Efficient Vector Clock for Predicate Detection in Multithreaded Programs

Anurag Agarwal
Dept of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-0233, USA
anurag@cs.utexas.edu

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

Abstract

Monitoring global predicates is a fundamental concern in distributed and concurrent systems for its importance during the design, testing and debugging phases of program development. In this paper, we present a generalization of local predicates called the *chain predicates*. In the context of multithreaded programs, these predicates can be thought of as predicates defined over shared variables instead of local variables as in the case of local predicates. We present a new vector clock algorithm, called the *chain vector clock*, which tracks only the variables relevant to the predicate. This results in a vector clock of size equal to the number of variables being tracked instead of the number of threads in the system. Based on the chain vector clocks, we modify the algorithm for detecting conjunctive predicates to detect conjunction of chain predicates under the possibly modality. The other algorithms for predicate detection which were based on the local predicates can also be generalized for chain predicates. We also present a lattice theoretic characterization of chain predicates which could be useful in further understanding of the properties of these predicates.

1 Introduction

A multithreaded program involves parallel execution of multiple threads. This invariably leads to problems due to potentially infinite ways of interleaving between the threads. Testing and debugging is an important and practical way to ensure reliability and dependability of such systems. For this purpose, predicate detection is a useful abstraction for analyzing the executions of multithreaded programs. For example, while debugging a mutual exclusion algorithm, detecting concurrent accesses to a shared resource is useful.

Our analysis of the multithreaded programs is based on a *partial order model* of the computation. Partial order model, by ordering the events based on causality, takes into consideration multiple *possible executions* of the system through one actual run. Thus, it gives us the capability to *predict* errors by observing even error-free executions. On the other hand, the partial order model also leads to combinatorial explosion problem - the number of global states that the system could have passed through could be exponential in terms of the number of local states. So a simple traversal of all the global states is not good enough. Chase and Garg [GC95] proved that detecting a predicate in 3-CNF is NP-complete in general. This was later generalized by Mittal and Garg [MG01] to prove that the detecting a predicate in k-CNF,

$k > 1$, in which no two clauses contain variables from the same process is NP-complete in general. As a consequence of these hardness results, the algorithms for solving the general version of the problem are exponential [CM91, RJN95, SDSL00]. One way to get around this problem is to consider certain special classes of useful predicates for which the problem can be solved efficiently such as stable [CL85, Bou87, SK86], conjunctive [GW94, MHS96], linear and semi-linear [CG98], and relational [TG97] predicates. Stoller and Schneider [SS95] presented an algorithm for detecting a predicate by decomposing it into multiple predicates each of which could be solved by using the algorithm in [GW94].

In this paper, we present a new class of predicates called *chain predicates* and give algorithms for their detection. The chain predicates are generalization of the local predicates based on the idea of generalizing a "process" to *any chain* in the poset representation of the computation. The conjunction of such predicates constitutes a large class of predicates which are normally required to be detected, especially for multi-threaded programs. Although these predicates are also regular predicates [**?**], but it is useful to study them separately as it leads to more efficient and simple algorithms.

The approach used in this paper is based on a modified version of vector clocks [Fid89, Mat89], called the *chain vector clocks*. Vector clocks play an important role in many distributed algorithms for they provide a means of ordering events and finding dependencies. They have also been used for many predicate detection and runtime verification algorithms [GC95, SRA04]. The conventional vector clock consists of a component for every process or thread in the system. As opposed to this, the size of our vector clock does not depend on the number of processes. Instead the chain vector clock is based on the number of shared variables (and local variables) that need to be monitored for predicate detection. This could be very useful for systems which have large number of threads but the predicate to be detected is based on a few variables. It also ameliorates the need to fix the number of threads apriori which is unrealistic for many systems. Chain vector clock provides the same guarantees as the original vector clocks with respect to events involving shared variables. As a result, it could also be integrated with other algorithms based on vector clocks without much change. Using the chain vector clocks, we present an algorithm based on the Garg and Waldecker's WCP algorithm [GW94] for detecting conjunction of chain predicates under the *possibly* modality [CM91]. The algorithm given by Stoller and Schneider [SS95] can also be used without much change for detecting an even larger class of predicates. This essentially allows us to detect the same classes of predicates for shared variables that were possible with local variables. Although the paper talks mainly about the multithreaded programs, similar techniques would apply to distributed computations as well.

We also present a lattice theoretic characterization of such predicates. This characterization is based on the global state lattice of the computation rather than the poset. It could be useful for further exploring the properties of this class of predicates.

2 Model and Notation

A multithreaded system consists of a set of threads $P = p_1, p_2, \ldots, p_n$. Each thread executes a predefined program. The threads communicate with each other via a set of shared variables. A *local computation* of a thread is defined by a sequence of events that transforms the *initial state* of the thread to the *final state*. At each step, the *local state* of a thread is captured by the initial state and the sequence of events that have been executed till that step. We use the terms thread and process interchangeably.

Each event is an *internal* event, a *write* event, or a *read* event. Read and write events are the read and write of the shared (and local) variables respectively and generically they would be referred to as *access* events. Every event causes the local state of the thread to be updated. The order of events on thread p_i is denoted by $<_{p_i}$ and let $<_P = \cup_{1\leq i\leq n} <_{p_i}$.Furthermore, we associate the following variables with an event e :

- $e(p)$: The process on which e occurred.
- $e(\nu)$: The variable accessed in e.

Let $X = x_1, x_2, \ldots, x_m$ be the set of shared variables. Here, local variables can be considered as a special case of shared variables and can be treated in a similar way. The access to the shared variables is assumed to be atomic. The access order of shared variables also generates dependencies between events. The precedence relation defined by $x_i \in X$ is denoted by $<_{x_i}$. For events e, f, $e <_{x_i} f$ if and only if e and f are access events to the variable x_i and e happened before f. The happened before relation is uniquely defined due to the atomicity of access events. Let $<_X = \cup_{1\leq i\leq m} <_{x_i}$. The set of access events for variable x_i is denoted by $\Psi(x_i)$. For $Y \subseteq X$, $\Psi(Y) = \bigcup_{x\in Y} \Psi(x)$ i.e. $\Psi(Y)$ is the set of access events for all the variables in set Y.

A *multithreaded computation* is modeled by an irreflexive partial order on the set of events of the underlying program's execution. We use $(E, \prec)$ to denote a multithreaded computation with the set of events E and the partial order $\prec$. The partial order $\prec$ is the smallest transitive relation which includes $<_P$ and $<_X$. A *run* of a multithreaded computation $(E, \prec)$ is some total order of events in E consistent with the partial order $\prec$. We use the terms "multithreaded computation" and "computation" interchangeably.

Given a set $F \subseteq E$, the *restriction of* $\prec$ *to* F, denoted $\prec_F$, is the partial order obtained by restricting $\prec$ to F. If $\prec_F$ is a total order then it is called a *chain*. We use $(F, \prec_F)$ to denote the *subcomputation* corresponding to the set of events F. If the partial order is clear from the context, we would simply use F to refer to the poset $(F, \prec_F)$. Therefore, $\Psi(x_i)$ denotes the subcomputation on the access events for variable x_i. Since access to a shared variable is atomic, any two access events can be ordered and hence, this subcomputation is a chain. Such a subcomputation for a shared variable would be called *access chain* for x_i.

A *global state* of a computation is a collection of local states, one from each process. Only the global states which respect causality can arise in the computation. A *cut* can be used to represent a global state and a *consistent* cut represents a possible global state. A cut C is consistent iff, for every event e in C, all its preceding events are also in C.

A *predicate* is a boolean-valued function defined on a cut or global state. The truth value of predicate Φ at the global state g is denoted by $\Phi(g)$. The set of variables on which value of predicate Φ depends is given by $\Pi(\Phi)$. A predicate is *local* iff it is a function of variables of a single thread.

The predicate detection problem can be defined under two modalities, namely *possibly* and *definitely* [CM91]. The predicate *possibly:* Φ is true in a computation iff there is a consistent cut that satisfies Φ. The predicate *definitely:* Φ holds in a computation iff Φ eventually becomes true in all runs of the computation. In this paper, we would mainly focus on observing predicates under *possibly* modality and would omit the word "*possibly*" from the predicate *possibly:* Φ.

equivalent.0

Figure 1: Two representations of the same computation

example.0 example.0

Figure 2: A multithreaded computation

3 Generalizing Threads and Chain Predicates

We generalize the notion of a "thread" in the definition of local predicates for a computation to obtain *chain predicates*. In the poset $(E, \prec)$, threads are certain predefined chains. We can generalize a thread to *any chain* in the poset $(E, \prec)$. In fact, we can rearrange the whole set of events to form a new set of "threads" while keeping the same dependencies. For example, consider the Figure 1. In the Figure 1(a) we represent the computation in the normal way - the threads p_1, p_2, and p_3 are represented as chains and there are dependencies between the chains as well. We could equivalently represent the same computation as Figure 1(b). Here our threads, g_1, g_2, and g_3, are some other chains in the previous poset while maintaining the same dependencies between events. In terms of the poset model, these are equivalent representations of the poset and all the properties and algorithms of the former model would work with this new model as well. At the same time, thinking about the threads in terms of any chain of the poset provides new insights into analyzing the computations and through chain predicates, we present one such application. A chain predicate is defined as follows:

Definition 1 *A predicate Φ is a* chain predicate *iff the poset $\Psi(\Pi(\Phi))$ is a chain.*

Informally, a predicate which is a function of variables having all their access events along one generalized thread is called a chain predicate. For example, consider the multithreaded computation given in Figure 2. It involves three threads, t_1, t_2 and t_3. There is a variable x local to thread t_1 and a shared variable y. The access events of y are the circled events - e, f, g, h. In this computation, the set of circled events form a chain and so any predicate on y, like $y > 0$, can be considered to be a chain predicate. The events for t_1 also form a chain and therefore, any predicate on x is also a chain predicate. In general, any predicate defined on a shared variable is a chain predicate.

A chain predicate may involve more than one shared variable as well. For example, consider a computation in which the access to a set of shared variables, S, is guarded through one lock. By the property of the locks, no two threads can access any variable in S concurrently. The acquisition and release of the lock creates a dependency between any two accesses by different threads. As a result any predicate defined over S is a chain predicate. Due to these features, chain predicates can be very useful for a programming language like Java which guarantees atomicity of shared variable access and uses *monitors* for synchronization among various threads. Informally, a monitor is a lock associated with an object which a thread has to acquire before executing a *synchronized* region. A typical shared object in Java would have synchronized methods for accessing its member variables. As a result, in many cases the predicates defined on an object (with multiple member variables) would be chain predicates.

A chain predicate can be detected very easily. Any thread which is supposed to write to one of the variables involved in the predicate checks if the predicate is true, both before and after writing to the variable. This ensures that the predicate is checked whenever there is a possible change in its value.

4 Chain Vector Clocks

Vector clocks [Fid89, Mat89] provide a simple mechanism for tracking dependency between the events in a computation. In some scenarios like predicate detection, we need to track dependencies only between the *relevant* events. These relevant events are a subset of all the events in the computation e.g. the events where the value of some variable involved in the predicate changes. For this purpose, we describe a vector clock for tracking dependencies between a set of chains in the poset E, called the *chain vector clocks*. Such a vector clock could be particularly useful for tracking the dependencies between the access chains for shared variables. Rather than having a component for every process in the vector clock, we have a component for every chain that needs to be tracked. This could result in substantial savings when the number of processes involved in the computation is significantly higher than the number of chains that need to be monitored.

For the purpose of this paper, we describe the chain vector clocks in terms of the access chains of variables but any other description of the chains would work equally well. A chain vector clock, $V(M,\theta)$, is associated with a set of variables, $M \subseteq X$, and a function θ. The variables in M are the variables that we are interested in monitoring. Function θ provides a mapping from a variable to its chain (or more precisely to the chain number) i.e. for a variable $x \in M$, $\theta(x)$ is the chain to which the variable belongs. For a set $S \subseteq M$, $\theta(S) = \{\theta(x) : x \in S\}$. For an event e, $\theta(e)$ would mean $\theta(e(\nu))$. We also define the *relevant* set of events, R, as $R = \bigcup_{x_i \in M} X_i$.

The chain vector clock V is a m-dimensional vector of natural numbers with a component for every chain. A thread p_i maintains a local vector, V, and every shared variable $x_i \in X$ maintains a vector, V_{x_i}. For an event e on thread p_i, $V(e)$ denotes the value of vector clock at thread p_i when it executes event e. By $\max V, V'$, we denote the vector with $\max\{V, V'\}[j]$ = max $\{V[j], V'[j]\}$ for each $1 \leq j \leq m$. Also, for two m-dimensional vectors we define the 'less than' relation as follows :

$$V < V' \equiv (V \neq V') \wedge (\forall k : 1 \leq k \leq n : V[k] \leq V'[k])$$

The rules for updating the chain vector clock on occurrence of an event e are given in Figure 3.

The following lemmas state the crucial properties satisfied by chain vector clock algorithm:

Lemma 1 *If* $\theta(e) = \theta(f)$*, then either* $e \prec f$ *or* $f \prec e$*.*

Proof: $\theta(e) = \theta(f)$ implies that both e and f access some variable of the same chain. Therefore, the events e and f are comparable and hence, the result follows.

```
Pi::
 var
      V: array[1 ... m] of integer
           initially (∀j : V[j] = 0)

 On occurrence of event e :

      If e ∈ R :
           V[θ(e)] := V[θ(e)] + 1

      If e is access of variable x :
           V := max(V, V_x)
           V_x := V
```

Figure 3: Chain vector clock algorithm

∎

Lemma 2 *If $e, f \in R$ and $f \not\to e$, then $V(e)[\theta(f)] < V(f)[\theta(f)]$.*

Proof: If $\theta(e) = \theta(f)$, then by the given condition and Lemma 1, $e \prec f$. Since the component corresponding to $\theta(f)$ is increased after every event, $V(e)[\theta(f)] < V(f)[\theta(f)]$. So consider the case when $\theta(e) \neq \theta(f)$. Since $V(f)[\theta(f)]$ is the component for the chain $\theta(f)$ and e could not have seen this value as $f \not\prec e$, it follows that $V(e)[\theta(f)] < V(f)[\theta(e)]$.

∎

Lemma 3 *If $e, f \in E$ and $e \prec f$, then $V(e) \leq V(f)$.*

Proof: If $e \prec f$, then there is a path from e to f. For the events along a process the vector clock's value never decreases and on access events we update the vector clock by taking component wise maximum, so :

$$\forall k : V(e)[k] \leq V(f)[k]$$

Hence, $V(e) \leq V(f)$.

∎

The following theorem shows that the chain vector clock satisfies the *strong clock condition* for the relevant events.

Theorem 1 *For $e, f \in R$, $e \prec f \equiv V(e) < V(f)$*

Proof: We first prove that $(e \prec f)$ implies $(V(e) < V(f))$. If $e \prec f$, then by Lemma 3, $V(e) \leq V(f)$. Moreover, since $f \not\prec e$, from Lemma 2, $V(e)[\theta(f)] < V(f)[\theta(f)]$. Hence,

$$(\forall k : V(e)[k] \leq V(f)[k]) \wedge (V(e)[\theta(f)] < V(f)[\theta(f)])$$

Thus $e \prec f \Rightarrow V(e) < V(f)$. The converse follows from Lemma 2.

■

5 Predicate Detection Algorithms for Chain Predicates

In this section we present a brief description of modifications required to the existing algorithms for predicate detection.

5.1 Conjunctive Chain Predicates

A conjunction of chain predicates is referred to as *conjunctive chain predicate* (CCP). So we can write a CCP Φ as $\Phi = \wedge_{i=1}^{m} \Phi_i$, where Φ_i is a chain predicate and m is the total number of chains. For example, in Figure 2 the predicate $x > 0 \wedge y > 0$ is a CCP. CCP is a generalization of the conjunctive predicates which are defined in a similar way for local predicates. Here we adapt the algorithm given by Garg and Waldecker [**?**] to use the chain vector clocks and obtain an algorithm for CCP detection.

For a given CCP Φ, we maintain a chain vector clock $V(M, \theta)$. Here $M = \Pi(\Phi)$ and for $x \in M, \theta(x) = i$ such that $x \in \Pi(\Phi_i)$. In the algorithm, one of the threads serves as a checker. Each of the chain predicate to be detected is associated with a variable l_i for storing the last thread which accessed the chain. At the start of the trace, l_i is set to 1. When a thread p_i accesses a variable $x \in \Pi(\Phi_j)$, the variable l_j is set to i. On access to any shared variable, a thread p_i checks for the truthness of Φ_j, for all j such that $l_j = i$. If Φ_j is true, then p_i sends a message to the checker thread containing the vector clock timestamp of the present event. The setting of l_i, checking for the predicates and the sending of the appropriate messages must be done atomically with the access of the variable. If the variable accesses are synchronized (say by using the *synchronized* keyword in Java), then this can be achieved by simply adding the appropriate code in the synchronized section. Otherwise, we need to associate a lock with every chain and make the execution of these group of statements atomic.

The checker thread collects the responses from the threads and finds a consistent cut that satisfies the CCP. This algorithm is the same as described in the original paper with a slight modification and we would describe it very briefly. The checker process maintains a queue of timestamps for every chain to be monitored and a vector *cut* of timestamps which is a potential global state of the computation. On receiving a timestamp from a thread, the checker thread puts it in the queue corresponding to the chain to which it belongs. The vector *cut* is repeatedly checked for inconsistency. On finding a local state which happened before some other local state in *cut*, the former is removed from the cut and the next local state from the same chain is inserted in *cut*. This is repeated till we manage to find a consistent cut.

5.2 Other Predicate Classes

Stoller and Schneider [SS95] presented an algorithm for detecting a predicate Φ by decomposing the problem into multiple detection problems, each solvable using the Garg and Waldecker algorithm. The same algorithm could be used with predicates based on shared variables by using the algorithm for CCP detection instead of the original Garg and Waldecker algorithm.

6 Lattice Theoretic Characterization of Chain Predicates

In distributed systems, a local predicate is defined as a predicate which depends on the variables of just one process and therefore, can be observed locally. Posets and lattices are widely used for representing a distributed computation and the set of global states in a distributed computation respectively. However, the notion of processes is not defined for lattices or posets in general and the definition of local predicates in these structures is less intuitive. The chain predicates provide a generalization of local predicates and in this section we provide a characterization of chain predicates based on the global state lattice of the computation.

6.1 Notation and Standard Results

In this section we review the notation and some basic results in lattice theory [DP90]. Let X be any finite set. A pair (X,P) is called a partially ordered set or poset if X is a set and P is a reflexive, antisymmetric, and transitive binary relation on X. We call X the *ground set* while P is a *partial order* on X. We simple write P as a poset when X is clear from the context. Let $e, f \in X$. We write $e \leq f$ in P when $(e, f) \in P$. We say that f covers e if $e < f$ and there is no g such that $e < g < f$. If either $e < f$ or $f < e$, e and f are *comparable*. On the other hand, if neither $e < f$ nor $f < e$, then e and f are incomparable. A poset (X,P) is called *chain* if every distinct pair of points from X is comparable in P. Similarly, a poset in an *antichain* if every distinct pair of points from X is incomparable in P.

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 $x \vee y$); and the greatest lower bound of x and y exists, called the *meet* of x and y (denoted by $x \wedge y$). A *sublattice* is a subset of L closed under join and meet. A sublattice for which there exists two elements c and d such that it includes all x which lie between c and d (i.e. $c \leq x \leq d$) is called an *interval lattice* and denoted by $[c,d]$. A lattice L is distributive if for all $x,y,z \in X : x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$.

Next we provide the definition of *order ideals* or *down-set* of a poset P. Let (X,P) be a poset and let $G \subseteq X$. G is called an *order ideal* in (X,P) if $e \in G$ whenever $f \in G$ and $e \leq f$ in P. We simple use *ideal* for order ideal in this paper. Dually, G is an *up-set* in (X,P) if $e \in G$ whenever $f \in G$ and $e \geq f$. Given $x \in X$, we define $\downarrow x := \{y \in X | y \leq x\}$ and $\uparrow x := \{y \in X | y \geq x\}$

Let L denote the family of all ideals of P. Define a partial order on L by $G \leq H$ in L if and only if $G \subseteq H$. It is well known that the set of ideals forms a distributive lattice and conversely every finite distributive lattice can be constructed in this manner. Given a finite distributive lattice L, one can determine the poset that generates L as follows. An element $e \in L$ is *join-irreducible* if it cannot be written as joins of two elements different from e. Pictorially, in a finite lattice, an element is join-irreducible iff it has exactly one lower cover, that is, there is exactly one edge coming into the element. For any $e \in P$, we use $J(e)$ to denote the least ideal in L that contains e. It is easy to show that $J(e)$ is join-irreducible. Let $\mathcal{J}(L)$ denote the set of all join-irreducible elements in L. Birkhoff's theorem states that any finite distributive lattice L is isomorphic to the set of ideals of the poset $\mathcal{J}(L)$. Meet-irreducible elements of L can be defined in an analogous manner. $M(f)$, the greatest ideal that does not contain f, is meet-irreducible. The set of all meet-irreducible elements of L are denoted by $\mathcal{M}(L)$ and Birkhoff's theorem can also be stated using $\mathcal{M}(L)$.

The following lemma shows the way to find a meet-irreducible corresponding to a join-irreducible. It forms the basis of the theory of chain predicates.

Lemma 4 *Let L be a finite distributive lattice. Let $x \in \mathcal{J}(L)$. Then $\hat{x} = \vee(L \setminus \uparrow x) \in \mathcal{M}(L)$.*

6.2 Characterization

Definition 2 *A meet irreducible cover of $x \in \mathcal{J}(L)$ is defined as an upper cover of $\hat{x}$ in Q, where Q is the partial order defined on the set of elements in $\mathcal{M}(L)$. If $\hat{x}$ is a maximal element in Q, then the top element of L can be considered to be meet irreducible cover of x.*

Clearly, the meet irreducible cover of an element in $\mathcal{J}(L)$ is not unique. For our purposes, only one of the meet irreducible cover suffices. So we define the following mapping :

Definition 3 *Given a chain $C \subseteq \mathcal{J}(L)$, we define a injection $\mu_C : C \to \mathcal{M}(L)$ with $\mu_C(x)$ as a meet irreducible cover of x for $x \in C$. This mapping would be called the interval mapping.*

Based on this, we define an event predicate.

Definition 4 *Let P be a poset and L be the lattice of ideals of P. Let $e \in P$. Then, an* event predicate $E(e)$ *for e is defined as the interval sublattice $[J(e), \mu_C(J(e))]$ with $C = J(e)$.*

Intuitively, these are precisely the set of states in which we have executed event e but not executed one of its successors. This successor corresponds to the next event on the "process" which executed e. Although the definition of the event predicate was given with respect to an event e, we can also define an event predicate $E_C(x)$ corresponding to $x \in \mathcal{J}(L)$ and a suitable chain C of elements in $\mathcal{M}(L)$.

Now a chain predicate is a collection of event predicates with an event predicate for every event on which the chain predicate is true. By definition of chain predicates, all these events must form a chain. In terms of the global state lattice, a chain of events would be a chain of elements in $\mathcal{J}(L)$ and a corresponding chain of elements in $\mathcal{M}(L)$. These two chains must satisfy the following property (referred to as *Lattice Chain Property* (LCP)) :

Definition 5 *A chain $C = x_1, x_2, \ldots, x_m$ (of elements in $\mathcal{J}(L)$) and an interval mapping μ_C are said to satisfy the lattice chain property iff*

$$\forall i, j : i > j : x_i \notin [x_j, \mu_C(x_j)]$$

At this point it is not clear if it is possible to find an interval mapping for a chain such that it satisfies LCP. The following theorem shows that it is indeed possible to do so.

Theorem 2 *Consider a chain $C = x_1, x_2, \ldots, x_m$ of elements in $\mathcal{J}(L)$ and an interval mapping μ_C satisfying LCP. If C is extended by adding an element x_{m+1} satisfying the property :*

$$\forall i : i < m+1 : x_{m+1} \notin [x_i, \mu_C(x_i)]$$

then μ_C can also be extended for x_{m+1}.

Proof: Let $B(C) = \bigcup_{i=1}^{m} [x_i, \mu_C(x_i)]$. Suppose we can find a suitable x_{m+1} that does not belong to the previously chosen intervals and is greater than $x_i's$. Since $\forall i : i < m+1 : x_{m+1} \notin [x_i, \mu_C(x_i)] \wedge x_{m+1} > x_i$, $B(C) \subseteq (L \setminus \uparrow x_{m+1})$. Hence $\forall i : \hat{x_{m+1}} \geq \mu_C(x_i)$. So we can extend the mapping μ_C by choosing an appropriate covering meet-irreducible element. Such a successor would exist as we could choose x_{m+1} indicating that $B(C)$ did not include the top element of L.

■

With the guarantee provided by the above theorem, we can define the chain predicate as follows :

Definition 6 *: Let L be a distributive lattice. Let $C = x_1, x_2, \ldots, x_m$ be a chain of elements in $\mathcal{J}(L)$ and an interval mapping μ_C satisfying LCP. Then, a local predicate $B(C)$ is defined to be*

$$B(C) = \bigcup_{i=1}^{m} [x_i, \mu_C(x_i)]$$

This definition conforms to the following important property of a local predicate :
If S is the sublattice of L formed by a local predicate B, then the complement of S, S^c, is also a sublattice. This again stresses the closeness between the concepts of local and chain predicates.

The next two theorems establish this important property satisfied by the chain predicates.

Theorem 3 *Let $B(C)$ be a chain predicate of a distributive lattice L for some chain C of $\mathcal{J}(L)$ and an interval mapping μ_C satisfying LCP. Then $B(C)$ forms a sublattice of L.*

Proof: Let $y, z \in B(C)$. Since $B(C) = \bigcup_{i=1}^{k} [x_i, \mu_C(x_i)]$, $y \in [x_a, \mu_C(x_a)]$ and $z \in [x_b, \mu_C(x_b)]$ for some $a, b \in [k]$. As $x_a, x_b \in C$, without loss of generality assume $x_a < x_b$. This implies that $\mu_C(x_a) < \mu_C(x_b)$. Hence, $z \geq x_a$ and $y \leq \mu_C(x_b)$. Now consider, $y \wedge z$. Since $y \geq x_a$ and $z \geq x_a$, $y \wedge z \geq x_a$. Moreover, $y \wedge z \leq y$ and therefore, $y \wedge z \leq \mu_C(x_i)$. Hence, $y \wedge z \in [x_a, \mu_C(x_a)]$. Similarly, $y \vee z \in [x_b, \mu_C(x_b)]$. This shows that the join and meet of any two elements in $B(C)$ belongs to $B(C)$. Hence, $B(C)$ is a sublattice.

■

Theorem 4 *Let $B(C)$ be a local predicate of a distributive lattice L for some chain C of elements in $\mathcal{J}(L)$ and an interval mapping μ_C satisfying LCP. Then $B^c(C)$, the complement of $B(C)$, forms a sublattice of L.*

Proof: Suppose that $B^c(C)$ is not a sublattice. Then there exist elements $y, z \in B^c(C)$ such that either $y \wedge z \notin B^c(C)$ or $y \vee z \notin B^c(C)$. First suppose that $y \vee z \notin B^c(C)$.

$\quad y \vee z \notin B^c(C)$
$\Rightarrow y \vee z \in [x_a, \mu_C(x_a)]$ for some $x_a \in C$
$\Rightarrow y \vee z \geq x_a$
$\Rightarrow y \geq x_a$ or $z \geq x_a$

Without loss of generality, assume that $y \geq x_a$.

$\quad y \vee z \in [x_a, \mu_C(x_a)]$
$\Rightarrow y \vee z \leq \mu_C(x_a)$
$\Rightarrow y \leq \mu_C(x_a)$
$\Rightarrow y \in [x_a, \mu_C(x_a)]$

This gives us a contradiction as we assumed that $y \in B^c(C)$. Hence, $y \vee z \in B^c(C)$. Similarly, $y \wedge z \in B^c(C)$. Therefore, $B^c(C)$ is a sublattice.

■

Note that in the above proofs we never used the relationship between x and $\mu_C(x)$. We only exploited the ordering between the join-irreducibles and meet-irreducibles involved. This shows that any interval involving join-irreducible and meet-irreducible elements with required ordering property would give us similar results.

6.3 Relation with Simple Predicates

In [Gar02], simple predicates are defined as follows :
Definition : A predicate B is simple if there exists $e, f \in P$ such that

$$\forall G \in I(P) : B(G) \equiv ((f \in G) \Rightarrow (e \in G))$$

This predicate is denoted by $S(e, f)$. Intuitively, a cut G satisfies B iff whenever it includes f it includes e. It was also shown that $\neg S(e, f) \equiv [J(f), M(e)]$.

Using this definition, we find that for an event e, $E(e) = \neg S(succ_\mu(e), e)$ where $succ_m u(e)$ is the successor of the event conforming to the interval mapping μ_C.

In [Gar02], a regular predicate B was defined as :

$$B = \bigwedge_{e,f \in E_B} S(e, f)$$

This implies that,

$$\neg B = \bigvee_{e,f \in E_B} [J(f), M(e)]$$

In other words, the complement of a sublattice can be written as the union of interval lattices of the form $[x, y]$ where $x \in \mathcal{J}(L)$ and $y \in \mathcal{M}(L)$. If the complement of the sublattice is also a sublattice, then the sublattice can itself be written as the union of such interval lattices. This is the best characterization that we know of for a sublattice whose complement is also a sublattice. Note that the chain predicates also have a representation very similar to the negation of regular predicates.

7 Conclusion

In this paper, we presented a technique to analyze shared variables in a way similar to the local variables. For this purpose, we defined a generalization of local predicates called the *chain predicates*. We also defined the *chain vector clock* which could be used for tracking dependency only between the relevant chains in the computation. Chain vector clock was then used for developing an algorithm for detecting conjunction of chain predicates. The algorithms for detecting other classes of predicates based on local predicates can also be modified in a similar way to obtain algorithms for corresponding classes of predicates for chain predicates. We also presented a lattice theoretic characterization of chain predicates.

References

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

[CG98] C. Chase and V. K. Garg. Detection of global predicates : Techniques and their limitations. *Distributed Computing*, 11(4):191–201, 1998.

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

[CM91] R. Cooper and K. Marzullo. Consistent detection of global predicates. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, California, 1991.

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

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

[Gar02] Vijay K. Garg. Algorithmic combinatorics based on slicing posets. In *22nd Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, Kanpur, India, December 2002.

[GC95] V. K. Garg and C. Chase. Distributed detection of conjunctive predicates. In *IEEE International Conference on Distributed Computing Systems*, pages 423–430, June 1995.

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

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

[MG01] N. Mittal and V. K. Garg. On detecting global predicates in distributed computations. In *IEEE International Conference on Distributed Computing Systems*, pages 322–329, Phoenix, May 2001.

[MHS96] M. Raynal M. Hurfin, M. Mizuno and M. Singhal. Efficient distributed detection of conjunctions of local predicates in asynchronous computations. In *8th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, pages 588–594, New Orleans, October 1996.

[RJN95] R. Medina R. Jegou and L. Nourine. Linear space algorithm for on-line detection of global predicates. In *International Workshop on Structures in Concurrency Theory (STRICT)*. Springer-Verlag, 1995.

[SDSL00] L. Unnikrishnan S. D. Stoller and Y. A. Liu. Efficient detection of global properties in distributed systems using partial-order methods. In *12th International Conference on Computer-Aided Verification (CAV)*, volume 1855 of *Lecture Notes in Computer Science*, pages 264–279. Springer-Verlag, July 2000.

[SK86] M. Spezialetti and P. Kearns. Efficient distributed snapshots. In *6th International Conference on Distributed Computing Systems (ICDCS)*, pages 382–388, 1986.

[SRA04] K. Sen, G. Rosu, and G. Agha. Online efficient predictive safety analysis of multithreaded programs. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 04)*, 2004.

[SS95] S.D. Stoller and F. Schnieder. Faster possibility detection by combining two approaches. In *Workshop on Distributed Algorithms (WDAG)*, France, Sept. 1995.

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