Definitely (Definitely:B)

Abhaya Agarwal , Aditya Mahajan and Gaurav Sharma

November 9, 2002

Abstract

In this paper we review the existing literature on detecting definitely:B under different modalities. Cooper and Marzullo's algorithm for detecting definitely:B, Garg and Waldecker Algorithm for string unstrable preocates and and Garg's polynomial space algorithm are discussed. We discuss regular predicates and the decomposition theorem. Later we consider simple predicates and how they are useful in detecting definitely:B. Finally we present an $O(e_1 + e_2)$ algorithm for detecting definitely:B in a two processor system.

1 Introduction

Global predicate detection is of fundamental importance in distributed computing. Many problems in distributed systems like deadlock detection, termination detection can be easily described in terms of observing global predicates. Also many times, we want to enforce some relationships between the states of various processes. By observing the appropriate global predicates, we can determine if the relationships are maintained in the system. Global predicates can be of two types - *stable* and *unstable*. Once it has become true, a stable predicate always remains true. So stable predicates can be detected by using global snapshots [ChaLam85,SpeKea86,Bouge87]. Detecting unstable predicates is more challenging as it may become true and false severel times during the run of the system.Since the problem of predicate detection suffers from the combinactorial explosion, the main emphasis of the research in the area has been to define bigger and bigger classes of predicates which can be detected efficiently. Some importent classes are Linear ,regular , linked and local predicates.

Depending on the kind of application, we may be interested in observing predicates over whole computation, under different modalities for their truthness. Some important modalities are definitely,possibly and controllable modalities [1]. Some authors refer to these as strong and weak predicates [4,7]. [1] gave algorithms for detecting definitly:B and possibly:B where B is any predicate defined on the states of distributed system but their algorithms are exponential in terms of number of processes. [4,7] have described efficient algorithms for the detecting linked and conjunctive predicates under definitely modality. [5] define region predicates and show how to control them in the distributed computation. They have also given the notion of admissible sequence of events and used that to derive an algorithem for controling disjunctive predicates. The notion of slice of a distributed computation with respect to a predicate was introduced and developed extensively in [8]. Using computation slicing, we can reduce the search space for detecting a predicate under various modalities. [8] give efficient algorithms for detecting a regular predicate under possibly,invariant and controllable modalities. Recentely [2] have given an algorithm for observing definitely:B where B is any predicate, that requires only polynomial space. The algorithms in [1] were exponential in terms of space also.

The report is organized as follows. We define various modalities in section 2. In section 3, we describe in detail some of the algorithms mentioned above. In section 4, we define regular predicates and simple predicates and some new results.

2 Various Modalities

A global predicate is a boolean valued function on the local variables of the processes and the messages in the channels. But this definition only captures the truthness of a predicate on a perticular cut of computation. We need to define the meaning of truthness of predicate on the whole computation. Now this can be done in more then one ways, each of which is important in different types of situations. These are called various modalities for truthness of a predicate. *possibly:B* and *definitely:B* are two that seem most intutive.

Define (*possibly:B*) A global predicate B is possibly true for a computation S iff there exists a path from the initial state to the final state in the lattice C(S) along which B is true on some state.

Define (*definitely:B*) A global predicate B is definitely true for a computation S iff for all the paths from the initial state to the final state in the lattice C(S), B is true on some state.

We can easily see that *possibly:B* is useful for detecting bad predicates such as violation of some condition whereas *definitely:B* is useful for checking conditions that we want to enforce on the system. *invariant:B* and *controllable:B* are duals of these modalities.

Define (*invariant:B*) *invariant:B* $\cong\neg$ *possibly* : $\neg B$

Define (*controllable:B*) *controllable:B* $\cong$ $\neg$ *definitely*: $\neg B$

As the name suggests, *invariant:B* represents that B is true throughout on every possible run of the computation whereas *controllable:B* represents a situation where it is possible for B to be true throughout a possible run of computation. Controllable predicates are specially useful in debuggers where we want to run the computations such that they satisfy some added safety conditions. Controllable predicates tell us that it is possible to add extra synchronization to the computation such that predicate is always satisfied in that run.
In this report, we concentrate on the *definitely* modality. *definitely:B* is also refered to as strong B predicate. *definitely:B* represents the guarntee that the predicate indeed becomes true in every execution of computation. In the next section, we will descibe some algorithms proposed for detecting *definitely:B* .

3 Previous Work on Detecting Predicates

The algorithms that have been proposed for *detecting definitely:B* can be devided in two classes. First of these try to address the problem of detecting *definitely:B* where B is any global predicate. But it is known that predicate detection problem suffers from combinactorial explosion, so these algorithms are invariably exponential in terms of number of processes. Algorithms that are efficient in terms of space have been proposed by [2]. The other class comprises of algorithms for detecting *definitely:B* where B belongs to some restricted class of predicates. Efficient algorithms are available for linked predicates and conjunctive predicates.

- **Cooper-Marzullo Algorithm :**

  This algorithm tries to iteratively construct the set of global states that have a level l and it is possible to reach them from the initial event without passing through an state where predicate is true. If anytime this set becomes null, *definitely:B* holds. This algorithm is exponential in terms of number of processes because in the worst case, the algorithm is forced to visit all the consistent states. If there are n processes and k is the maximum number of events on any monitored process, then this algorithm has complexity O(k^n).

Algorithm 1 Cooper & Marzullo's Algorithm for detectign Definitely:Φ

Definitely(Φ) : **begin**
%Synchronize process and distribute Φ
send Φ to all processes
last:=global state S(0,0,....0)
release processes
remove all states in *last* that satisfy Φ
lvl := 1
%Invariant: *last* contains all states of level *lvl* -1 that are accessable
%from S(0,0,...,0) without passing through a state satisfying Φ

- **do** *last* $\neq$ { }$\rightarrow$
  current := states of *lvl* reachable from a state in *last*
  remove all states in *current* that satisfy Φ
  lvl := lvl + 1; last := current
  end do

end
report Definitely:Φ

- **Garg-Waldecker Algorithms :**

  Garg and Waldecker gave algorithm to detect what they call strong unstable predicates in 2 classes-linked predicates and conjunctive predicate. The algorithm for linked predicates is shown in the figure. The algorithm exploits the fact that if B_1 -> B_2 is true then the state in which B_2 occured must have happened before the state in which B_1occured. This means that either both of them are on the same process or there exists a message path between the two. Algorithm uses this message path to convey the information about the occurence of B_1 to B_2.

Algorithm 2 Garg-Waldecker's Algorithm for strong unstable predicates

P_i ::

- **var**
  detectflag: boolean **always** (true iff *curpred* = m+1)
  pred_list: list of {index:1...m;pred:local predicate}
  curpred: integer **initially** 1
- **Upon** (*head(pred_list).index = curpred*)$\bigwedge$*(head(pred_list).pred = true)*
  begin /*Update what predicate is the next one this process is to detect*/
  curpred++ ;
  pred_list := tail(pred_list);
  end ;
- **Upon rcv** (prog , hiscurpred,....) from P_s
  curpred := tail(pred_list)
- **To send** /*we include curpred in message */
  send(prog,curpred,....) to destin;

The Algorithm for the strong conjunctive predicates involves a checker process. The checker process gets debug messages from all the processes indicating the intervals during which their local predicates were true. Now it checks for overlaps between these intervals and tries to find out if the strong predicate was true at some point

Complexity of checker process is O(m^2p) where m is number of processes and p is the at most number of events on any process. Nonchecker process has worst case message complexity of O(m_r) where m_r is the number of program messages received.

Algorithm 3 Algorithm for strong conjunctive checker process

- **var**

  $q_1, q_2.....q_m$: queue of **record** lo , hi : timeinterval ; **end ;**
  changed, newchanged: set of {1,2,.....,m}

- **Upon** recv(elem) from P_k **do**

  insert(q_k , elem) ;
  if(head(q_k) = elem) **then begin**

  * changed := {k} ;
  * **while** (changed $\neq \phi$) **begin**
    newchanged[i] = { } ;
    for i **in** changed, and j **in** [1,2 ...m] **do begin**
    if *head(q_j).lo >head(q_i).hi*
    newchanged = newchanged$\bigcup${i}
    if *head(q_i).lo > head(q_j).hi*
    newchanged = newchanged$\bigcup${j}
    changed = newchanged ;
    for i in changed do deletehead(q_i)

  end

if $\forall i$: ¬emplty(q_i) **then** found = true;
end

Algorithm 4 Algorithm for strong conjunctive process- non checker process P_{id}

Process P_{id} ::

- var

  lcmvector: **array**[1...n] of (0...MAXMID);

  init ∀i:i≠id:lcmvector[i] = 0 , lcmvector[id] = 1 ;
  / last causal msg rcvd from porcess 1 to n, respc. */*
  Current_Interval: **record** lo, hi : (0,MAXMID); **end**;
  firstflag: boolean inint true;
  local_pred: Boolean_Expression ; /*local predicate to be tested*/

- **for** sending **do**

  send(prog,midgen,lcmvector,....);

  lcmvector[id]++ ;

- **Upon** receive (prog, mid , msg_lcmvector,....) **do**

  ∀i : lcmvector[i] := max(lcmvector[i] , msg_lcmvector[i]) ;
  firstflag := true ;

- **upon** (local_pred ↑) $\bigwedge$ firstflag **do**

  Current_interval.lo = lcmvector ;

- **Upon** (local_pred ↓)$\bigwedge$ firstflag **do**

  Current_Interval.hi := lcmvector ;
  send(dbg, Current_Interval) to CHECKERPROC ;
  firstflag := flase ;

- **Polynomial space algorithm :**

  The space requirements of the original Cooper marzullo algorithm were proportional to the size of largest level set. Recently [2] has given an algorithm that requires only polynomial space. The algorithm performs a DFS kind of search on execution lattice recursively. The maximum depth of the recursion is E, where E is total number of events in the system. If global state is described by the n element vector where n is number of processes then space complexity of the algorithm is O(nE).

Algorithm 5 Polynomial space algorithm for definitely:B

- booleam *nonBPath (G , CGS)*
- if *B(G)* return *false*
- if *finalCGS(G)* return *true ;*
- for e ∈enabled(G) do
  - H = G $\bigcup$\{e\}
  - if ¬B(H) then
    - *prenH* \{H - \{f\} | f ∈maximal(H)
    - K = max \{ W ∈*prenH* | ¬B(W) \}
    - if (K = G)
      if *nonBPath(H)* then return false ;
- endfor
- return false ;

4 Regular Predicates

In this section we describe regular predicates that form an important class of predicates. The notion of regular predicates was proposed in [] and various efficient algorithms for this class of predicates have been developed.

Define (Regular Predicate) : Let C(E) be the lattice of consistent cuts of a computation (E, ->). A predicate B is regular iff for any G,H ∈C(E):

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

Regular predicates have many interesting properties. One of them is that the set of consistent cuts satisfying the regular predicate forms a sublattice of lattice of consistent cuts. In terms of slice, the slice with respect to regular predicate is always lean [8]. This induced structure along with the fact that the lattice of consistent cuts is distributive, helps us in designing efficient algorithms for regular predicates. Infact using the slicing techniques, regular predicate can be observed under possibly,invariant and controllable modality efficiently. But the complexity of observing *definitely:B* is still not known [8].

5 Simple predicates

5.1 Simple Predicate S(e,f)

Define ***(Simple Predicate)*** A predicate B is simple if ∃ events e, f ∈P such that
$\forall G \in \mathcal{I}(P) : B(G) \equiv ((f \in G) \Rightarrow (e \in G))$

This is denoted by S(e,f). Thus a simple predicate is of the form: G satisfies B iff whenever it includes f, it includes e.

5.2 Properties of Simple Predicates

1. S(e,f) is co-regular i.e. S(e,f) partitions the lattice into two sub-lattices.

2. ¬S(e,f) is equivalent to the interval lattice [J(f),M(e)].

3. Conjunction of simple predicates defines a regular predicate.
   $\bigwedge_{(e,f) \in E_B}$ S(e,f) = (regular predicate) B , where E_B = { (e,f) | B ⇒S(e,f)}.

Using these propoerties we get some interesting facts about the slices produced by a simple predicate

Fact 1 : No levels are missing in the sub-lattice formed by ¬S(e,f). This follows direclty from property 2

Fact 2 : if B = S(e,f) then
$\forall G \in I(P)$ ∃ atmost one $H \in Succ(G)$: B(G) ≠ B(H)
(Suppose $\exists H_1, H_2$: [{B(G) ≠ B(H_1)} ⋀ {B(G) ≠ B(H_2)}]⇒ [B(H_1) = B(H_2)]
⇒ [B(G) = B(H_1) = B(H_2)] since B and ¬B are both regular.
The same is true for pred(G).

Fact 3 : All simple predicates are trivially definitely true i.e. they are true at initial or the final state.

Fact 4 : Predicates of the form S(e,f) are trivially true at the initial **and** final states. S(e,⊥) is false on the initial cut, while S(⊤,f) is false at the final cut.Thus to account for regular predicates that are not trivially true i.e. they are false at the initial and final state, we have to introduce predicates of the form S(e,⊥) and S(⊤,f) in the conjunction.

5.3 Checking definitely:B using simple predicates

Decomposition Theorem: For any regular predicate B, let E_B = { (e,f) | B ⇒ S(e,f) }. Then,

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

In general, the problem of detecting definitely:B can be divided into detecting predicates of the forms:

1. B = S(e,f)

2. B = S(e,f) ⋀ S(g,h) , where both predicates are definitely true

3. B = S(e,f) ⋀ B_1 where B_1 is a regular predicate and both predicates are definitely true.

Now we explain how these cases can be dealt with.

1. **Checking definitely:B for a simple predicate**

   The only simple predicate that is non-trival is S(⊤, ⊥). But this predicate is false at all global states. So this case in not interesting.

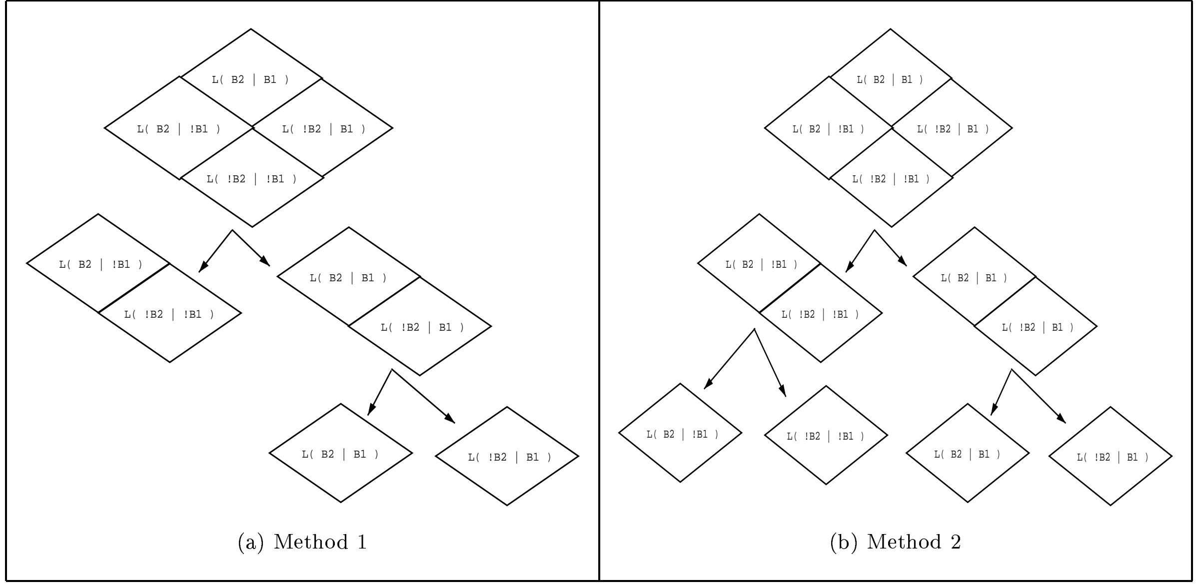
2. **Checking definitely:B for conjunction of two simple predicates**

   In the only non-trivial case, i.e. when the predicate is not true in the initial and the final cut, the predicate has to be of the form

   $$S(\top,f) \bigwedge S(e,\bot)$$

   Such predicates can be detected in the following ways:

   Method 1:



(a) Method 1 (b) Method 2

Let B_1 = S(⊤,f)and B_2 = S(e,⊥). Both B_1 and B_2 are definitely true. So we look at there conjunction. Take the slice of the lattice L, with respect to predicate B_1. As B_1 is a simple predicate, we get two sub-lattices, say L(B_1) and L($\neg B_1$). Take the slice of L(B_1) wr.t. B_2 obtaining L($B_2|B_1$) and L($\neg B_2|B_1$).

Claim: *For any two simple predicates B_1 and B_2*

[∃ e,X,Y ; e∈E(L) , X ∈L($\neg B_2|B_1$) , Y∈L($\neg B_1$) : e = (X,Y)]

*⇒[∃ e',X',Y' ; e'∈E(L) , X' ∈*MIR*(L($\neg B_2|B_1$)) , Y'∈L($\neg B_1$) : e' = (X',Y')]*

Proof: Suppose ∃ e,X,Y ; e∈E(L) , X ∈L($\neg B_2|B_1$) , Y∈L($\neg B_1$) : e = (X,Y). Let B_1 = S(g,h). ∀H ∈L($\neg B_1$) ⇒ (h∈H) ⋀ (g∉H)

∀G ∈L($\neg B_2|B_1$), G ∈L(B_1) ⇒ [(h∈G) ⇒(g∈G)]

Thus, e= (G,H) ⇒ (h∈H)⋀ (g∉H) ⋀ (g∉G) ⋀ (h∉G). Hence we can add the event h to cut G and get the cut H.

If G is a meet irreducible element of L($\neg B_2|B_1$) we are done.

Else assume that there are two atleast two successors G_1 and G_2. ∃h'∈P , h' ≠h such that G⋃{h'} = G_1 (or G_2).

H_1 = H⋃{h'} is a consistent state *{becuase L is a distributive lattice and H_1 = H⨆G_1}* and H_1 ∈L($\neg B_1$) *{h∈H_1 ⋀ g∉H_1}* thus there exists an edge from G_1 ∈L($\neg B_2|B_1$) and H_1 ∈L($\neg B_1$).

Either G_1 is a meet irreducible element or we can find a G_2 and continuing this way we must reach a meet irrdeducible element.

If B_1 and B_2 are reversed then we will be able to find a join irreducible element in L($\neg B_2|B_1$)

Algorithm: Let B_1 = S(e,f) , B_2 =(g,h). B_1 and B_2 are definitely true. B = B_1⋀B_2

- Obtain L(B_1) and L($\neg B_1$).
- Obtain L($B_2|B_1$) and L($\neg B_2|B_1$).
- if ({⊥} ∈L($\neg B_2|B_1$) and {final state}∈ L($\neg B_1$))
  - Find M = {all meet-irreducible elements of L($\neg B_2|B_1$)}
  - A = {G | G∈M⋀(e∉G⋀f∉G))}
  - (∃ G | G⋃{f} is consistent) ≡ ¬(definitely:B)
- else if ({*final* state} ∈L($\neg B_2|B_1$) and {⊥}∈ L($\neg B_1$))
  - Find J = {all join-irreducible elements of L($\neg B_2|B_1$)}
  - A = {G | G∈J⋀(e∈G⋀f∈G))}

- ($\exists$ G | G-{f} is consistent) $\equiv \neg$(definitely:B)

- else (definitely:B)

Proof: The algorithm searches for a path along where $\neg$B on each cut. The only way that definitely:($B_1 \bigwedge B_2$) can be false is that there is a path from the initial cut to the final cut exclusively from L($\neg B_2|B_1$) and L($\neg B_1$). $\neg B_1$ is connected to the final cut and $\neg B_2|B_1$ is connected to the initial cut. For a path to exist there must be one edge from L($\neg B_2|B_1$) to L($\neg B_1$). The algorithm detects such a edge using the above mentioned claim

Method 2: Let B_1 and B_2 be simple predicates. B_1 = S($\top$,f) and B_2 = S(e,$\bot$). B=$B_1 \bigwedge B_2$

Algorithm:

- Slice the lattice L with respect to B_1 to obtain L(B_1) and L($\neg B_1$).
- Now further slice both these lattices with restect to B_2 to obtain lattice L($\neg B_2|\neg B_1$), L($B_2|\neg B_1$), L($\neg B_2|B_1$) and L($B_2|B_1$).
- (L($\neg B_2|\neg B_1$) = $\{\phi\}$) $\equiv$ (B=$B_1 \bigwedge B_2$ is definitely true)

Proof: L($\neg B$) = L($\neg B_2|\neg B_1$) $\bigcup$L($\neg B_2|B_1$) $\bigcup$ L($B_2|\neg B_1$)
f $\in$L($\neg B_1$) $\bigwedge$ f$\notin$L(B_1) . Also e $\notin$L($\neg B_2$) $\bigwedge$ e$\in$L(B_2) .
L($B_2|\neg B_1$) and L($\neg B_2|B_1$) can never be connected by an edge $_{(1)}$
{$\bot$}$\in$ L($\neg B_2|B_1$) and {final state}$\in$ L($B_2|\neg B_1$)

[$\Rightarrow$] (L($\neg B_2|\neg B_1$) = $\{\phi\}$)
From (1) we get that there can be no connection between any two of L($\neg B_2|\neg B_1$), L($B_2|\neg B_1$), L($\neg B_2|B_1$).
Hence definitely:B

[$\Leftarrow$] definitely:B
Thus there is no path from L($\neg B_1$) to L($\neg B_2$)
if (L($\neg B_2|\neg B_1$)$\neq$ $\{\phi\}$) then L($\neg B_2|\neg B_1$) is connected with both L($B_2|\neg B_1$) and L($\neg B_2|B_1$).
Hence L($\neg B_1$) is connected to L($\neg B_2$) $\sharp$

3. **Checking definitely:B for a general case**

In the general case, we have to find the conjunction of a simple predicate with a regular predicate. If we are able to do this, then using the decomposition theorem, we can check for definite modality of any predicate. Here we suggest some method that can be used for detecting definitely:B, B=S(e,f)$\bigwedge B_1$, where $B_1 is$ regular.

- By decomposition theorem B_1 is a conjunction of simple predicates. Thus $\neg B_1$ is a disjunction of negation of simple predicates, so the slice of $\neg B_1$will be a ranked poset
- Consider the ranked poset formed by the union of L($\neg$S) and L($\neg B_1$). Let h be the "height" if this poset.
defintitely:B $\equiv$h $\leq$(n-1), where n is the height of the original lattice.

Thus we need to find an efficient algorithm of determining the height of a ranked poset.

Define (non-trivial Simple Predicate): A predicate $B\prime$ is a non-trivial simple predicate such that [$B\prime$ is false at initial and final state] $\bigwedge$ [$B\prime$=S(e,f) otherwise]
Note that non trivial simple predicates are not regular.

5.4 detecting definitely:B for the type of predicates $B\prime$

B' can be of the following forms

1. B'= { $\begin{array}{cc} false, & at\,initial\,and\,final\,state \\ S(e,\bot) & otherwise \end{array}$ }
   $B\prime$ will not be definitely true only if ¬S(e,⊥) is connected to the final state.
   This is possible only if e is the last event on a process.
2. B'= { $\begin{array}{cc} false, & at\,initial\,and\,final\,state \\ S(\top,f) & otherwise \end{array}$ }
   $B\prime$ will not be definitely true only if ¬S(⊤,f) is connected to the initial state.
   This is possible only if f is the first event on a process.
3. B'= { $\begin{array}{cc} false, & at\,initial\,and\,final\,state \\ S(e,f) & otherwise \end{array}$ }
   $B\prime$ will not be definitely true only if ¬S(e,f) is connected to the initial and also to the final state.
   So height of L(¬S(e,f)) $<$ (n-2)) ⇒ definitely:B

6 definitely:B for Two Processes

6.1 Theorem

Let B be a regular predicate
If ¬B is true at the final state and every join-irreducible element has a ¬B predecessor then
definitely:B is false

Proof: Every state x other than the join-irreducible state has atleast two predecessors, say y and z.
if ¬B(x)
then ¬B(y) $\bigvee$ ¬B(z).
{ if B(y) $\bigwedge$ B(z) ⇒ B(x) as B is regular }
Hence starting from the final state (where the predicate B is false) we can always move to a predecessor state where B is false. Continuing this way, we will reach the initial state and hence we have found a path along which B is always false. Hence definately:B is false.

Define: *Corner Element*
x is a corner element iff
{($\exists m \in MIR$: m ∈predecessor(x)) $\bigwedge$ ($\exists j \in JIR$: j∈successor(x)) }

6.2 Claim

For a two process system,
(definately:B) ⇒ $\exists x \in \{J \bigcup C\} : B(x)$
where
J = (Set of all join irreducible elements)
C = (Set of all corner elements)

Proof: The proof of the claim follows immediately from theorem 6.1

6.3 Algorithm

Functions:

```
boolean canMove( Process P_i )
{ return (P_i.nextState is consistent && ¬B(P_i.nextState) }

void move(Process P_i)
{   move forward on process P_i }

boolean exhaust(Process P_i)
{   check if have reached final state in P_i  }
boolean detect(Process P_i , Process P_j)
//P_i is the process on which we are moving
//P_j is the other process
{
 boolean flag = true ;
 while(¬exhaust(P_i) && flag)
 {
    flag = false ;
    while(canMove(P_i))
    {    move(P_i) ; flag = true ; }
    while(¬exhaust(P_j))
    {
         move(P_j) ; flag = true ;
         if(canMove(P_i) && ¬B(present state))
         {  move(P_i) ; break ; }
    }
 }
 return ¬flag ;
}

boolean detectDefinately()
{   return (detect(P_1,P_2)&&detect(P_2,P_1)) ;   }
```

6.4 Explanation of the proof

The algorithm is based on the claim 6.2. We reduce the original lattice to a sub-lattice, containing initial and final state such that

Lemma 1: One path is eliminated at a step.

Lemma 2: All paths eliminated were *'bad'*.

Lemma 3: Either the condition of the claim are true on the sub-lattice, implying that definately:B is false ;
else such a sub-lattice can not be found, and hence definately:B is true.

Proofs:

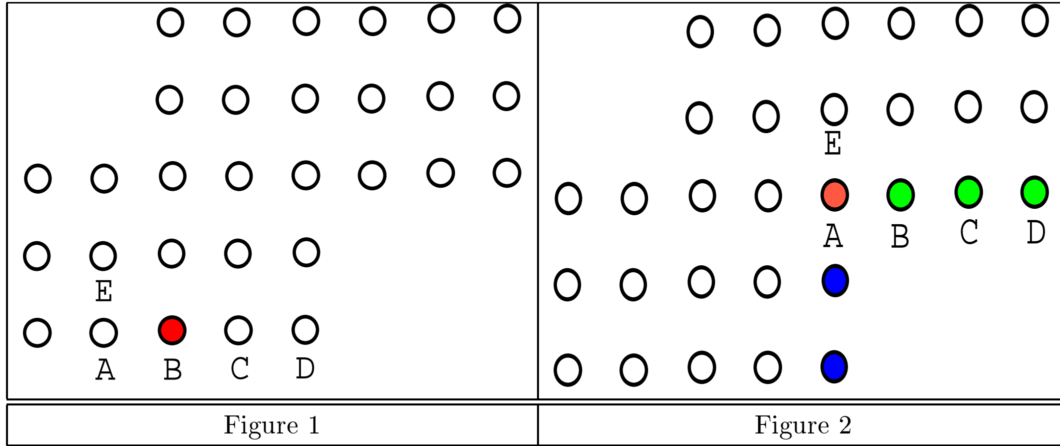


Figure 1 | Figure 2

There are two cases as shown in the above figures. The state where the predicate is true can be a join irreducible state (6.4) or a corner element (6.4). In both cases we go to element *E*. The path that is eliminated is *B*-*C*-*D*. The only way that we can go to these elements is through element where the predicate is true. So only the *bad* paths are eliminated.

We can also consider the elimination of points as a happened-before relationship. We are introducing happened-before realtionship to make the cut where where the predicate is true on a join irreducible element or corner element to an inconsistent state.

When one process is exhausted, we get one boundary of the sub-lattice. We repeat the process for the other process to get the other boundary. Definitely true modality of the predicate can be determined by the fact that the final state is in the sub-lattice or not.

Order of the Algorithm: In the worst case the algorithm takes $2 \times (e_1 + e_2)$ moves, where e_i is the number of events on process P_i . Thus the algorithm is of O$(e_1 + e_2)$

7 Conclusion and Future Work

In this paper we present algorithms for detecting definitely true on a two processor system. We are also able to detect definite:B when B is a conjunction of two simple predicates. We believe that it is possible to detect definitely:B using conjunction of simple predicates.

8 References

[1] 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.

[2] Vijay K. Garg, Detecting Global Predicates in Distributed Computations. Submitted to *IEEE International Conference on Distributed Computing Systems (ICDCS)*, Providence, RI, July 2003.

[3] Vijay K. Garg, Algorithmic Combinatorics based on Slicing Posets. To appear in *Proc. in 22nd conference on the Foundations of Software Technology & Theoritical Computer Science (FSTTCS)*, Kanpur, India, December 2002.

[4] Vijay K. Garg and B. Waldecker, Detection of Strong Unstable Predicates in Distributed Programs. *IEEE Transactions on Parallel and Distributed Systems*, Vol 7, No. 12, December 1996, pp 1323-1333.

[5] Neeraj Mittal and Vijay K. Garg, Debugging Distributed Programs Using Controlled Re-execution. *ACM Symposium on Distributed Computing (PODC'00)*, Portland, Oregan, July 2000, pp. 239-248.

[6] C. M. Chase and Vijay K. Garg, Efficient Detection of Global Predicates in a Distributed System. *Distributed Computing,* Vol. 11, No.4, 1998, pp. 169-189.

[7] Vijay K. Garg and B. Waldecker, Detection of Weak Unstable Predicates in Distributed Programs.*IEEE Transactions on Parallel and Distributed Systems*, Vol 5, No. 3, March 1994, pp 299-307.

[8] Neeraj Mittal and Vijay K. Garg, Computation Slicing: Techniques and Theory. *5th International Symposium on Distributed Computing(DISC'01).* Lisbon, Portugal, 2001. pp. 593-606.