Using Order in Distributed Computing

Vijay K. Garg
ECE Department
University of Texas
Austin, TX, USA
garg@ece.utexas.edu

Neeraj Mittal
CS Department
University of Texas, Dallas
Richardson, TX, USA
neerajm@utdallas.edu

Alper Sen
Freescale Semiconductor Inc.
Austin, TX, USA
alper.sen@freescale.com

Abstract

We survey applications of the theory of partial orders to distributed computing. A distributed computation is a poset on the set of events based on the observable happened-before relation. Online chain decomposition of posets has applications in timestamping events with vectors such that the vectors preserve the partial order. The lower bounds on the dimension of these vectors use the dimension theory of posets.

Global states of a distributed computation corresponds to its order ideals. An important problem in distributed computing is to find a global state that satisfies a given predicate. Although, this problem is NP-complete in general, it has efficient solution for classes of predicates called linear and relational. The concept of meet-closure is used for detecting linear predicates and the algorithms for merging chains are used for detecting relational predicates. For general predicates, one can use algorithms for enumeration of all order ideals of the poset.

A slice of a computation captures all global states that satisfy a given predicate. Slicing is based on Birkhoff's duality theorem of distributive lattices. Further, lattice congruences help in reducing the size of the global state graph for verification of temporal logic formulas.

1 Introduction

The theory of order has yielded significant and useful applications in many areas of distributed computing. We will survey applications in four of these areas - tracking dependencies between events in a distributed system, detecting global predicates in distributed computations, slicing distributed computations, and analyzing distributed computations. Before we delve into each of these problems, note that the reader may also consult the companion survey paper [GMS03] that is primarily meant for computer scientists.

The problem of tracking dependency of events in a distributed system concerns with timestamping events in a distributed computation such that the happened-before order between them can be determined. The happened-before order between events in a distributed system was first defined formally by Lamport in 1978 [Lam78]. He argued that the order of events that can be observed in a distributed computation is only partial. He also presented a mechanism called *logical clocks* that gave a timestamp in a totally ordered domain preserving the happened-before order. In 1989, Mattern [Mat89a] and Fidge [Fid91] defined a vector clock mechanism that can be used to timestamp events in a distributed computation such that the partial order on timestamps is isomorphic to the partial order on events. Formally, let E be the set of events of a computation with n processes and $\rightarrow$ be the happened-before order on E. The vector clock mechanism is a mapping v from E to set of n-dimensional vectors such that:

$$e \rightarrow f \equiv v(e) < v(f)$$

where $<$ is the usual component-wise order relation on vectors. Vector clocks have been used extensively in many distributed algorithms [Gar02b]. Much of the research in this area has focused on reducing the dimension of the vector or proving lower bounds on the dimension. This research has used offline and online chain decomposition, dimension theory and other order-theoretic results. These results are surveyed in Section 3.

The problem of detecting global predicates concerns with the techniques for detecting a consistent global state of a distributed system that satisfies a given predicate. Chandy and Lamport [CL85] were the first to define a consistent global state (or a consistent cut). A subset G of E is a consistent cut if whenever it contains an element f then it contains all elements e that happened-before f. This concept is identical to the notion of order ideal in the lattice theory. In that paper, they also gave a distributed algorithm to record a consistent cut. Their algorithm is useful in detecting any stable predicate, a predicate that stays true once it becomes true. The research in this area has focused on developing algorithms for other classes of predicates and establishing intractability for the general class. In particular, the class of linear predicates and relational predicates have been defined. Algorithms in this area have used properties of inf-semilattices and chain merging algorithms. These results are surveyed in Section 4.

Slicing a distributed computation is required to restrict one's attention to a subcomputation. A slice of a computation with respect to a predicate B is a concise representation of all consistent cuts that satisfy B. The concept of computation slice was introduced in 2001 by Garg and Mittal [GM01]. They exploit the fact that the set of consistent cuts form a *distributive* lattice and Birkhoff's Theorem on finite distributive lattices. Slice has benefits in terms of state space reduction for predicate detection and evaluation of temporal logic predicates on the lattice of consistent cuts. These applications were further explored by Mittal and Garg in [MG01a, MG03] and Sen and Garg in [SG03b]. These results are surveyed in Section 5.

Analyzing a distributed computation requires, in the worst case, construction and exploration of the lattice of consistent cuts. Since this lattice may be prohibitively large to construct, it is useful to construct a homeomorphic image of this lattice that captures all the variables of distributed computation of interest. This construction uses interval clocks introduced by Alagar and Venkatesan [AV01a]. The connection to lattice congruences was explicitly observed by Chakraborty and Garg [CG04]. These results are surveyed in Section 6.

The purpose of this note is to provide the reader with relevant concepts in distributed computing and a brief survey of lattice-theoretic applications in distributed computing. The note is organized as follows. Section 2 provides the basic definitions in distributed computing. Section 3 gives applications of lattice theory in timestamping events, Section 4 in global predicate detection, Section 5 in computation slicing, and Section 6 in partial order trace analysis.

2 Distributed Computation

We will be concerned with a single computation of a distributed program. Each process P_i in that computation generates a sequence of *events*. It is clear how to order events within a single process. If event e occurred before f in the process, then e is ordered before f. How do we order events across processes? If e is the send event of a message and f is the receive event of the same message, then we can order e before f. Combining these two ideas, we obtain the following definition. The *happened-before relation* ($\rightarrow$) is the smallest relation that satisfies

1. If e immediately occurred before f in the same process, then $e \rightarrow f$.

2. If e is the send event of a message and f is the receive event of the same message, then $e \rightarrow f$.

3. If there exists an event g such that $(e \rightarrow g)$ and $(g \rightarrow f)$, then $(e \rightarrow f)$.

In Figure 1, $e_2 \rightarrow e_4$, $e_3 \rightarrow f_3$, and $e_1 \rightarrow g_4$.

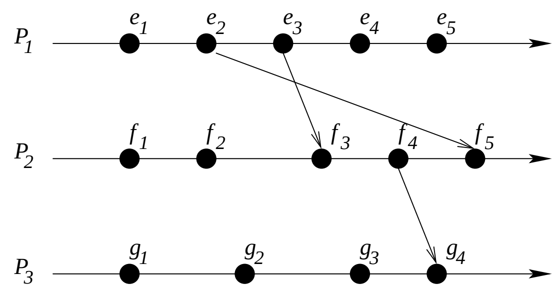


Figure 1: A run in the happened-before model

A *computation* is defined as a tuple $(E, \rightarrow)$ where E is the set of all events and $\rightarrow$ is a partial order on events in E such that all events within a single process are totally ordered. Figure 1 illustrates a run.

A subset G of E is a consistent cut if whenever it contains an element f then it contains all elements e that happened-before f. Thus, G satisfies

$$\forall e, f \in E : (f \in G) \wedge (e \rightarrow f) \Rightarrow (e \in G)$$

3 Timestamping Events and Global States

In this section, we show applications of theory of partial orders to timestamping events in a distributed computation. The goal of timestamping events is to assign a vector of dimension t so that the happened-before relationship can be determined from the vector timestamps. Let $u[i]$ denote the i^{th} component of the vector. Then, we can define a partial order on the set of all vectors of dimension t as follows.

$$\begin{aligned} u < v \quad \equiv \quad & \forall k : 1 \leqslant k \leqslant N : u[k] \leqslant v[k] \wedge \\ & \exists j : 1 \leqslant j \leqslant N : u[j] < v[j] \end{aligned} \tag{1}$$

In a distributed computation, an online algorithm to determine these vector timestamps was first given by Fidge [Fid89] and Mattern [Mat89b]. This algorithm, called vector clock algorithm, works as follows. It is assumed that the distributed computation is on n sequential processes. Each event e is assigned a vector v of dimension n, where $v[i]$ equals the number of events on process i, that are less than or equal to e. Figure 2 gives an example. To implement this assignment in a distributed manner, it is sufficient for each process to maintain a vector that is sent along all outgoing messages. Further, on receiving a message, a process updates its vector as component-wise maximum of its own vector and the vector received with the message. With these rules, it is a simple matter to show that

$$\forall e, f \in E : e \rightarrow f \equiv e.v < f.v$$

It is a natural question to ask whether one can design an algoirthm that uses fewer coordinates than n. The answer to this question follows from the classical dimension theory introduced by Dushnik and Miller. The least k for which events can be assigned vectors of dimension k equals the dimension of the poset. Therefore,

Theorem 1 *For every N, there exists a distributed computation $(E, \rightarrow)$ on N processes such that any assignment from E to $\mathcal{N}^k$ that captures concurrency relation on E has $k \geqslant N$.*

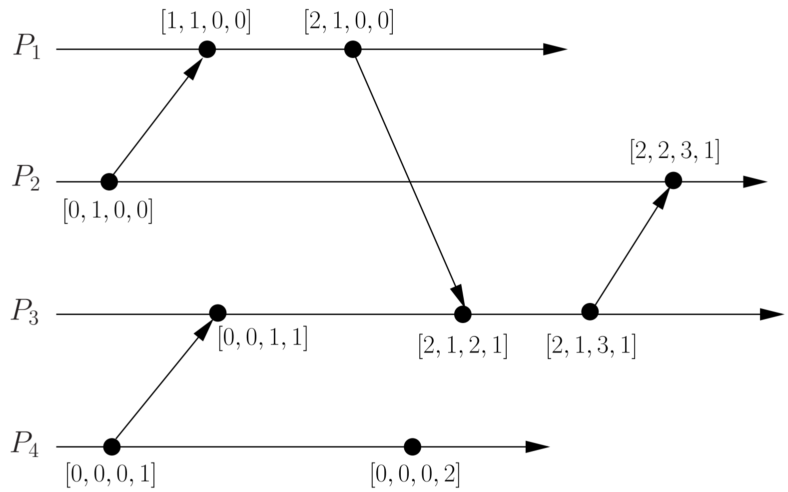


Figure 2: A sample execution of the vector clock algorithm

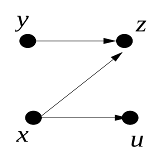


Figure 3: A poset of width 2 forcing an algorithm to use 3 chains for decomposition

Using vectors of size N leads to scalability problems for large systems. Moreover, association of components with processes makes vector clocks cumbersome and inefficient for systems with a dynamic number of processes. In many applications the order between only the *relevant* events needs to be tracked and these relevant events constitute a small percentage of the total number of events. Aggarwal and Garg [AG05] present a class of logical clock algorithms, called chain clock, for tracking dependencies between relevant events based on generalizing a process to any chain in the computation poset. Chain clocks are generally able to track dependencies using fewer than N components and also adapt automatically to systems with dynamic number of processes.

Dilworth's famous theorem states that a finite poset of width k requires at least k chains for decomposition [Dil50]. However, constructive proofs of this result require the entire poset to be available for the partition. The best known *online* algorithm for partitioning the poset is due to Kierstead [Kie81] which partitions a poset of width k into $(5^k - 1)/4$ chains. The lower bound on this problem due to Szemérdi (1982) as given in [Wes04] states that there is no online algorithm that partitions all posets of width k into fewer than $\binom{k+1}{2}$ chains.

However, the problem of online partitioning of the computation poset is a special version of this general problem where the elements are presented in a total order consistent with the poset order. Felsner [Fel97] has shown that even for the simpler problem, the lower bound of $\binom{k+1}{2}$ holds. As an insight into the general result, we show how any algorithm can be forced to use 3 chains for a poset of width 2. Consider the poset given in Figure 3. Initially two incomparable elements x and y are presented to the chain decomposition algorithm. It is forced to assign x and y to different chains. Now an element z greater than both x and y is presented. If algorithm assigns z to a new chain, then it has already used 3 chains for a poset of width 2. Otherwise, without loss of generality assume that the algorithm assigns z to x's chain. Then the algorithm is presented an element u

```
var
    B_1, ..., B_k: sets of queues
        ∀i : 1 ⩽ i ⩽ k, |B_i| = i
        ∀i : q ∈ B_i, q is empty

When presented with an element z:
    for i = 1 to k
        if ∃q ∈ B_i : q is empty or q.head < z
            insert z at the head of q
            if i > 1
                swap the set of queues B_{i-1} and B_i \ {q}
            return
```

Figure 4: Chain Partitioning algorithm

which is greater than x and incomparable to y and z. The algorithm is forced to assign u to a new chain and hence the algorithm uses 3 chains for poset of width 2.

Furthermore, Felsner showed the lower bound to be strict and presented an algorithm which requires at most $\binom{k+1}{2}$ chains to partition a poset. However, the algorithm described maintains many data structures and it can require a scan of the whole poset for processing an element in the worst case. We present a simple algorithm which partitions the poset into at most $\binom{k+1}{2}$ chains and requires at most $O(k^2)$ comparisons per element.

The algorithm for online partitioning of the poset into at most $\binom{k+1}{2}$ chains is given in Figure 4. The algorithm maintains $\binom{k+1}{2}$ chains as queues partitioned into k sets $B_1, B_2 ..., B_k$ such that B_i has i queues. Let z be the new element to be inserted. We find the smallest i such that z is comparable with heads of one of the queues in B_i or one of the queues in B_i is empty. Let this queue in B_i be q. Then z is inserted at the head of q. If i is not 1, queues in B_{i-1} and $B_i \setminus q$ are swapped. Every element of the poset is processed in this fashion and in the end the non-empty set of queues gives us the decomposition of the poset.

To prove the correctness of the algorithm, it is sufficient to show that the algorithm maintains the followings invariant:
(I) For all i: Heads of all nonempty queues in B_i are incomparable with each other.

Note that the algorithm does not need the knowledge of k in advance. It starts with the assumption of $k = 1$, i.e., with B_1. When a new element cannot be inserted into B_1, we have found an antichain of size 2 and B_2 can be created. Thus the online algorithm uses at most $\binom{k+1}{2}$ chains in decomposing posets without knowing k in advance.

4 Detecting Global Predicates

A predicate is simply a boolean function from the set of all consistent cuts to $\{0, 1\}$. Equivalently, a predicate specifies a subset of consistent cuts in which the boolean function evaluates to 1.

We now define various classes of predicates. The class of meet-closed predicates are useful because they allow us to compute the least consistent cut that satisfies a given predicate.

Definition 1 (Meet-Closed Predicates) *A predicate B is meet-closed if for all consistent cuts G, H:*

$$B(G) \wedge B(H) \Rightarrow B(G \sqcap H)$$

The predicate "does not contain x" in the Boolean lattice is meet-closed whereas the predicate "has size k" is not.

In a distributed computation, we define a predicate to be *local* if its truth value depends only on the state of a single process. Any global predicate that can be expressed as a conjunction of local predicates is meet-closed.

It follows from the definition that if there exists any consistent cut that satisfies a meet-closed predicate B, then there exists the least one. Note that the predicate *false* which corresponds to the empty subset and the predicate *true* which corresponds to the entire set of consistent cuts are meet-closed predicates. We now give another characterization of meet-closed predicates that will be useful for computing the least consistent cut that satisfies the predicate. To this end, we first define the notion of a crucial event for a consistent cut.

Definition 2 (Crucial Element) *For a consistent cut $G \subsetneqq E$ and a predicate B, we define $e \in E - G$ to be crucial for G as:*

$$crucial(G, e, B) \stackrel{\text{def}}{=} \forall H \supseteq G : (e \in H) \vee \neg B(H).$$

Definition 3 (Linear Predicates) *A predicate B is linear if for all consistent cuts $G \subsetneqq E$,*

$$\neg B(G) \Rightarrow \exists e \in E - G : crucial(G, e, B).$$

Intuitively, this means that any consistent cut H, that is at least G, cannot satisfy the predicate unless it contains e. Now, we have

Theorem 2 ([CG95]) *A predicate B is linear if and only if it is meet-closed.*

Proof: First assume that B is not closed under meet. We show that B is not linear. Since B is not closed under meets, there exist two consistent cuts H and K such that $B(H)$ and $B(K)$ but not $B(H \sqcap K)$. Define G to be $H \sqcap K$. G is a strict subset of $H \subseteq E$ because $B(H)$ but not $B(G)$. Therefore, G cannot be equal to E. We show that B is not linear by showing that there does not exist any crucial element for G. A crucial element e, if it exists, cannot be in $H - G$ because K does not contain e and still $B(K)$ holds. Similarly, it cannot be in $K - G$ because H does not contain e and still $B(H)$ holds. It also cannot be in $E - (H \cup K)$ because of the same reason. We conclude that there does not exist any crucial event for G.

Now assume that B is not linear. This implies that there exists $G \subsetneqq E$ such that $\neg B(G)$ and none of the elements in $E - G$ is crucial. We first claim that $E - G$ cannot be a singleton. Assume if possible $E - G$ contains only one element e. Then, any consistent cut H that contains G and does not contain e must be equal to G itself. This implies that $\neg B(H)$ because we assumed $\neg B(G)$. Therefore, e is crucial contradicting our assumption that none of the elements in $E - G$ is crucial. Let $W = E - G$. For each $e \in W$, we define H_e as the consistent cut that contains G, does not contain e and still satisfies B. It is easy to see that G is the meet of all H_e. Therefore, B is not meet-closed because all H_e satisfy B, but not their meets. $\Box$

Example 1 Consider the Boolean Lattice generated by all subsets of $\{1, ..., n\}$. Let the predicate B defined to be true on a consistent cut G as "If G contains any odd $i < n$, then it also contains $i + 1$." It is easy to verify that B is meet-closed. Given any G for which B does not hold, the crucial elements consist of

$$\{i | i \text{ is even},\ 2 \leqslant i \leqslant n, i - 1 \in G, i \notin G\}$$

Example 2 Consider a distributed computation on two processes P_1 and P_2 and the predicate B to be true on a consistent cut if both the processes are in the critical section. Given any consistent cut G for which B does not hold, either P_1 is not in the critical section, or P_2 is not in the critical section. In the former case, the next event of P_1 after G, entering the critical section is crucial and in the latter case the event of P_2 entering the critical section is crucial. This example can be easily generalized to any global boolean predicate that can be expressed as a conjunction of local predicates.

Our interest is in detecting whether there exists an consistent cut that satisfies a given predicate B. We assume that given a consistent 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 consistent cut G, then we know that there exists a crucial event in $E-G$. We make an additional assumption:

(Efficient Advancement Property) There exists an efficient (polynomial time) function to determine the crucial event.

We now have an efficient algorithm to find the *least* cut in which B is true. We search for the least consistent cut starting from the *empty* consistent cut. If the global predicate B is true in the current consistent cut, then we have found the least consistent cut in which B is true. If the predicate is false in the consistent cut, then we find the crucial event using the efficient advancement property. We advance to the least consistent cut that includes the crucial event and then repeat the procedure. If we reach the end of the computation without ever discovering any consistent cut that satisfies B, then we return false. Assuming that $crucial(G, e, B)$ can be evaluated efficiently for a given poset, we can determine the least consistent cut that satisfies B efficiently even though the number of consistent cuts may be exponentially larger than the size of the poset. In practice, most meet-closed predicates B satisfy the efficient advancement property. All the examples in this paper do.

So far we have focused on meet-closed predicates. All the definitions and ideas carry over to join-closed predicates. If the predicate B is join-closed, then one can search for the largest consistent cut that satisfies B in a fashion analogous to finding the least consistent cut when it is meet-closed.

Predicates that are both meet-closed and join-closed are called regular predicates.

Definition 4 (Regular Predicates [GM01]) *A predicate is regular if the set of consistent cuts that satisfy the predicate forms a sublattice of the lattice of consistent cuts. Equivalently, a predicate B is* regular *with respect to P if it is closed under $\sqcup$ and $\sqcap$, i.e., for all consistent cuts G, H of the poset P:*

$$B(G) \wedge B(H) \Rightarrow B(G \sqcup H) \wedge B(G \sqcap H)$$

The set of consistent cuts that satisfy a regular predicate forms a sublattice of the lattice of all consistent cuts. Some examples of regular predicates are: "there is no outstanding message in the channel," "every *request* message has been *acknowledged*" in the system," and "there are at most k messages in transit from P_i to P_j."

4.1 Relational Predicates

Another class of predicates that can be efficiently detected is relational predicates. A predicate is relational if it can be expressed in the form of

$$\sum_i x_i \leqslant C$$

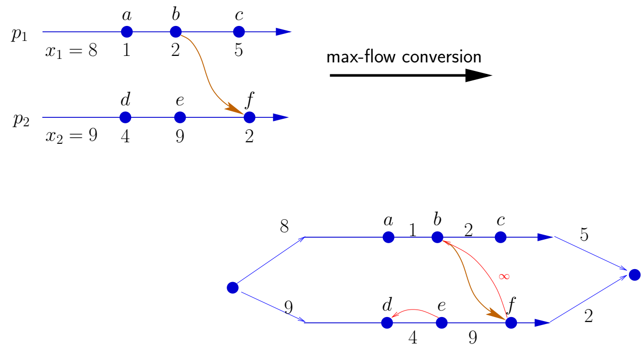


Figure 5: Max Flow Conversion for Detecting Relational Predicates

where x_i is a variable on P_i. To detect such predicates, a distributed computation (a poset) is converted into a max-flow graph. Given an acyclic graph $(E, \rightarrow)$ and a labeling function that assigns value of a variable, it is converted into a max-flow graph such that the consistent cut (the order ideal) of the graph with the least value of Σx_i corresponds to the min cut in the max-flow graph. The value of a cut in the max-flow graph is defined as the sum of capacities of all edges that cross the cut in the forward direction.

We transform the poset 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 with vertex set V and edge set F is obtained as follows. $V = \bigcup_i E \cup \{source, sink\}$ The edge set F is given below.

- First, we add edges from the *source* to all initial events of every process. If the initial value of x_i is c_i, then the capacity of the edge from the source to the initial event of P_i is c_i. For example, in Figure 5, the capacities of these edges are 8 and 9 for P_1 and P_2, respectively.

- For any two events α and β such that α immediately precedes β on the same process P_i and the value of the variable x_i after executing α is $\alpha(x_i)$, we add an edge from α to β with capacity $\alpha(x_i)$ and a directed edge from β to α with capacity ∞. The edge with infinite capacity guarantees that every finite-cut that includes β will also include α. For example, in Figure 5, an edge is added from e to d with capacity ∞. (There are also edges from b to a, c to b, and f to b. These edges are not shown in the figure to avoid the clutter.)

- We add edges from all final events α on P_i to the *sink* with the capacity $\alpha(x_i)$.

- For any two events α and β such that α and β are the send and receive events of a message, We add an edge from β to α with capacity ∞. For example, in Figure 5, an edge is added from f to b with capacity ∞.

In our example, the consistent cut $\{a, b, d, e, f\}$ is the min-cut with the value 4. Note that the $\{a, d, e, f\}$ is not a min-cut because it is not consistent and in the max-flow formulation, it results in cutting the edge (f, b), resulting in infinite cost.

An interesting special case of relational predicates is when all variables x_i are boolean. In particular, consider the question of determining whether there exists a consistent cut such that

$$\sum_i x_i \geqslant K$$

where $x_i \in \{0, 1\}$ and $0 \leqslant K \leqslant N$. In this case, by focusing only on events that have value of x_i's as 1, we can reduce the problem to determining if a given poset has width greater than or equal to K. By Dilworth's theorem, the poset can be partitioned into $K-1$ chains iff there does not exist any antichain of size K. Since the original computation (poset) is given as partitioned into N chains one natural approach is to reduce the number of chains that are used to represent the poset. There are two main techniques discussed in literature for computing the optimal chain partition of a poset.

1. Partitioning the poset greedily into N initial chains and reducing the number of partitions by finding *reducing sequences* [FJN96], or

2. Reducing the problem to a maximum matching problem in a bipartite graph and then using the results of the maximum matching problem [FF62].

For an execution trace of a distributed program, the first approach is more appealing since the history of execution can easily be recorded as a chain on each process. Hence, we do not consider the bipartite approach in this paper. The algorithms by Bogart and Magagnosc [FJN96] and Tomlinson and Garg [TG97] are based on the first approach. Given a chain partition of size K, they answer the question whether the partition could be reduced to $K-1$ chains. The algorithm by Bogart and Magagnosc assume an adjacency list representation of poset and its description can be found in [FJN96]. The algorithm by Tomlinson and Garg [TG97] assume that events are represented using vector clocks.

4.2 General Predicates

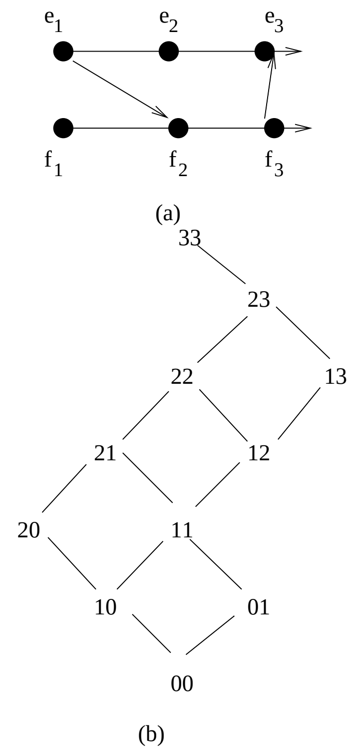
In general, the global predicate detection is a hard problem because of combinatorial state explosion. If there are n processes, each with at most k events, then the total number of consistent global states can be as large as $O(k^n)$. Detecting a simple global predicate such as predicates in 2-CNF form even when no two clauses contain variables from the same process is NP-complete in general [MG01b].

The lattice of all consistent cuts can be traversed in multiple ways as shown in Figure 6(c). Cooper and Marzullo's algorithm[CM91a] performs a breadth-first-search (BFS) traversal and requires space proportional to the size of the biggest level of the lattice which, in general, is *exponential* in the size of the computation. Alagar and Venkatesan's algorithm[AV01a] 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 [Gar03], Garg has proposed 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.

Note that all the traversals are straightforward if one explicitly generates the graph of the 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 lex order is also useful in combinatorial applications. Many families of combinatorial objects can be mapped either to the ideal lattices of appropriate posets [Gar02a] Thus, the algorithm for lex traversal 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. The algorithm for lexcial enumeration of ideals of any poset is



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

generic and by instantiating it with different posets all the above combinatorial lex enumeration can be achieved.

We note here that there are other approaches in mathematics and operations research literature for enumeration of ideals of a poset. See, for example, papers by Steiner [Ste86], Bordat [Bor91], Squire[Squ95], Jegou, Medina, and Nourine [JMN95], and Habib, Medina, Nourine and Steiner[HMNS01]. The algorithm in [HMNS01] 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 order.

5 Slicing Distributed Computations

Suppose we are not interested in all consistent cuts of a computation but in only a subset of them, namely those that satisfy some property of interest to us expressed as a predicate mapping a consistent cut to a boolean value. Further, suppose the set of consistent cuts for which the predicate evaluates to true forms a sublattice of the lattice of consistent cuts. A sublattice of a distributive lattice is also a distributive lattice [DP90a]. Therefore, using Birkhoff's Theorem, the sublattice generated by the consistent cuts satisfying the predicate is completely characterized by

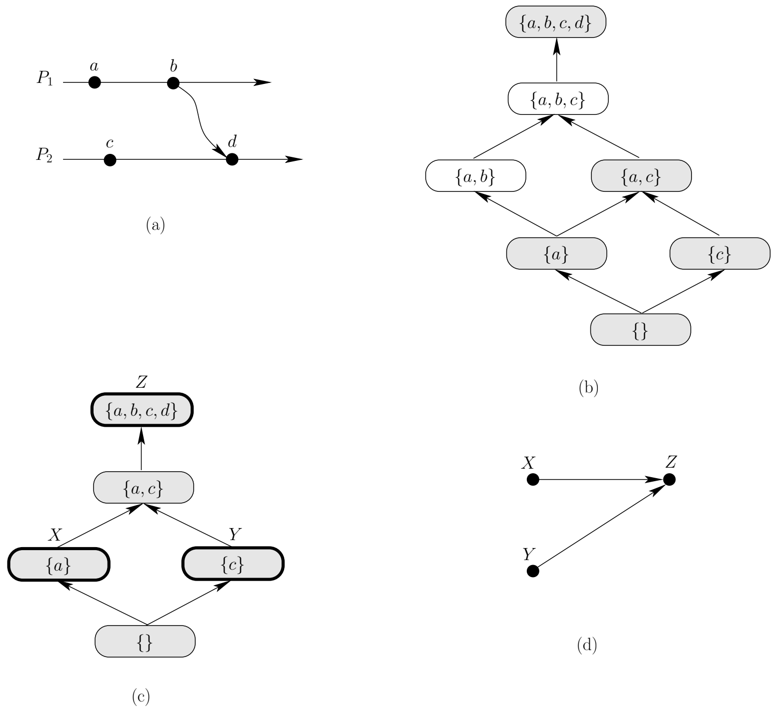


Figure 7: (a) A distributed computation, (b) the distributive lattice generated by its consistent cuts, (c) the sublattice containing all consistent cuts for which no messages are in transit, and (d) the poset induced on the set of join-irreducible elements of the sublattice.

the join-irreducible elements of the sublattice.

Example 3 The distributed computation shown in Figure 7(a) consists of two processes P_1 and P_2. Process P_1 executes events a and b, whereas process P_2 executes events c and d. On executing b, P_1 sends a message to P_2, which is received by P_2 at d. The set of consistent cuts of the computation are shown in Figure 7(b). Suppose we are interested in only those consistent cuts for which no messages are in transit—also known as *strongly consistent cuts*. They have been shaded in Figure 7(b) and are shown separately in Figure 7(c). The set of strongly consistent cuts forms a sublattice and its join-irreducible elements have been drawn with thick boundaries. The poset induced on the set of join-irreducible elements of the sublattice is shown in Figure 7(d).

In case the set of consistent cuts that satisfy the predicate does not form a sublattice, we add one or more other consistent cuts—that do not satisfy the predicate—to complete the sublattice. The consistent cuts are added in such a way so as to minimize the total number of consistent cuts in the resulting sublattice. The sublattice is then represented using the set of its join-irreducible elements. This succinct representation of a possibly large set of consistent cuts satisfying some property is referred to as a *slice* [GM01, MG01a].

Theorem 3 *The slice of a distributed computation is uniquely defined for all predicates.*

The slice for a predicate may contain consistent cuts that do not satisfy the predicate—namely those that are added to complete the sublattice. A slice is *lean* if it contains only those consistent cuts that satisfy the predicate [MG01a]. Clearly, the slice of a computation for a predicate is lean if and only if the predicate is regular.

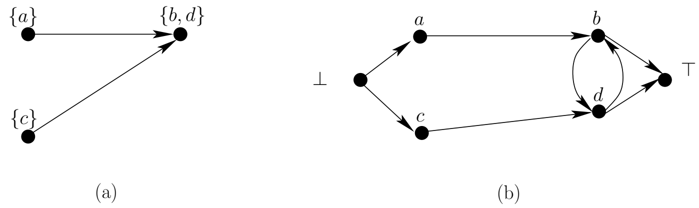


Figure 8: (a) A slice depicting the events that are to executed atomically, and (b) the graph representation of the slice in (a).

Another way of looking at slice is that it specifies which events should be executed in an atomic fashion and the order in which they should be executed. For example, the slice shown in Figure 7(d) and redrawn in Figure 8(a) specifies that events b and d should be executed atomically after events a and c have been executed. This is expected because any consistent cut which includes the send event of a message but not its receive will have at least one message in transit.

For algorithmic purposes, it is more convenient to represent a slice using a directed graph on events possibly containing cycles; all events that are to be executed atomically form a strongly connected component. The notion of consistent cut, of course, has to be extended appropriately.

We define a *consistent cut* (global state) on directed graphs as a subset of vertices such that if the subset contains a vertex then it contains all its incoming neighbors. Observe that the empty set $\emptyset$ and the set of all vertices are *trivial* consistent cuts.

We introduce a fictitious *global initial* and a *global final event*, denoted by $\bot$ and $\top$, respectively. The global initial event occurs before any other event on the processes and initializes the state of the processes. The global final event occurs after all other events on the processes. Any non-trivial consistent cut will contain the global initial event and not the global final event. Therefore, every consistent cut of a computation in the model without $\bot$ and $\top$ is a non-trivial consistent cut of the computation in the model with $\bot$ and $\top$ and vice versa. Note that the empty consistent cut, $\emptyset$ and the final consistent cut E, in the model without $\bot$ and $\top$ correspond to $\{\bot\}$ and $E - \{\top\}$ in our model, respectively.

We denote the slice of a computation $\langle E, \rightarrow \rangle$ with respect to a predicate p by $\mathsf{slice}(\langle E, \rightarrow \rangle, p)$. Note that $\langle E, \rightarrow \rangle = \mathsf{slice}(\langle E, \rightarrow \rangle, \mathsf{true})$. Every slice derived from the computation $\langle E, \rightarrow \rangle$ has the trivial consistent cuts ($\emptyset$ and E) among its set of consistent cuts. A slice is *empty* if it has no non-trivial consistent cuts [MG01a]. In the rest of the paper, unless otherwise stated, a consistent cut refers to a non-trivial consistent cut. In general, a slice will contain consistent cuts that do not satisfy the predicate (besides trivial consistent cuts).

The graph representation of the slice shown in Figure 8(a) is depicted in Figure 8(b). Every sublattice of the lattice of consistent cuts (of a computation) can be generated by a graph obtained by simply adding zero or more edges to the computation [Gar02a].

Now, the slice of a computation for a predicate can be computed as follows. For every pair of events e and f, detect whether there is a consistent cut of the computation satisfying the predicate that contains f but does not contain e. An edge is added from e to f if and only if the detection algorithm returns "no" as the answer. The reason is that, on adding an edge from e to f in a graph, the resulting graph retains all consistent cuts of the original graph except those that contain f but not e. Therefore if no consistent cut satisfying the predicate that contains f but not e exists, then an edge from e to f can be safely added to the graph without eliminating any of the desired consistent cuts. Also, note that given a slice of a computation for a predicate, we can detect the predicate in the computation easily by simply testing the slice for emptiness. Therefore it follows that:

Theorem 4 *There exists an efficient algorithm for computing the slice for a predicate if and only if there exists an efficient algorithm for detecting the predicate.*

More efficient algorithms for computing the slice for special classes of predicates including linear (and regular) predicates, complement of regular predicates, and k-local predicates for constant k can be found elsewhere [GM01, MG01a, MG03].

A useful operation on slices is *composition* [MG01a]. Given two slices, slice composition can be used, for example, to compute a graph whose consistent cuts are exactly those that belong to both the slices. This is referred to as composition with respect to conjunction. Dually, slices can also be composed with respect to disjunction. Slices can be composed by simply manipulating edges in their graph representation. Specifically, to compose slices with respect to conjunction, we add an edge from an event e to an event f if and only if the edge is present in the (transitively-closed) graph representation of at least one of the slices [MG01a]. Similarly, to compose slices with respect to disjunction, we add an edge from an event e to an event f if and only if the edge is present in the graph representation of both the slices [MG01a]. Also, an algorithm to compute the slice with respect to the negation of a regular predicate has been given in [MG01a].

Slicing can be used to facilitate predicate detection as illustrated by the following scenario. Consider a predicate B that is a conjunction of two clauses B_1 and B_2. Now, assume that B_1 is such that it can be detected efficiently but B_2 has no structural property that can be exploited for efficient detection. An efficient algorithm for locating *some* consistent cut satisfying B_1 cannot guarantee that the cut also satisfies B_2. Therefore, to detect B, without computation slicing, we are forced to use techniques such as *breadth first search* [CM91b], *depth first search* [AV01b], and *partial-order methods* (a model-checking technique) [SUL00], which do not take advantage of the fact that B_1 can be detected efficiently. With computation slicing, however, we can first compute the slice for B_1. If only a small fraction of consistent cuts satisfy B_1, then instead of detecting B in the computation, it is much more efficient to detect B in the slice. Therefore by spending only polynomial amount of time in computing the slice we can throw away exponential number of consistent cuts, thereby obtaining an exponential speedup overall. In fact, our experimental results indicate that slicing can indeed lead to an exponential improvement over existing techniques for predicate detection in terms of time and space [MG03, SG03c].

6 Analyzing Partial Order Traces

Traditional techniques for eliminating bugs in concurrent programs (message-passing or shared-memory based) include *testing* and *formal methods*. Testing techniques are ad-hoc and do not allow for formal specification and verification of logical properties that a program needs to satisfy. Formal methods such as model checking and theorem proving do not scale well and need considerable manual effort. Given that formal methods, in general, work on an abstract model of a program and make assumptions on the environment, even if a program has been formally verified, we still cannot be sure of the correctness of a particular implementation. However, for highly dependable systems such as avionics or automobiles, it is crucial to reason on the particular implementation.

We focus on a technique called runtime verification that addresses some of the problems in testing and formal methods. This technique enables automatic verification of implementations of *large* programs using temporal logic specifications. The scalability in runtime verification comes from examining only a *single* execution trace of a program like in testing.

Next we show how to use computation slicing with respect to temporal logic predicates for partial order trace analysis.

We model a finite trace of a program as a partial order between events, for example Lamport's *happened-before* relation [Lam78]. Most runtime verification tools such as MaC tool [KKL$^+$01] and NASA's JPaX tool [HR01] model a trace as a total order (interleaving) of events. Using a partial order model, we can capture exponential number of *possible* total order traces succinctly. This may translate into finding bugs that are not found with MaC or JPaX tools as we show elsewhere. Also, a partial order model is a more faithful representation of concurrency [Lam78] and this model enables us to apply our theory to distributed programs as well as shared memory programs.

6.1 Computation Slices for Temporal Logic Predicates

Many specifications of distributed programs are temporal in nature because we are interested in properties related to the sequence of states during an execution rather than just the initial and final states. Temporal logic enables us to check for safety and livenes properties of programs. Informally, a safety property expresses that "something (bad) will never happen" during a system execution. A liveness property expresses that eventually "something (good) must eventually happen" during an execution. For example, the liveness property in dining philosophers problem, "a philosopher, whenever gets hungry, eventually gets to eat", is a temporal property. The concept of slicing is useful for detecting temporal logic predicates since it enables us to reason only on the part of the global state space that could potentially affect the predicate.

We show in [SG03b] that temporal predicates $\mathbf{EF}(p)$, $\mathbf{EG}(p)$, and $\mathbf{AG}(p)$ are regular when p is regular and we call such predicates as *temporal regular predicates*. Intuitively, $\mathbf{EF}(p)$ checks whether there exists a consistent cut that satisfies p, $\mathbf{AG}(p)$ checks whether p is invariant, that is, all consistent cuts satisfy p, and $\mathbf{EG}(p)$ checks whether p is invariant on a path, where a path is a sequence of consistent cuts starting from the initial consistent cut. More formally, we say that a consistent cut C satisfies $\mathbf{EF}(p)$ if p holds for some consistent cut on some path from C to the final consistent cut. We say that a consistent cut C satisfies $\mathbf{EG}(p)$ (resp. $\mathbf{AG}(p)$) if p holds for all cuts on some (resp. all) path from C to the final consistent cut. Slicing algorithms in [GM01, MG01a] for regular predicates assume the efficient advancement property and the property that given a consistent cut, it is efficient to determine whether the predicate holds for the cut or not. However, these properties do not hold for temporal regular predicates. The predicate detection problem (runtime verification) is to decide whether the initial cut of the computation satisfies a given predicate.

Examples of temporal predicates are the complement of the liveness property in dining philosophers stated above in temporal form as $\mathbf{EF}(\mathit{hungry} \wedge \mathbf{EG}(\neg \mathit{eat}))$ or "the reset state is eventually reachable" in temporal form as $\mathbf{AG}(\mathbf{EF}\ \mathit{reset})$. Next, we briefly describe our computation slicing algorithms presented in [SG03b].

Since the consistent cuts of the slice of a computation is a subset of consistent cuts of the computation, the slice can be obtained by adding edges to the computation. In other words, the slice contains *additional edges* that do not exist in the computation. Below, we will show which edges we should add to a computation for computing slices.

Now we explain Algorithm A1 in Figure 9 for generating the slice of a computation with respect to $\mathbf{EF}(p)$. From the definition of $\mathbf{EF}(p)$, all consistent cuts of the computation that can reach the greatest consistent cut that satisfies p, call this cut W, also satisfies $\mathbf{EF}(p)$. Furthermore, these cuts are the only ones that satisfy $\mathbf{EF}(p)$. We can find W using $\mathsf{slice}(\langle E, \rightarrow \rangle, p)$ when it is nonempty. To ensure that all cuts that cannot reach W do not belong to $\mathsf{slice}(\langle E, \rightarrow \rangle, \mathbf{EF}(p))$, we add edges from $\top$ to the successors of events in the frontier of W in $\langle E, \rightarrow \rangle$. A *frontier* of a consistent cut is the set of those events of the cut whose successors, if they exist, are not contained in the cut. Adding an edge from $\top$ to an event makes any cut that contains that event trivial.

Algorithm A1

Input: A computation $\langle E, \rightarrow \rangle$ and slice($\langle E, \rightarrow \rangle, p$)
Output: slice($\langle E, \rightarrow \rangle, \mathbf{EF}(p)$)
1. Let G be $\langle E, \rightarrow \rangle$ and let W be the final cut of slice($\langle E, \rightarrow \rangle, p$)
2. **If** W exists **then**
3. $\quad$ $\forall\, e \in frontier(W)$: add an edge from the vertex $\top$ to $succ(e)$ in G
4. $\quad$ **return** G
5. **else return** empty slice

Algorithm A2

Input: A computation $\langle E, \rightarrow \rangle$ and slice($\langle E, \rightarrow \rangle, p$)
Output: slice($\langle E, \rightarrow \rangle, \mathbf{AG}(p)$)
1. Let G be slice($\langle E, \rightarrow \rangle, p$)
2. For each pair of vertices (e, f) in G such that,
$\quad$ (i) $\neg(e \rightarrow f)$ in $\langle E, \rightarrow \rangle$, and
$\quad$ (ii) $(e \rightarrow f)$ in G
add an edge from vertex e to the vertex $\perp$
3. **return** G

Algorithm A3

Input: A computation $\langle E, \rightarrow \rangle$ and slice($\langle E, \rightarrow \rangle, p$)
Output: slice($\langle E, \rightarrow \rangle, \mathbf{EG}(p)$)
1. Let G be slice($\langle E, \rightarrow \rangle, p$)
2. For each pair of vertices (e, f) in G such that,
$\quad$ (i) $\neg(e \rightarrow f)$ in $\langle E, \rightarrow \rangle$, and
$\quad$ (ii) $(e \rightarrow f)$ and $(f \rightarrow e)$ in G
add an edge from vertex e to the vertex $\perp$
3. **return** G

Figure 9: Algorithm for generating a slice with respect to $\mathbf{EF}(p)$, $\mathbf{AG}(p)$ and $\mathbf{EG}(p)$

The following theorem is crucial in obtaining Algorithm A2 in Figure 9 that generates the slice for $\mathbf{AG}(p)$.

Theorem 5 ([SG03a]) *Given a computation $\langle E, \rightarrow \rangle$ and* slice($\langle E, \rightarrow \rangle, p$)*, a consistent cut D in $\langle E, \rightarrow \rangle$ satisfies $\mathbf{AG}(p)$ iff it includes vertex e of every additional edge (e, f) in* slice($\langle E, \rightarrow \rangle, p$)*.*
Proof Sketch:
If a consistent cut D does not include vertex e then there exists a consistent cut H that can be reached from D in the computation such that H does not include e but includes f. In this case, it is clear that H does not satisfy p since (e, f) is an edge in the slice($\langle E, \rightarrow \rangle, p$) and every consistent cut of slice($\langle E, \rightarrow \rangle, p$) that includes f must include e. Therefore from the definition of $\mathbf{AG}(p)$, D does not satisfy $\mathbf{AG}(p)$.

Now we prove the other direction. If a consistent cut D does not satisfy $\mathbf{AG}(p)$ then there exists a consistent cut H reachable from D such that H does not satisfy p. We know that only the consistent cuts that include f but not e do not satisfy p. Since H is reachable from D and H does not include e, we have that D also does not include e. $\square$

Since the consistent cuts that satisfy $\mathbf{AG}(p)$ is a subset of consistent cuts that satisfy p, the slice for $\mathbf{AG}(p)$ can be obtained by adding edges to the slice for p. Using the above Theorem, we add an edge from e to $\perp$ for any additional edge (e, f) in slice($\langle E, \rightarrow \rangle, p$) to obtain slice($\langle E, \rightarrow \rangle, \mathbf{AG}(p)$).

This ensures that consistent cuts that do not include vertex e of any additional edge (e, f) are disallowed, whereas the rest belongs to $\mathsf{slice}(\langle E, \rightarrow\rangle, \mathbf{AG}(p))$.

The algorithm for $\mathbf{EG}(p)$ slicing displayed in Figure 9 is similar to the $\mathbf{AG}(p)$ slicing algorithm. However in this case, for each additional edge (e, f) *that generates a non-trivial strongly connected component* in $\mathsf{slice}(\langle E, \rightarrow\rangle, p)$, we add an edge from the vertex e to the vertex $\perp$. Intuitively, if a cut C does not include such a component then, as in the case of $\mathbf{AG}(p)$, there exists a cut D reachable from C such that D does not satisfy p. However, different from $\mathbf{AG}(p)$ case, now there exists such a cut D on *all* paths from C to the final state. Using the definition of $\mathbf{EG}(p)$, it is clear that C does not satisfy $\mathbf{EG}(p)$.

6.2 Using Lattice Congruences to Reduce Global State Graph

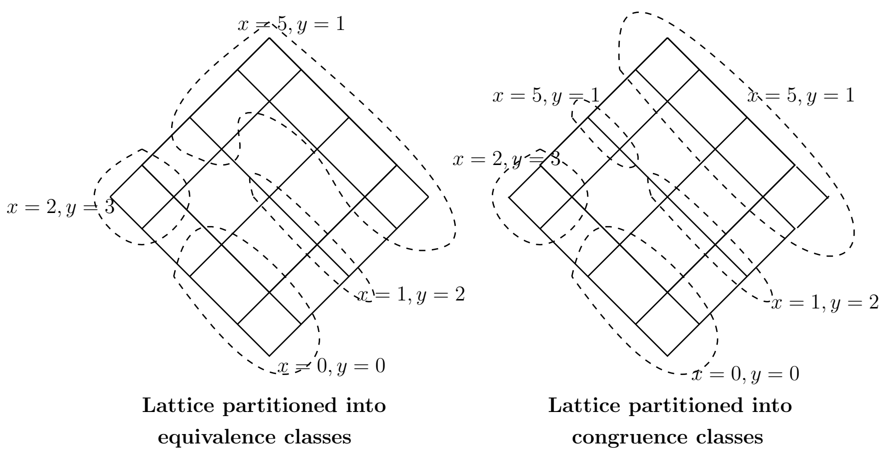


Figure 10: The problem of finding the largest congruence θ that is smaller than a given equivalence partition of the global state lattice L.

The theory of lattice congruences has found applications in reducing the size of the global state graph for analysis of distributed computations. In analyzing the lattice of global states, we are interested in grouping together global states which have the same state with respect to the property that we wish to verify. For example, if we are interested in detecting the property $(x^2+y > 10)$, then it would simplify the problem of detection if we can group together states that have the same x and y values. Doing this will induce an equivalence relation on the lattice and partition it into equivalence classes. However the structure formed by collapsing together the equivalence class elements does not in general form a lattice and thus does not represent a valid distributed computation. The reduced structure should also be a distributive lattice, in order for us to be able to apply other detection techniques on this reduced lattice. Congruences are equivalence relations that preserve distributivity in the reduced lattice [DP90b]. The set of all congruences of a lattice also forms a lattice structure. Thus the *largest* congruence that is contained in a given equivalence relation is well defined. The problem of finding the greatest state space reduction possible is equivalent, to the problem of finding the largest congruence that is contained in the equivalence relation derived from the properties that we are trying to verify. Figure 10 illustrates this problem. The top most equivalence class in the first lattice is not a congruence class and hence it needs to be partitioned into two, to form a congruence class (second lattice). Note that the structure formed by collapsing the equivalence classes in the first lattice does not form a distributive lattice while it does so in the second lattice.

In [CG04], it is shown that checking for any temporal logic formula Φ in CTL (without the next time operator) in the original lattice is equivalent to checking it in the quotient lattice (the quotient lattice is obtained using the lattice congruence relation derived from Φ.)

7 Conclusions

The theory of posets and lattices has many practical applications in distributed computing. Besides the applications in predicate detection, lattice theory is also useful in predicate control [TG99, MG00]. We believe that the future will bring even more applications of the theory of order to distributed computing. For example, the concepts of Möbius functions, Zeta polynomial and Generating functions (see the book on Enumerative Combinatorics, Vol 1, by R.Stanley Chapter 3 [Sta86]) in posets, or modular lattices, geometric lattices etc. (see the book on General Lattice Theory by Grätzer [Gra78]) have not yet found applications in distributed computing.

References

[AG05] A. Agarwal and V. K. Garg. Efficient dependency tracking for relevant events in shared-memory systems. In *PODC*, pages 19–28, 2005.

[AV01a] 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.

[AV01b] 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.

[CG95] C. Chase and V. K. Garg. On Techniques and their Limitations for the Global Predicate Detection Problem. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 303–317, France, September 1995.

[CG04] A. Chakraborty and V. K. Garg. On reducing the global state graph for verification of distributed computations. Technical report, University of Texas at Austin, 2004. Available as `"http://maple.ece.utexas.edu/TechReports/2004/TR-PDS-2004-002.ps"`.

[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.

[CM91a] 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.

[CM91b] R. Cooper and K. Marzullo. Consistent Detection of Global Predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, California, 1991.

[Dil50] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51:161–166, 1950.

[DP90a] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[DP90b] B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.

[Fel97] S. Felsner. On-line chain partitions of orders. *Theoretical Computer Science*, 175:283–292, 1997.

[FF62] L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[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.

[Fid91] C. J. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer*, 24(8):28–33, August 1991.

[FJN96] R. Freese, J. Jaroslav, and J. B. Nation. *Free Lattice*. American Mathematical Society, 1996.

[Gar02a] V. K. Garg. Algorithmic combinatorics based on slicing posets. In *Proc. of 12th Conference on the Foundations of Software Technology & Theoretical Computer Science*, pages 169–181. Springer Verlag, December 2002. Lecture Notes in Computer Science.

[Gar02b] V. K. Garg. *Elements of Distributed Computing*. John Wiley and Sons, Incorporated, New York, NY, 2002.

[Gar03] V. K. Garg. Enumerating global states of a distributed computationin lexicographic and breadth-first manner. In *International Conference on Parallel and Distributed Computing Systems*, pages 134–139, November 2003.

[GM01] V. K. Garg and N. Mittal. On Slicing a Distributed Computation. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 322–329, Phoenix, Arizona, April 2001.

[GMS03] V. K. Garg, N. Mittal, and A. Sen. Applications of lattice theory to distributed computing. *ACM SIGACT Notes*, 34(3):40–61, September 2003.

[Gra78] G. Gratzer. *General Lattice Theory*. Academic Press, New York, NY, 1978.

[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.

[HR01] Klaus Havelund and Grigore Rosu. Monitoring java programs with java pathexplorer. *Electr. Notes Theor. Comput. Sci.*, 55(2), 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.

[Kie81] H. A. Kierstead. Recursive colorings of highly recursive graphs. *Canad. J. Math*, 33(6):1279–1290, 1981.

[KKL$^+$01] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-mac: a run-time assurance tool for java programs. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.

[Mat89a] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms: Proceedings of the Workshop on Distributed Algorithms (WDAG)*, pages 215–226. Elsevier Science Publishers B. V. (North-Holland), 1989.

[Mat89b] 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.

[MG00] N. Mittal and V. K. Garg. Debugging Distributed Programs using Controlled Re-execution. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 239–248, Portland, Oregon, July 2000.

[MG01a] N. Mittal and V. K. Garg. Computation Slicing: Techniques and Theory. In *Proceedings of the Symposium on Distributed Computing (DISC)*, pages 78–92, Lisbon, Portugal, October 2001.

[MG01b] 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.

[MG03] N. Mittal and V. K. Garg. Software Fault Tolerance of Distributed Programs using Computation Slicing. In *Proceedings of the 23rd IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 105–113, Providence, Rhode Island, May 2003.

[SG03a] A. Sen and V. K. Garg. Automatic Generation of Computation Slices for Detecting Temporal Logic Predicates. Technical Report TR-PDS-2003-001, The Parallel and Distributed Systems Laboratory, Department of Electrical and Computer Engineering, The University of Texas at Austin, 2003.

[SG03b] A. Sen and V. K. Garg. Detecting temporal logic predicates in distributed programs using computation slicing. In *International Conference on Principles of Distributed Systems (OPODIS), LNCS*, volume 7, 2003.

[SG03c] A. Sen and V. K. Garg. Partial Order Trace Analyzer (POTA) for Distributed Programs. In *Proceedings of the Third Workshop on Runtime Verification (RV)*, volume 89. Elsevier, 2003.

[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.

[SUL00] S. D. Stoller, L. Unnikrishnan, and Y. A. Liu. Efficient Detection of Global Properties in Distributed Systems Using Partial-Order Methods. In *Proceedings of the 12th International Conference on Computer-Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 264–279. Springer-Verlag, July 2000.

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

[TG97] A. I. Tomlinson and V. K. Garg. Monitoring functions on global states of distributed programs. *J. Parallel Distrib. Comput.*, 41(2):173–189, 1997.

[TG99] A. Tarafdar and V. K. Garg. Software Fault Tolerance of Concurrent Programs Using Controlled Re-execution. In *Proceedings of the 13th Symposium on Distributed Computing (DISC)*, pages 210–224, Bratislava, Slovak Republic, September 1999.

[Wes04] D. B. West. *The Art of Combinatorics Volume III: Order and Optimization*. Preprint edition, 2004.