Enumerating Global States of a Distributed Computation

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
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. We give an efficient algorithm that perform the *lex* traversal of the lattice. We also give a space efficient algorithm for the breadth-first-search (BFS) traversal.

KEY WORDS
Global Predicate Detection, Combinatorial Enumeration, Lattices, Ideals

1 Introduction

Global predicate detection is a fundamental problem in distributed debugging [1, 2]. 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.

Given a distributed computation, the global predicate detection problem asks whether there exists a consistent global state (CGS) [3] 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 [4].

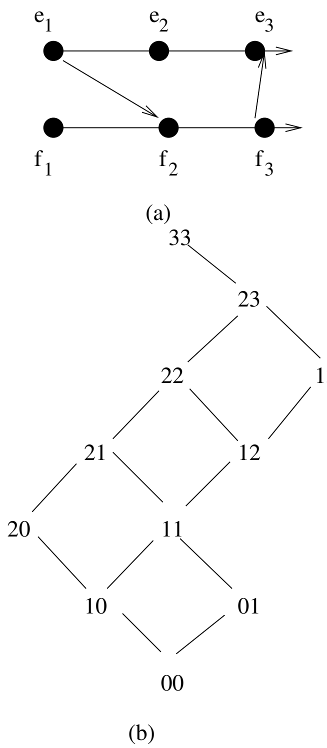
The CGS lattice can be traversed in multiple ways as shown in Figure 1(c). Cooper and Marzullo's algorithm[2] performs a breadth-first-search (BFS) traversal and requires space proportional to the size of the biggest level of the CGS lattice which, in general, is *exponential* in the size of the computation. Alagar and Venkatesan's algorithm[5] 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. The main disadvantage of their algorithm is that it requires recursive calls of depth $O(E)$ with each call requiring $O(n)$ space resulting in $O(nE)$ space requirements besides storing the computation itself.

In this paper, we propose a new algorithm that performs the *lexicographic (lex)* traversal of the lattice with $O(n)$ space (besides the input) and $O(n^2M)$ time complexity. Lex 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 lex 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 lexicographically 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

*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. Part of the work was performed when the author was visiting Computer Science and Engineering Department at Indian Institute of Technology Kanpur as N. Rama Rao Visiting Chair Professor.

size of a level of a lattice. It is based on efficient enumeration of a level set by enumerating all *integer compositions*.



BFS: 00, 01, 10, 11, 20, 12, 21, 13, 22, 23, 33

DFS: 00, 10, 20, 21, 22, 23, 33, 11, 12, 13, 01

Lexical: 00, 01, 10, 11, 12, 13, 20, 21, 22, 23, 33

(c)

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 CGS in the lex and the BFS order is also useful in combinatorial applications. In [6] 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. Thus, algorithms for lex 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 [7] gives different algorithms for these enumerations. Our algorithm is generic and by instantiating it with different posets all the above combinatorial lex enumeration can be achieved.

2 Model and Background

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 $\rightarrow$ [8]. 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 $(P, \rightarrow)$. Two events e and f are concurrent in $(P, \rightarrow)$, (denoted by $e||f$), if $e \not\rightarrow f$ and $f \not\rightarrow e$.

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 *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 CGS form a distributive lattice under $\subseteq$ relation.

3 An Algorithm for Enumeration of Ideals in Lex order

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

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

This imposes a total order on all global states by assigning higher priority to small numbered processes. For example, in Figure 1, global state $(01) <_l (10)$ 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 $01 \leq_l 10$.

Note that we have two orders on the set of global states—the partial order based on containment ($\subseteq$), and the total order based on lex ordering ($\leq_l$). 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 lex 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 *lexicographically minimum* for infimum over the $\leq_l$ order.

Let $nextLex(G)$ denote the CGS that is the successor of G in the lex 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 lex 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 $leastConsistent(K)$ which returns the least consistent global state greater than or equal to a given global state K 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 K in the CGS lattice is a sublattice. Therefore, there exists the least CGS H that is greater than or equal to K.*

Proof: It is easy to verify that if two consistent global states H_1 and H_2 are both greater than or equal to K, 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: We define the following global states for convenience:

$K := succ(G, k)$
$H := leastConsistent(K)$, and
$G' := nextLex(G)$.
Our goal is to prove that $G' = H$.

G' contains at least one event f that is not in G; otherwise $G' \subseteq G$ and therefore cannot be lexically bigger. Choose $f \in G' - G$ from the highest priority process possible.

Let e be the event in the smallest priority process enabled in G, i.e., e is on process P_k. Let $proc(e)$ and $proc(f)$ denote the process indices of e and f. We now do a case analysis.

Case 1: $proc(f) < k$
In this case, e is from a lower priority process than f. We show that this case implies that H is lexically smaller than G'. We first claim that $H \subseteq G \cup \{e\}$. This is because $G \cup \{e\}$ is a CGS containing K and H is the smallest CGS containing K. Now, since $H \subseteq G \cup \{e\}$ and G' contains an event $f \in G' - G$ from a higher priority process than e, it follows that H is lexically smaller than G', a contradiction.

Case 2: $proc(f) > k$
Recall that event e is on the process with the smallest priority that had any event enabled at G. Therefore, existence of f in CGS G' implies existence of at least another event in $G' - G$ at a process with higher priority than e. This contradicts choice of event f because, by definition, f is from the highest prioriy process in $G' - G$.

Case 3: $proc(f) = k$.
Then, $K \subseteq G'$ because both G' and K have identical events on process with priority k or higher and K has no events in lower priority processes. Since G' is a CGS and $K \subseteq G'$, we get that $H \subseteq G'$ by definition of H. From Lemma 1, it follows that $H \leq_l G'$. But G' is the next lecical state after G. Therefore, $H = G'$.

■

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[9, 10]. 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 $leastConsistent(K)$ it is sufficient to take the componentwise maximum of the vector clock of all maximal events in K. 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* $H = leastConsistent(K)$. *Then,*

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

Incorporating these observations, we get the algorithm in Figure 2. 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 lexicographically minimum CGS. Lines (4)-(22) generate $nextLex(G)$. Lines (4)-(14) determine the lowest priority process k which has an event enabled in G. The *for* loop on line (4) is exited when an enabled event is found at line (12). We are guaranteed to get an enabled event because G is not the final CGS. Lines (8)-(12) check if the next event on P_k is enabled. This is done using the vector clock. An event e in enabled in a CGS G iff all the events that e depend on have been executed in G; or equivalently, 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. This test is performed in lines (8)-(11). Lines (15) to (17) compute $succ(G, k)$. Finally, lines (18)-(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 algorithm uses variables G, K, H and m which requires $O(n)$ space. We also assume that the events are represented using their vector clocks.

```
lexTraverse(P, B)

Input: a distributed computation P, a predicate B
Output: the smallest CGS in lex order that satisfies B,
        null if none exists;
var
// current CGS
     G:array[1...n] of int initially ∀i : G[i] = 0;
// K = succ(G, k)
     K:array[1...n] of int;
// H = leastConsistent(K)
     H:array[1...n] of int;
// m[i] equals the number of events at P_i
     m:array[1...n] of int;

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

(4)      for k := n down to 1 do
             // if next event on P_k exists
(5)          if (G[k] ≠ m[k]) then
(6)              e := next event on P_k after G[k]
(7)              boolean enabled := true;
(8)              for j := 1 to n, j ≠ k do
(9)                  if e.v[j] > G[j] then
(10)                     enabled := false;
(11)             end// for;
(12)             if (enabled) break; //goto line (15);
(13)         end// if next event exists;
(14)     end// for;

         // compute K := succ(G, k)
(15)     K := G; //
(16)     K[k] := K[k] + 1; // advance on P_i
(17)     for j := k + 1 to n do K[j] := 0;

         // compute H := leastConsistent(K);
(18)     H := K; // initialize H to K
(19)     for i := 1 to n do
(20)         for j := 1 to n do
(21)             H[j] := max(H[j], K[i].v[j]);

(22)     G := H;
(23) end// while;
```

Figure 2. An Algorithm for Traversal in Lex Order

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 lex algorithm is not useful for that purpose.

Cooper and Marzullo [2] 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.

We now show an extension of their algorithm which ensures 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 extension 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.

■

Thus, to avoid enumerating H from both G and G', 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.

$$\{f \mid f \in G, f||e\} \cup \{e\}$$

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

Figure 3. An Extension of Cooper Marzullo Algorithm for BFS enumeration of CGS

The main disadvantage of Cooper and Marzullo's algorithm even with proposed extension 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 lex) 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[11] (pp. 40-46) runs through the compositions in lexicographic order reading from right to left). For every composition, the corresponding global state can be checked for consistency.

The second algorithm exploits the fact that the DFS and the lex 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 4 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 lex traversal of the CGS lattice using the algorithm in Figure 2. 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 lex order.

5 Conclusions

We have presented an algorithm that does lex traversal of the CGS lattice in $O(n)$ additional space and $O(n^2M)$ time. We have also shown that at the expense of more time, BFS traversal can be accomplished in polynomial space. The previous algorithm for BFS traversal uses exponential space.

```
var
    G:array[1...n] of integer;

for levelnum := 0 to E do
    Q := {e ∈ P | Σ_i e.v[i] ≤ levelnum}
    G := lexTraverse(Q, B ∧ (lvl = levelnum));
    if (G ≠ null) then
        return G;
endfor;

return null;
```

Figure 4. A Space Efficient algorithm for BFS Enumeration

We note here that there are other approaches in the mathematics and operations research literature for enumeration of ideals of a poset. See, for example, papers by Steiner[12], Bordat[13], Squire[14], Jegou, Medina, and Nourine [15], and Habib, Medina, Nourine and Steiner[16]. The algorithm in [16] is the most efficient known for generating all ideals in $O(nE)$ space. None of these algorithms enumerate consistent global states (or ideals) in the lex 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., Alper Sen and Stephan Lips for discussions on the topic.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] V. K. Garg. Algorithmic combinatorics based on slicing posets. In *Proc. of 22th Conference on the Foundations of Software Technology & Theoretical Computer Science*, pages 169 – 182. Springer Verlag, December 2002. Lecture Notes in Computer Science.

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

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

[9] 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.

[10] 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.

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

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

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

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

[15] 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.

[16] 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.