boolean nonBpath(G : CGS)

if B(G) return false;
if finalCGS(G) return true;

for e ∈ enabled(G) do
      H := G ∪ {e};
      if ¬B(H) then
            prevH := {H − {f} | f ∈ maximal(H)};
            K := max{W ∈ prevH | ¬B(W)}; // max in lexical order
            if (K = G) then
                  if nonBpath(H) then return true;
endfor;

return false;
```

Figure 11: An algorithm to detect $definitely : B$ with polynomial space complexity

- We show that at the expense of more time, BFS traversal can be accomplished in polynomial space. The previous algorithm for BFS traversal uses exponential space.
- We show that our algorithm for generation of ideals in the lexical order and the BFS order can be viewed as a generic algorithm for enumerating many families of combinatorial objects.
- Finally, we discuss detection of global predicate under definitely modality. Cooper and Marzullo's algorithm requires exponential space. We give a simple algorithm that requires only polynomial space.

We note here that there are other approaches in the mathematics and operations research literature for enumeration of ideals of a poset. Steiner[Ste86] gave an algorithm to list the ideals in $O(EM)$ time. Bordat[Bor91] proposed an $O(nM)$ time algorithm with $O(E^2)$ space. Squire[Squ95] has proposed an $O(log(E)M)$ algorithm. Jegou, Medina, and Nourine [JMN95] have proposed an $O(cM)$ algorithm with space complexity of $O(M)$ where c is the maximum in-degree in the covering graph of the poset. Habib, Medina, Nourine and Steiner[HMNS01] have proposed an algorithm with $O(cM)$ time and $O(nE)$ space. Their algorithm is the most efficient known for generating all ideals in $O(nE)$ space and is based on defining a spanning tree in the covering graph of the ideal lattice and enumerating vertices of the spanning tree. None of these algorithms enumerate consistent global states (or ideals) in the lexical or the BFS order.

The most interesting question left open is: Is there any traversal algorithm for the CGS lattice in $O(M)$ time and polynomial space?

Acknowledgments

I am thankful to James Roller Jr. and Alper Sen for some discussions on the topic.

References

[AV01] S. Alagar and S. Venkatesan. Techniques to tackle state explosion in global predicate detection. *IEEE Transactions on Software Engineering*, 27(8):704 – 714, August 2001.

[Bor91] J.P. Bordat. Calcul des ideaux d'un ordonne fini. *Operation Research*, 25(4):265 – 275, 1991.

[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.

[DP90] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[Fid89] C. J. Fidge. Partial orders for parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN Notices*, 24(1):183–194, January 1989.

[Gar02] V. K. Garg. Algorithmic combinatorics based on slicing posets. In *Proc. of 12th Conference on the Foundations of Software Technology & Theoretical Computer Science*, pages to appear, available at `http://www.ece.utexas.edu/~garg`. Springer Verlag, December 2002. Lecture Notes in Computer Science.

[GW91] V. K. Garg and B. Waldecker. Detection of unstable predicates. In *Proc. of the Workshop on Parallel and Distributed Debugging*, Santa Cruz, CA, May 1991. ACM/ONR.

[HMNS01] M. Habib, R. Medina, L. Nourine, and G. Steiner. Efficient algorithms on distributive lattices. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 110:169 – 187, 2001.

[JMN95] Roland Jégou, Raoul Medina, and Lhouari Nourine. Linear space algorithm for on-line detection of global predicates. In Jörg Desel, editor, *Proceedings of the International Workshop on Structures in Concurrency Theory (STRICT)*, Workshops in Computing, pages 175–189. Springer-Verlag, 1995.

[Knu98] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, second edition, 1998.

[Knu02] D.E. Knuth. *The Art of Computer Programming, Volume 4*. Addison-Wesley, Reading, MA, 2002. Pre-Fascicle available from `http://www-cs-faculty.stanford.edu/~knuth`.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Mat89] 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.

[MG01a] N. Mittal and V. K. Garg. On detecting global predicates in distributed computations. In *21st International Conference on Distributed Computing Systems (ICDCS' 01)*, pages 3–10, Washington - Brussels - Tokyo, April 2001. IEEE.

[MG01b] N. Mittal and V. K. Garg. Slicing a distributed computation: Techniques and theory. In *5th International Symposium on DIStributed Computing (DISC'01)*, pages 78 – 92, October 2001.

[NW78] A. Nijenhuis and H. S. Wilf. *Combinatorial Algorithms for Computers and Calculators.* Academic Press, London, 2 edition, 1978.

[Squ95] M. Squire. *Gray Codes and Efficient Generation of Combinatorial Structures.* PhD Dissertation, Department of Computer Science, North Carolina State University, 1995.

[Sta86] R. Stanley. *Enumerative Combinatorics Volume 1.* Wadsworth and Brookes/Cole, Monterey, California, 1986.

[Ste86] G. Steiner. An algorithm to generate the ideals of a partial order. *Operations Research Letters*, 5(6):317 – 320, 1986.

[SW86] D. Stanton and D. White. *Constructive Combinatorics.* Springer-Verlag, 1986.

[TG99] A. Tarafdar and V. K. Garg. Software fault-tolerance of concurrent programs using controlled reexecution. In *Proceedings of the 13th International Symposium on DIStributed Computing (DISC)*, pages 210 – 224, Bratislava, Slovakia, September 1999.