**The Report Committee for Don Tak-San Wong**
**Certifies that this is the approved version of the following report:**

# Distributed Global Predicate Detection Algorithms

Three Distributed Implementations for Algorithms to Detect Possibly True Global States

**APPROVED BY**
**SUPERVISING COMMITTEE:**

**Supervisor:**

Vijay Garg

Thomas Graser

# Distributed Global Predicate Detection Algorithms

Three Distributed Implementations for Algorithms to Detect Possibly True Global States

**by**

**Don Tak-San Wong, B.S.**

**Report**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Master of Science in Engineering**

**The University of Texas at Austin**

**August 2012**

# Abstract


## Distributed Global Predicate Detection Algorithms

Three Distributed Implementations for Algorithms to Detect Possibly True Global States

Don Tak-San Wong, MSE

The University of Texas at Austin, 2012

Supervisor: Vijay Garg

Detecting the existence of a consistent global state that satisfies a predicate in a distributed environment is a processing intensive task since all the consistent global states must be checked to verify that none of them satisfies the predicate. Three different serial implementations have been provided for a breath-first, depth-first, and lexical traversal of the lattice generated by enumerating the possible consistent global states has been provided by Alagar and Venkatesan, Cooper and Marzullo, and Garg. This paper modifies those implementations to perform the checks in a distributed environment, providing the final algorithms, source code, and preliminary results for comparisons with the original algorithms.

# Table of Contents

# List of Tables

# List of Figures

## MOTIVATION

In 2003, Lips [5] implemented the algorithms proposed by Alagar and Venkatesan [1], Cooper and Marzullo [2], and Garg [3]. These three algorithms were implemented to determine if a predicate was possibly satisfied in any of the global state generated in the lattice representing the possible execution paths, with each of the algorithms performing traversal of the lattices in breath-first, depth-first, or lexical order depending on the algorithm respectively.

This project provides an extension to the three algorithms implemented. Providing an implementation that allows the determination to be performed in parallel amongst distributed systems instead of in serial as described in the original papers.

## CONCEPTS

As with the work from Lips, the paper is built with the definition that a consistent global state is a global state where if the receive event is in the global state, then the corresponding send event causing the receive event is also in the global state. A global state is just a set of local states that are concurrent, or have no happens-before relationship, with each other [4]. Each individual local state is composed of a state variable to denote the event in the current process, along with a vector clock to denote when the event occurred in relations to the other processes.

The implementation of the algorithms continues to use the data structure of a lattice to represent the set of possible consistent global states and the observed execution amongst the states.

## IMPLEMENTATION LANGUAGE

Since this is an extension of an existing work, the choice of continuing to use Sun Microsystem's Java Programming Language [6] seemed to be the natural choice, avoiding any unnecessary porting of the existing code to another language. In addition to the availability of the language and open source Integrated Development Environment [7], the language also provided the necessary APIs for performing networking to allow for communication between the distributed systems. Although other popular alternatives such as C# and .NET also provided the necessary libraries, they did not have enough advantages compared to Java to warrant the rewriting of all existing code to the new language.

## DESIGN

The original paper divided the design of the implementation into two aspects: the data structure models and the implementation of the algorithms themselves. In the initial endeavor to make the algorithms parallel, no changes were necessary for the data structures themselves, though such a change could potentially enhance the implementation in terms of simplicity or performance.

The original implementation of the algorithms itself was also divided into sub-categories: Common Functionality, Template Methods, and Predicates. Common Functionality contained functionality used by all 3 algorithms, and as such this included additions to help with the networking aspects of the implementations hence having the largest amount of changes. The Template Method involved the use of inheritance, as such did not include a large amount of changes since no new classes were introduced but rather extensions to existing classes were done. Lastly, Predicates, which were the interface to which an algorithm would attempt to detect, had no changes either, as the logic behind it remained the same also.

### Algorithms for Possibly True

As with the original version of the code, the three algorithms contained different code at the core for performing the lattice traversal while sharing a large subset of supporting functionalities. The extensions continued to use the `AbstractPossiblyTrue` class as the abstract base class, residing in the `wong.master.common` package. This abstract class continues to provide common functionality as well as defining the template method for the inheriting classes. This section summarizes the original functions provided as well as addition made to the original version.

### COMMON FUNCTIONALITY

Functionalities common between all three algorithms are provided in the `AbstractPossiblyTrue` class along with the `ProcessTraceReader` class. Each of the three algorithms inherits from the AbstractPossiblyTrue class directly to get access to the functionalities in the base `AbstractPossiblyTrue` class. The following is the list of functions used by all the classes

- The input to each function is the trace file described in section 4.3. These files reside on the primary system and will be sent to the secondary systems upon connection. These files are then loaded into memory via `ProcessTraceReader` for traversal by each of the algorithms.
- Since the predicate checking for a particular global state is common across the algorithms, and the difference between the algorithms is only in the order of traversal of the global states, the predicate checking function is also defined in the `AbstractPossiblyTrue` class.
- The reporting of results is another common feature between all three algorithms. As such the formatting and display of the possibly true candidates or lack thereof is performed in the `AbstractPossiblyTrue` class all three algorithms would inherit from.
- There is no special case for any of the algorithms for connecting from the primary to the secondary, they all need to connect to the secondary client and transmit the data file before any of calculation is performed. Hence this initial process before the actual execution of the individual algorithms is defined again in the `AbstractPossiblyTrue` class to be inherited from also.

### ABSTRACT METHOD

`AbstractPossiblyTrue` continues to only have one abstract method, which is the `compute()` method. This method is required to be defined by each of the three

algorithms, and is implemented based on how the algorithm will traverse the lattice. The information gathered and used such as connections available and trace files are generally common across all three algorithms, hence the member variables are in the `AbstractPossiblyTrue` class instead of the class defining the algorithms themselves. The actual implementation of this function will define how to traverse the lattice.

#### PREDICATES

The predicates used by each of the three algorithms did not change. The predicates used are required to implement the `GlobalStatePredicate` interface, which defines the `execute()` method and the `toString()` method. The `execute()` method takes in an `ArrayList` representing the current global state, to which the predicate will be applied, returning the `Boolean` value representing rather the global state satisfies the predicate. The `toString()` method will return the predicate's class name along with an optional value representing the predicate's current state.

Not defined in the `GlobalStatePredicate` interface is a constructor. One can still be provided if the predicate used has parameters requiring initialization before being used though.

## Input Format

The flat file input format proposed is also kept to avoid the need to perform serialization and de-serialization of the data when transferred between the primary and the secondary clients. The syntax for each input file remains of the form

```
state=v;vectorClock=x,y,z
```

where such an entry composes a line. The `state=` portion determines what the value is at the instance represented by the `vectorClock` value. The `vectorClock` itself does not need to have exactly 3 entries as long as the entries are delimited by comma (,).

## Graphical User Interface

Two new fields were introduced into the graphical user interface for the primary client in addition to what was there before, now providing the following list of inputs as shown in Figure 1:

- Drop down menu for selecting which `PossiblyTrue` algorithm to use.
- Text input field for the trace files to process, where each individual file should be space delimited
- Drop down menu for selecting available predicates.
- Input field for the parameter to check for if required by the predicate
- Input field for the number of secondary clients. Computation will not start until it is connected to this number of clients, and after it has connected to this number of clients it will not wait for any more connections.
- Input field for the port number the primary client is listening on.
- Checkbox for amount of output. If neither `Trace` nor `Debug` is checked, it will be minimal output, providing the consistent global state that satisfies the predicate if available, stating otherwise if not. If `trace` is checked, each consistent global state checked will be displayed along with the result of the computation. If `debug` is checked, `trace` will be enabled by default and intermediate steps between the computations will also be displayed.
- All output for the primary client will be displayed in the text area provided in the graphical user interface. This area is updated as input from `stdout` and `stderr` is available.

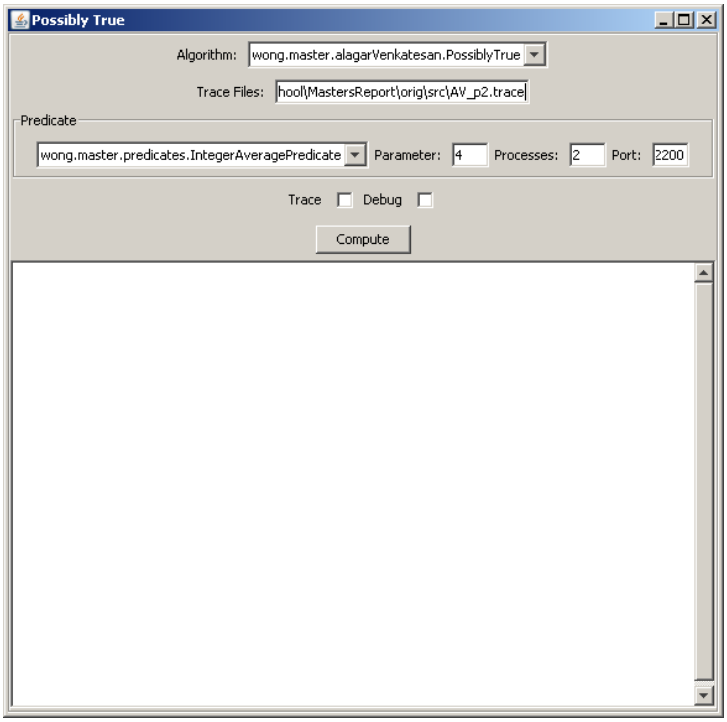



Figure 1: Sample Primary Client GUI

## ALGORITHM IMPLEMENTATIONS

All three algorithms share a common structure, having a constructor, a main function, and a compute function.

- The constructor takes a `GlobalStatePredicate` as an argument, and is implemented so the Graphical User Interface or other third-party clients can invoke the specific implementation programmatically. This particular predicate will be used for evaluation during execution.
- The main function has been modified so four arguments are required at the command line. The four arguments are as follows:
    - Primary host name/IP address – the IP address or host name of the primary system to connect to
    - Primary host name port – the port number the primary system is listening on
    - Port to listen on – the port number this client should listen on
    - Client number – the client number for this particular client. This should be a unique integral number between 1 to *n*, where *n* is the number of secondary clients.

The compute function is where each individual algorithm's logic resides, but they still share the common behavior that if one or more results were found, the result returned would be the first global state encountered satisfying the predicate. The transfer of the necessary information also happens at the entry of the compute function. The implementation of each algorithm is as follows:

### Alagar and Venkatesan

```
1.  begin
2.  i = 1;
3.  Let <k1,...,kn> = S
```

```
4.  while( i <= n ) do
5.    Begin
8.    G = chkForRspFrmClients()         // smallest returns CGS or null
9.    if(G != null)  report possiblyTrue(G)
10.   S'=<k1,...,kn>   //S', if consistent, is a successor of S
11.   if S' is a global state then
12.     if value(S) = max( value( pred( S' ) ) ) then
13.       freeClient = getFreeClient()  // see if any clients are free
14.       if(freeClient)                // client is free, send it
15.         send(S', freeClient)
16.       else                          // no free client, check here
17.         DFT( S' )
18.     i++
19.   end
20. end
```

Algorithm for Primary Client for Alagar and Venkatesan

```
1.  while(true)                 // Wait for input
2.    S = InputFromPrimary
3.    if(S = null or repeatedTimedOut)
4.      exit                    // null or repeated timeout = terminate
5.    if(!predicate(S))         // otherwise check it…
6.      send DFT(S')            // …and its childs if it fails predicate
7.    else                      // otherwise send it back to the primary
8.      send( S, primary);
```

Algorithm for Secondary Client(s) for Alagar and Venkatesan

Alagar and Venkatesan presented a parallel algorithm in their original paper. The implementation presented in this report follows closely to that. To satisfy the assumption of the algorithm presented all inputs such as the timestamps of events, predicate to apply, and value to test for are on all the systems either by transfer at the start of the execution as is done with the timestamps of events and values or assumed to be present as part of the library package as is with the predicate. At the start, the primary client tests the first global state. If the global state does not satisfy the predicate and has more than one global state to test in the next step, it will attempt to send a global state to each of the secondary clients until one has been sent to each client, as is done with lines (13) through (15), after which it will check the rest of the global states at the primary client until it is able to find another connection that is free. Once a primary client or secondary client has tested a global state, it will attempt to continue a depth first traversal of the lattice from that node. Any global states it finds and has tested the greatest predecessor for, it will check the predicate for or offload it to another client as is denoted by line (12). If it

comes across a global state where it did not test its greatest predecessor, then it ignores that global state and all global states reached from that particular global state. If the secondary client ever finds a global state satisfying the predicate it will respond to the primary client with the global state with the valid global state, where the primary client will inform all other secondary clients to cease execution. If the secondary client runs out of global states to check, it will inform the primary client that it found nothing and might be assigned another global state to check later on. All secondary clients will either terminate when it receives the termination notice from the primary client (either due to something being found or all global states being checked already) or if it stops receiving information from the primary client after a prolonged period of time. An example of the expected behavior assuming there are two secondary clients is provided in Figure 2.

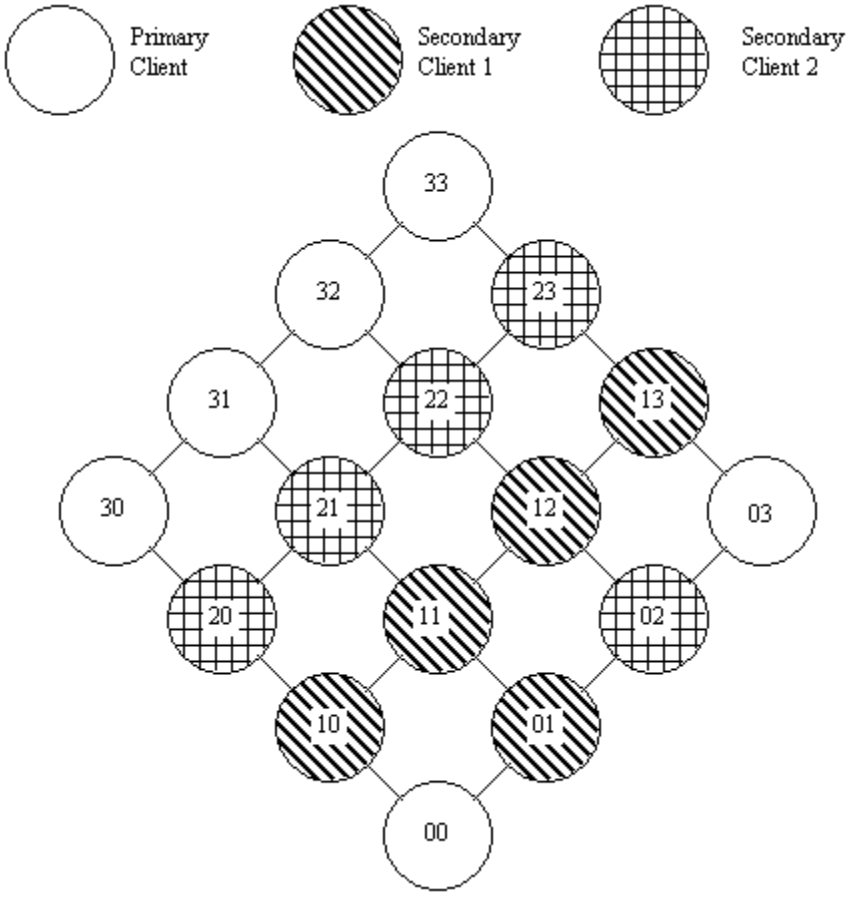



Figure 2: Expected Behavior With 2 Secondary Clients for Alagar And Venkatesan

## Cooper and Marzullo

```
1.  listOfStates     = arrayOfLists[numberOfSecondary]
2.  currListOfStates = initialGlobalState  // initial list of states
3.  listOfStates[0]  = currListOfStates
4.  level = 0
5.  while ( !Predicate(each state in listOfStates[0]) ) do
6.    begin
7.      G = chkForRspFrmClients()       // returns smallest CGS or null
8.      if(G != null)
9.        report possiblyTrue(G);
10.       exit
11.     last = current;
12.     current = getAllReachableStates( last )
13.     listOfStates[] = split(current,numOfSecondary) // split list
14.     for(int i = 1; I < numOfSecondary; i++)
15.       send(client i, listOfStates[i])              // send the list
16.   end
17. G = chkForRspFrmClients();
18. report possiblyTrue(G);
```

Algorithm for Primary Client for Cooper and Marzullo

```
1.  while(true)                            // wait for input
2.    S = InputFromPrimary
3.    if(S = null) exit                    // exit if null
4.    if(!Predicate(each state in list)) // check all the states…
5.      send(null, primary);               // …return null if none worked
6.    else                                 // otherwise send the state
7.      send(stateSatisfied, primary)
```

Algorithm for Secondary Client(s) for Cooper and Marzullo

Since Cooper and Marzullo's algorithm performed breath first traversal by enumerating all the possible global states at a specific level, the same idea is used in the parallel algorithm. At each level of the lattice the global states of that specific level is enumerated then divided approximately evenly into *p* sets as denoted by line (13), where *p* equals the number of total clients, and each client will get one of the sets to apply the predicates on as denoted by line (15). All clients will then either respond with the global state that they found to have satisfied the predicate or null if no such global state exist. If one or more global state satisfying the predicate does get reported back to the primary client, the smallest global state is reported back and all secondary clients are notified to terminate. If all secondary clients reports a null and the primary client does not find any global states satisfying the predicate either, then the next level of the lattice is enumerated

and the process repeats, until the lattice is exhausted. The primary client does not move on to the next level in the lattice until all the clients responds with either the success or failure of finding a global state satisfying the predicate. The secondary clients behave simply; applying the predicate to the list global states it receives from the primary and sending the result until it is time to terminate. An example of the expected behavior assuming two secondary clients is provided in Figure 3.

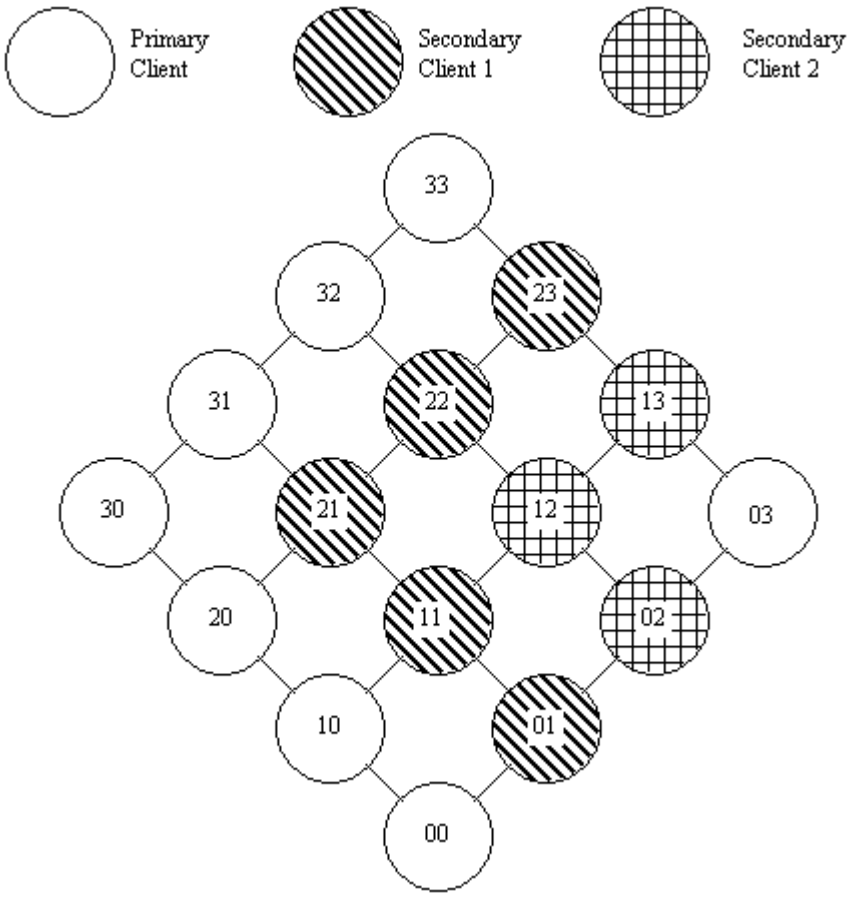



Figure 3: Expected Behavior With 2 Secondary Clients For Cooper and Marzullo

## Garg

```
1.  var
2.  G: array[0...n] of int initially for all i: G[i]=0; //current CGS
3.  K: array[0...n] of int // K=leastConsistent(succ(G,k))
4.  m: array[0...n] of int //m[i] equals the number of events at Pi
5.  n: numOfSecondary
6.  i: 0
```

```
7.  goodG: current CGS satisfying predicate
8.  while( G <= m ) do
9.    begin
10.   if(i % n != 0) send(G, client i % n,);
11.   else if( predicate( G ) ) then
12.     goodG = G;
13.   G = chkForRspFrmClients(goodG);   // returns smallest CGS or null
14.   if(G) then return G
15.   if (G == m ) then return null;
16.   i := 0                            // start over on the sends
17.   boolean found := false;
18.   int k:=m;
19.   while( !found ) do
20.     begin
21.     if(G[k] != m[k]) then           //next event on Pk exists
22.       e := next event on P[k] after G[k]
23.       boolean enabled := true;
24.       for j :=1 to n, j != k do
25.         if( e.v[j] > G[j] ) then
26.           enabled = false;
27.       found := enabled;
28.       if( !found ) k := k - 1;
29.     else k := k - 1;            //try the smaller process
30.     end
31.   G[k] := G[k + 1];             // advance on Pi
32.   for j := k + 1 to n do        // reset higher numbered processes
to 0
33.     G[j] := 0;
34.   K := G;                       // initialize K to G
35.   for i := 1 to n do            // compute K := leastConsistent(G)
36.     for j := 1 to n do
37.       K[j] = max( K[j], G[i].v[j] );
38.   G := K;
39.  end
```

Algorithm for Primary Client for Garg

```
1.  while(true)                           // wait for input
2.    S = InputFromPrimary
3.    if(S = null) exit                   // exit with null
4.    if(!Predicate(each state in list)) // check all the states…
5.      send(null, primary);              // …return null if none worked
6.    else
7.      send(stateSatisfied, primary)     // …otherwise send the state
```

Algorithm for Secondary Client(s) for Garg

To make Garg's algorithm perform in parallel a small modification from the serial behavior was made. The primary client performs the lexical traversal of the lattice, sending the next *n* global states to a secondary client before stopping to apply the predicate on the global states in the primary client as shown in lines (10) through (12) of

the pseudo code. Once the next *n* global states have been found and sent off, it processes the next few global states at the primary client, after which it checks the results from the secondary clients before continuing. If one or more client returns with a global state satisfying the predicate, the smallest of these global states is reported as the final result. If no client reports any predicate satisfying the predicate, then the primary client repeats the process by performing lexical traversal for the next *n* global states, either until a global state satisfying the predicate is found or the lattice is exhausted, reporting that no global state satisfies the predicate. To make use of all the secondary clients, *n* should be greater than the number of clients, otherwise some the secondary clients will not be utilized. In this particular implementation *n* was set to six but can easily be made into a user specified field. Once a client has finished checking all the global states it has received, or if it finds a global state satisfying the predicate, it will return to the result to the primary client, where the primary client will either find the next *n* global states if nothing was found yet, or terminate all connections to the secondary clients and report the smallest global state that has satisfied the client. This continues until either a global state satisfying the predicate is found or the lattice is exhausted. The primary client will not move on to generate the next *n* global state until all secondary clients have responded with their findings. The secondary client performs similarly to the one used in Cooper and Marzullo's algorithm, applying the predicate to the list of global states received from the primary client and returning the result until the primary signals that it is time to terminate. An example of the expected behavior assuming two secondary clients is provided in Figure 4.

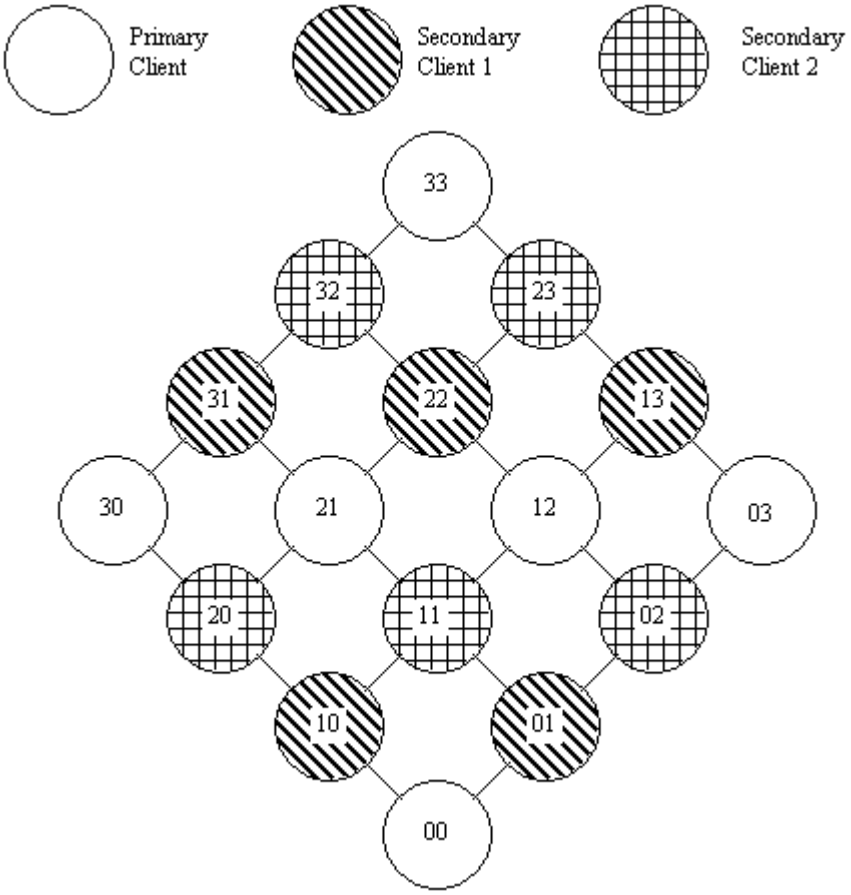



Figure 4: Expected Behavior With Two Secondary Clients For Garg

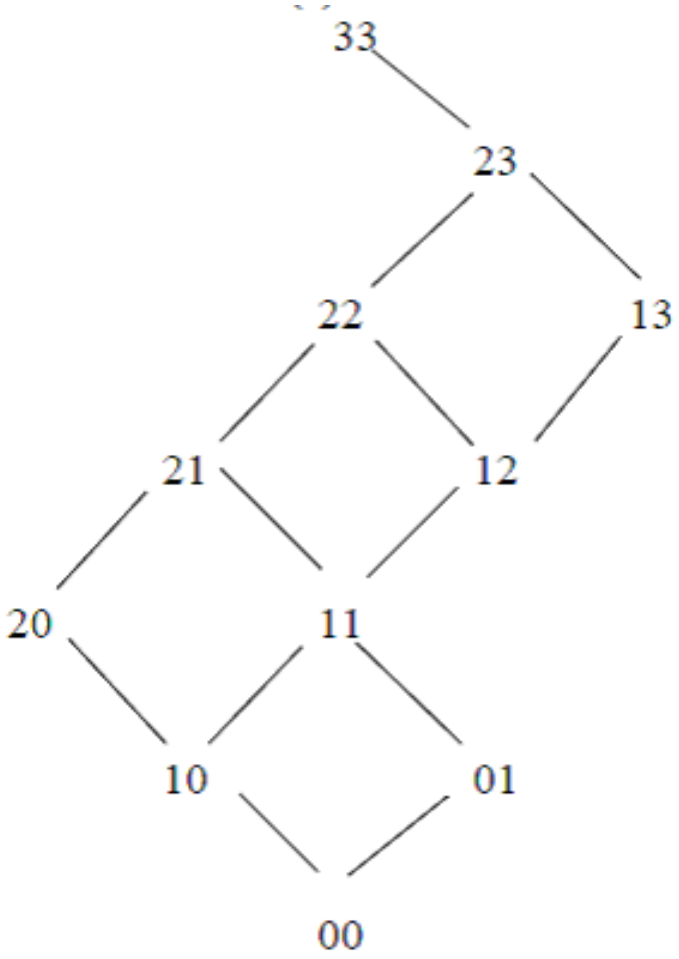
## EXPERIEMTAL RESULTS

The experiments were run on a single personal computer with an Intel Core i7-2600K CPU and 8 GB of memory. When performing the test in parallel the clients all reside on the same system while using different ports. The test cases used for all three algorithms had two processes each with four states in the input files as follows:

```
state=2;vectorClock=1,0
state=4;vectorClock=2,0
state=3;vectorClock=3,3
         Input1


state=2;vectorClock=0,1
state=4;vectorClock=1,2
state=3;vectorClock=1,3
         Input2
```

The expected traversal of each algorithm for the input is provided in Figure 5. For each of the algorithms three different test scenarios were used. In the first test case the first global state encountered will satisfy the predicate for all three traversal methods. In the second test the global state satisfying the predicate was in the middle of the lattice. The third test expected no global states to satisfy the predicate; hence the lattice must be fully traversed. The average results of the runs were summarized in Table 1, with a comparison to the serial version of the algorithms. The runtime for the serial run and the primary client was determined by taking the difference between the timestamp before the computation starts and after the computation is complete. The runtimes for the secondary clients are determined by the difference between the start and end of the application itself, taking into consideration the time necessary for the connection and input processing. In both cases the runtime displayed is in milliseconds. The memory usage for all the clients is determined by taking the difference between the total runtime memory versus the free runtime memory by the end of the execution, which coincided with the value displayed when using Java VisualVM[8]'s memory profiler. The unit here is bytes.




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

Figure 5: Expected Lattice Traversal For Input[3]

| | | Test1 | | Test2 | | Test3 | |
|---|---|---|---|---|---|---|---|
| | | Runtime | Memory Usage | Runtime | Memory Usage | Runtime | Memory Usage |
| AV | Secondary1 | 1629 | 1339696 | 1639 | 1339696 | 6457 | 1339600 |
| | Secondary2 | 1213 | 1339696 | 1639 | 1339696 | 6017 | 1339696 |
| | Primary | 2273 | 5359136 | 2144 | 6707536 | 5953 | 6028968 |
| | Total | 5115 | 8038528 | 5422 | 9386928 | 18427 | 8708264 |
| | Serial | 0 | 6029656 | 0 | 6028696 | 1 | 6699160 |
| CM | Secondary1 | 415 | 1339744 | 445 | 1339696 | 463 | 1339600 |
| | Secondary2 | 49 | 1339648 | 68 | 1339648 | 55 | 1339696 |
| | Primary | 968 | 6028936 | 1017 | 6028760 | 951 | 6045312 |
| | Total | 1432 | 8708328 | 1530 | 8708104 | 1469 | 8724608 |
| | Serial | 0 | 6031320 | 1 | 6028600 | 1 | 6028616 |
| Garg | Secondary1 | 410 | 1339600 | 364 | 1339696 | 1454 | 1339696 |
| | Secondary2 | 54 | 1339696 | 53 | 1339792 | 1054 | 1339648 |
| | Primary | 942 | 5376328 | 979 | 6037480 | 1889 | 6061624 |
| | Total | 1406 | 8055624 | 1396 | 8716968 | 4397 | 8740968 |
| | Serial | 0 | 6029048 | 0 | 6028672 | 0 | 6029184 |

Table 1: Experimental Results

From the results it appears that the parallel algorithm's performance is significantly slower and uses more memory than that of the serial algorithms. However this can be attributed to the fact that the sample test cases contained very small input, resulting in a small lattice to be traversed, and the overhead with starting the secondary clients, transmitting the files, and other communication overhead including user input delays before actual work is performed on the secondary clients led to the parallel algorithm seeming to have inferior performance compare to the serial algorithm. The runtime of the parallel version of Alagar and Venkatesan's algorithm suffered from this issue the most, since each client must wait until all other clients are complete before terminating, hence potentially just waiting for input when the lattice of consistent global states is just a linked list but still having the runtime be tripled in this case due to having three clients even though it is still just one client processing everything in serial fashion. Cooper and Marzullo's and Garg's algorithm still suffered from the overhead but did seem to perform better compare to Alagar and Venkatesan's algorithm. When looking at the memory

usage the statistics also appear to report the same result, that the parallel version will use more resources. This result should be expected, as some of the objects must be duplicated at each client.

## CONCLUSION

This paper presented a simple parallel implementation for Alagar and Venkatesan, Cooper and Marzullo, and Garg's algorithm for detecting possibly true consistent global states. Improvements on the algorithms could be made such as using selectors instead of waiting for input from other clients, performing compression on the input file before transmission to other clients, and failure recovery in the cases where one or more of the secondary clients cease to function normally. However it does provide a framework for potential future work.

# ADDENDUM: SOURCE CODE

## Package wong.master.alagarVenkatesan

```
/*
 * Created on Jul 9, 2003
 * Updated on May 23, 2012
 */
package wong.master.alagarVenkatesan;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.lang.reflect.Constructor;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.util.StringTokenizer;

import wong.master.common.AbstractPossiblyTrue;
import wong.master.common.TimeStamp;
import wong.master.predicates.GlobalStatePredicate;

/**
 * @author stephan
 * @author don wong
 * @version %I%, %G%
 */

// Primary server
public class PossiblyTrue extends AbstractPossiblyTrue {
//      inherited fields
//      protected ProcessEventSequence []     _eventSequences = null;
//      protected TimeStamp                                     _currentCGS = null;
  private static int ownID = 0;
  private static int localport = 0;
  private static Socket socket = null;
  private static ServerSocket listener = null;
  private static BufferedReader input = null;
  private static PrintWriter output = null;
  private static ServerSocketChannel schannel = null;
  private static SocketChannel channel = null;
  private static int checked = 0;

  public PossiblyTrue( GlobalStatePredicate gsp )
  {
    super( gsp );
  }

  public static void main ( String [] args )
  {
        Runtime rt = Runtime.getRuntime();
    long time = System.currentTimeMillis();
        long freeMem = rt.freeMemory();
```

```
    try {
      ownID = Integer.parseInt(args[3]);
      localport = Integer.parseInt(args[2]);
      int foreignport = Integer.parseInt(args[1]);
      listener = new ServerSocket(localport);
      socket = new Socket(args[0], foreignport);
      InetSocketAddress socketAddress = new InetSocketAddress(args[0], foreignport);
      input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
      output = new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));

      output.println(localport + ":" + ownID);
      output.flush();
      String additionalParams = input.readLine();
      StringTokenizer tok = new StringTokenizer(additionalParams);

      GlobalStatePredicate gsp = null;
      PossiblyTrue apt = null;
      try {
        Constructor     predicateConstructor     =     Class.forName(     tok.nextToken()
).getConstructors()[ 0 ];
        if ( predicateConstructor.getParameterTypes().length == 1 ) {
          gsp  =  (GlobalStatePredicate)predicateConstructor.newInstance(  new  Object  []  {
tok.nextToken()} );
        } else {
          gsp  =  (GlobalStatePredicate)predicateConstructor.newInstance(  new  Object  []  {}
);
        }

        String classname = tok.nextToken();
        Constructor algorithmConstructor = Class.forName( classname ).getConstructor( new
Class[] {GlobalStatePredicate.class} );

        apt = (PossiblyTrue)algorithmConstructor.newInstance( new Object [] { gsp} );
      } catch (Exception e) {
        e.printStackTrace();
        return;
      }

      int numberOfFiles = Integer.parseInt(input.readLine());
      String[] files = new String[numberOfFiles];
      for (int i = 0; i < numberOfFiles; i++) {
        File temp = File.createTempFile("trace"+i, ".trace");
        temp.deleteOnExit();
        files[i] = temp.getCanonicalPath();
        String tempLine = null;
        BufferedWriter out = new BufferedWriter(new FileWriter(temp));
        while (!(tempLine = input.readLine()).equals("+++")) {
          out.write(tempLine + "\n");

        }
        out.close();
      }

      String conns = input.readLine();

      // Calling APT to connect to everyone
      apt.connectAll(conns, apt, ownID, socket, input, output, listener);

      apt.compute( files, null, null, additionalParams);

      socket.setSoTimeout(0);
      System.out.flush();
      String info = null;
      TimeStamp result = null;
```

```
    int cycles = 0;
    boolean terminate = false;
    while (result == null) {
      for (int i = 0; i < apt.connections.length; i++) {
        try {
          apt.connections[i].setSoTimeout(1000);
          info = apt.istreams[i].readLine();
          if (info.equals("Closing")) {
            terminate = true;
          }
          apt.ostreams[0].println("working");
          apt.ostreams[0].flush();
          TimeStamp timestamp = new TimeStamp();
          tok = new StringTokenizer(info);
          int pid = Integer.parseInt(tok.nextToken());
          StringTokenizer tstok = new StringTokenizer(tok.nextToken(),",");
          int tokenCount = tstok.countTokens();
          for (i = 1; i <= tokenCount; i++) {
            timestamp.setClockValue(Integer.parseInt(tstok.nextToken()), i);
          }
          checked = 0;
          result = ((PossiblyTrue) apt).depthFirstTraversal(timestamp, pid, null);
        } catch (Exception e) {
        }
      }
      cycles++;
      if (terminate) {
        break;
      }
      if (cycles >= 3) {
        apt.ostreams[0].println("waiting");
        apt.ostreams[0].flush();
      }
      if (cycles >= 5) {
            break;
      }
    }
    if (result != null) {
      apt.ostreams[0].println(checked + "," + result.toString());
    } else {
      apt.ostreams[0].println(checked);
    }

    for (int i = 0; i < apt.connections.length; i++) {
      try {
        apt.ostreams[i].close();
        apt.istreams[i].close();
        apt.connections[i].close();
      } catch (Exception e) {
      }
    }

  } catch (IOException e) {
    e.printStackTrace();
  }
  time = System.currentTimeMillis() - time;
  freeMem = rt.totalMemory() - rt.freeMemory();  // assumes memory growth during

  System.out.println( "-----------------------------------------");
  System.out.println( "\n\tExecution Time:\t" + time + " ms" );
  System.out.println( "\tMemory Used:\t" + freeMem + " bytes" );
  System.out.println("Other method: " + Runtime.getRuntime().totalMemory());
}
```

```
  protected void compute()
  {
    int numProcesses = _eventSequences.length;
    TimeStamp result = depthFirstTraversal( initialGlobalState( numProcesses ), -1, this
);
    boolean nooneWorking = false;
    if (result != null) nooneWorking = true;
    while (!nooneWorking) {
      nooneWorking = true; // Assume no one working until proven otherwise
      for (int i = 0; i < connections.length; i++) {
        try {
          connections[i].setSoTimeout(2000);
          String input = istreams[i].readLine();
          if (input.equals("working")) {
            nooneWorking = false;
          } else if (input.contains(",")) {   // found timestamp that work, get out of
here
            TimeStamp p_currentCGS = new TimeStamp();
            StringTokenizer tstok = new StringTokenizer(input, ",");
            int tokenCount = tstok.countTokens() - 1; // First token is how many we
checked.
            int checkCount = Integer.parseInt(tstok.nextToken());
            super._numCGS += checkCount;
            for (i = 1; i <= tokenCount; i++) {
              p_currentCGS.setClockValue(Integer.parseInt(tstok.nextToken()), i);

              if (_currentCGS == null) {
                _currentCGS = p_currentCGS;
              }if (_currentCGS.greaterThan(p_currentCGS)) {
                _currentCGS = p_currentCGS;
              } else {
                for      (int      componentSize      =      1;      componentSize      <
p_currentCGS.getComponentCount(); componentSize++) {
                  if              (_currentCGS.getClockValue(componentSize)              >
p_currentCGS.getClockValue(componentSize)) {
                    _currentCGS = p_currentCGS;
                  }
                }
              }
            }
            break;
          } else {
            super._numCGS += Integer.parseInt(input); // just contains how many nodes we
checked.
          }
        } catch (Exception e) {
          e.printStackTrace();
          System.out.println(e);
        }
      }
    }
    // everyone done, let everyone know and terminate all connections
    for (int i = 0; i < connections.length; i++) {
      try {
        ostreams[i].println("Closing");
        ostreams[i].flush();
        connections[i].close();
        ostreams[i].close();
        istreams[i].close();
      } catch (Exception e) {
        e.printStackTrace();
      }
    }
    if (_currentCGS != null) {
```

```
      super._numCGS += checked;
    } else {
      super._numCGS++;
    }
  }

  /*
   * Algorithm DFT(S)
   * begin
   * (1)       i = 1;
   * (2)       Let <k1,...,kn> = S
   * (3)       while( i <= n ) do
   *                   begin
   * (4)                       S'=<k1,...,ki+1,...,kn> //S', if consistent, is a successor
of S
   * (5)                       if S' is a global state then
   * (6)                               if value(S) = max( value( pred( S' ) ) ) then
   * (7)                                       DFT( S' );
   * (8)                       i ++;
   *                   end;
   * end.
   */
  protected TimeStamp depthFirstTraversal( TimeStamp state, int pid, AbstractPossiblyTrue
apt )
  {
    if ( TRACE ) {
      println( "--> Entering DFT for " + state.toString() );
    }
    checked++;
    for (int i = 0; i < connections.length; i++) {
      try {
        connections[i].setSoTimeout(100);
        String info = istreams[i].readLine();
        if (info.equals("Closing")) {  // someone found something already, so quit
          return null;
        }
      } catch (Exception e) {
        System.out.println("Connections[" + i + "] had nothing to send, moving on");
      }
    }

    if ( !applyPredicate( state ) ) {
      TimeStamp nextState = null;
      pid=1;
      while ( pid <= state.size() ) { // number of nodes to check
        nextState = (TimeStamp)state.clone();
        nextState.setClockValue( nextState.getClockValue( pid ) + 1, pid);
        if ( TRACE ) {
          println( "\tnext state: " + nextState.toString() );
        }
        if ( isCGS( nextState ) ) {
          if ( TRACE ) {
            println( "\t\tis global state" );
          }
          if ( state.equals( getMaxPredecessor( nextState ) ) ) {
            if ((state.size() - pid) > 0) {
              int conn = getNextAvailable();
              if (conn != ownID && conn >= 0) {
                Socket socket = getConnection(conn);
                try {
                  PrintWriter         output         =         new         PrintWriter(new
OutputStreamWriter(socket.getOutputStream()));
                  output.println(pid + " " + nextState.toString());
                  output.flush();
```

```
              } catch (IOException e) {
                System.err.println(e);
              }
            } else {
              depthFirstTraversal( nextState, pid, apt );
            }
          } else {
            state = depthFirstTraversal( nextState, pid, apt );
            return state;
          }
        }
      } else {
        if ( TRACE ) {
          println( "\t\tnot a global state" );
        }
      }
      pid++;
    }
    ostreams[0].println(checked);
    return null;
  } else {
    if (pid != -1) {
      output.println(checked+","+state.toString());
    }
    _currentCGS = state;
    return state;
  }
}

private TimeStamp getMaxPredecessor( TimeStamp state )
{
  TimeStamp predecessor = (TimeStamp)state.clone();
  int currentValue = -1;
  for ( int pid = predecessor.size(); pid > 0; pid-- ) {
    currentValue = predecessor.getClockValue( pid );
    if ( currentValue > 0 ) {
      predecessor.setClockValue( currentValue - 1, pid );
      if ( isCGS( predecessor ) ) {
        break;
      } else {
        predecessor.setClockValue( currentValue, pid );
      }
    }
  }
  if ( DEBUG ) {
    println( "\t\tmax predecessor: " + predecessor.toString() );
  }
  return predecessor;
}
}
```

## Package wong.master.common

```
/*
 * Created on Jun 19, 2003
 * Updated on May 23, 2012
 */
package wong.master.common;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.net.InetAddress;
import java.net.ServerSocket;
import java.net.Socket;
import java.nio.ByteBuffer;
import java.nio.CharBuffer;
import java.nio.channels.SocketChannel;
import java.nio.charset.Charset;
import java.nio.charset.CharsetDecoder;
import java.nio.charset.CharsetEncoder;
import java.util.ArrayList;
import java.util.StringTokenizer;

import wong.master.predicates.GlobalStatePredicate;
import wong.master.util.ProcessTraceReader;

/**
 * @author stephan
 * @author don
 * @version %I%, %G%
 * Provide common functionality for PossiblyTrue algorithm implementations
 */
abstract public class AbstractPossiblyTrue {
  public static final String TRACE_SETTING = "wong.master.TRACE";
  public static final String DEBUG_SETTING = "wong.master.DEBUG";

  protected ProcessEventSequence []   _eventSequences = null;
  protected TimeStamp                 _currentCGS     = null;

  private GlobalStatePredicate _predicate = null;

  protected   boolean TRACE             = false;
  protected   boolean DEBUG             = false;
  protected   int   _numCGS             = 0;
  protected   Socket[] connections      = null; // array of connections
  protected   SocketChannel[] channels  = null; // array of socketChannels
  protected   int[]    available        = null; // buddy array to connections, to
determine how many nodes are queued up for a particular process
  protected   PrintWriter[] ostreams    = null; //O utput streams
  protected   BufferedReader[] istreams = null; // Input streams
  protected   static boolean terminate  = false;

  // setup
  protected Charset charset = Charset.forName("ISO-8859-1");
  protected CharsetDecoder decoder = charset.newDecoder();
  protected CharsetEncoder encoder = charset.newEncoder();

  // Allocate buffers
  protected ByteBuffer buffer = ByteBuffer.allocateDirect(1024);
  protected CharBuffer charBuffer = CharBuffer.allocate(1024);
```

```
  // to make global predicate configurable, use e.g. properties file as input, which
specifies predicate, or require
  // first argument to be predicate classname, etc.  For now concrete subclasses need to
specify what predicate to use.
  public AbstractPossiblyTrue( GlobalStatePredicate p )
  {
    _predicate = p;
  }

  public void compute(String[] args, String processes, String ports, String
additionalParam)
  {
    if ( args.length == 0 ) {
      println( "\nUsage:\t <VM> [-Dwong.master.{TRACE|DEBUG}={true|false}]
wong.master.{cooperMarzullo|alagarVenkatesan|garg}.PossiblyTrue processTraceFile1 ...
processTraceFileN\n\n" );
      return;
    }

    TRACE = Boolean.valueOf( System.getProperty( TRACE_SETTING ) ).booleanValue();
    DEBUG = Boolean.valueOf( System.getProperty( DEBUG_SETTING ) ).booleanValue();
    if ( DEBUG ) {
      TRACE = true;
    }

    if ( !processTraceFiles( args ) ) {
      return;
    }

    if (processes == null || ports == null) {
      return;
    }

    Runtime rt = Runtime.getRuntime();
    long time = System.currentTimeMillis();
    long freeMem = rt.freeMemory();

    int established = 0;
    int total = Integer.parseInt(processes);
    int port = Integer.parseInt(ports);

    connections = new Socket[total];
    channels = new SocketChannel[total];
    available = new int[total];
    ostreams = new PrintWriter[total];
    istreams = new BufferedReader[total];

    try {
      StringBuffer hostnports = new StringBuffer();
      ServerSocket ss = new ServerSocket(port + established);
      ss.setSoTimeout(5000);
      InetAddress addr = InetAddress.getLocalHost();

      // Get IP Address
      byte[] ipAddr = addr.getAddress();

      // Get hostname
      String hostname = addr.getHostName();
      while (established != total) {
        System.out.flush();

        connections[established] = ss.accept();//(new ServerSocket(port +
established)).accept();
```

```
        available[established] = 0;
        ostreams[established]= new PrintWriter(new
OutputStreamWriter(connections[established].getOutputStream()));
        istreams[established]= new BufferedReader(new
InputStreamReader(connections[established].getInputStream()));
        String portAndID = istreams[established].readLine();
        ostreams[established].println(additionalParam);
        ostreams[established].println(args.length); // send how many files
        for (int i =  0; i < args.length; i++) {
          BufferedReader inputBR = new BufferedReader(new FileReader(args[i]));
          String tempLine = null;
          while ((tempLine = inputBR.readLine()) != null) {
            ostreams[established].println(tempLine.trim());
            ostreams[established].flush();
          }
          ostreams[established].println("+++");
          ostreams[established].flush();
        }
        hostnports.append(" " +
(connections[established].getInetAddress()+"").substring(1) + ":" + portAndID);

        established++;
      }

      for (int i = 0; i < total; i++) {
        ostreams[i].println(total + " " + hostnports.toString());
        ostreams[i].flush();
      }
      compute();

      time = System.currentTimeMillis() - time;
      freeMem = rt.totalMemory() - rt.freeMemory();  // assumes memory growth during
      // CGS computation
      reportPossiblyTrue();

      for (int i = 0; i < connections.length; i++) {
        try {
          ostreams[i].close();
          istreams[i].close();
          connections[i].close();
        } catch (Exception e) {
          System.out.println("Problem closing something with " + i);
          System.out.println(e.toString());
        }
      }

      System.out.println( "-----------------------------------------");
      System.out.println( "\n\tExecution Time:\t" + time + " ms" );
      System.out.println( "\tMemory Used:\t" + freeMem + " bytes" );
      System.out.println("Other mememory: " + Runtime.getRuntime().totalMemory());
      ss.close();
    } catch (IOException e) {
      System.err.println("\n");
      e.printStackTrace();
    }
  }

  protected int getNextAvailable() {
    int ret = -1;
    int max = Integer.MAX_VALUE;
    if (available != null) {
      for (int i = 0; i < available.length; i++) {
        if (available[i] < max) ret = i;
      }
```

```
    }
    return ret;
  }

  protected Socket getConnection(int i) {
    return connections[i];
  }

  protected GlobalStatePredicate getPredicate ()
  {
    return _predicate;
  }

  private ArrayList getProcessStatesForGlobalState( TimeStamp globalState )
  {
    ArrayList processStates = new ArrayList();
    StringBuffer sb = new StringBuffer ();
    sb.append( "\tprocess states for global state " + globalState.toString() + ": " );
    sb.append( "[ ");
    for ( int pid = 1; pid <= globalState.size(); pid ++ ) {
      ProcessEvent e = _eventSequences[ pid - 1 ].getEventForID(
globalState.getClockValue( pid ) );
      if ( e == null ) {
        sb.append( "null, " );
        processStates.add( null );
      } else {
        sb.append( e.getState() + ", " );
        processStates.add( e.getState() );
      }
    }
    sb.delete( sb.length() - 2, sb.length() - 1 );
    sb.append( "]" );
    if ( DEBUG ) {
      println( sb.toString() );
    }
    return processStates;
  }

  protected TimeStamp initialGlobalState( int numProcesses )
  {
    TimeStamp state = new TimeStamp ();
    for ( int pid = 1; pid <= numProcesses; pid ++ ) {
      state.setClockValue( 0, pid );
    }
    return state;
  }


  protected void reportPossiblyTrue ()
  {
    StringBuffer sb = new StringBuffer ();
    sb.append( "\nnumber of CGS computed: " + _numCGS );
    if ( _currentCGS == null ) {
      sb.append( "\nNo global state satisfies " + getPredicate() + "\n" );
    } else {
      sb.append( "\nFound global state that satisfies " + getPredicate() + ":\n" );
      sb.append( "\t[ ");
      for ( int pid = 1; pid <= _currentCGS.size(); pid ++ ) {
        sb.append( _currentCGS.getClockValue( pid ) + ", " );
      }
      sb.delete( sb.length() - 2, sb.length() - 1 );
      sb.append( "]\n" );
    }
    println( sb.toString() );
```

```
  }

  protected boolean isCGS( TimeStamp state )
  {
    boolean isGlobalState = true;
    TimeStamp [] localStates = new TimeStamp [ state.size() ];
    ProcessEventSequence pes = null;
    // from each process event sequence, get the event where the clock value for that
process has the same
    // value as in the state argument for that process
    for ( int pid = 1; pid <= state.size(); pid++ ) {
      pes = _eventSequences[ pid - 1 ];
      if ( state.getClockValue( pid ) > pes.getMaxClockValueForProcess( pid ) ) {
        isGlobalState = false;
        break;
      }
      localStates[ pid - 1 ] = pes.getEventForClockValue( pid, state.getClockValue( pid )
).getTimeStamp();
    }
    if ( isGlobalState ) {
      for ( int pid = 1; pid <= state.size(); pid++ ) {
        for ( int localStateNdx = 0; localStateNdx < localStates.length; localStateNdx++
) {
          if ( localStateNdx == pid - 1 ) {
            continue;
          }
          if ( localStates[ localStateNdx ].getClockValue( pid ) > state.getClockValue(
pid ) ) {
            isGlobalState = false;
          }
        }
        if ( !isGlobalState ) {
          break;
        }
      }
    }

    return isGlobalState;
  }

  protected boolean applyPredicate( TimeStamp cgs )
  {
    if ( TRACE ) {
      println( "CGS: " + cgs.toString() );
    }
    _numCGS++;
    ArrayList processStates = getProcessStatesForGlobalState( cgs );
    return getPredicate().execute( processStates );
  }

  abstract protected void compute ();

  private boolean processTraceFiles( String [] args )
  {
    println( "" );
    _eventSequences = new ProcessEventSequence [ args.length ];
    ProcessTraceReader ptr = new ProcessTraceReader();
    int ndx = -1;
    try {
      for ( ndx = 0; ndx < args.length; ndx ++ ) {
        _eventSequences [ ndx ] = ptr.processFile( args [ ndx ], args.length );
      }
    } catch ( Exception e ) {
      println( "Exception while processing " + args [ ndx ] + ": " + e.getMessage() );
```

```
      println( "\tTrace file format: state=s;vectorClock=int1,int2,int3" );
      return false;
    }
    return true;
  }

  protected static void println( String s )
  {
    //System.out.println( s );
  }

  protected void connectAll(String conns, AbstractPossiblyTrue apt, int ownID, Socket
socket, BufferedReader input, PrintWriter output, ServerSocket listener) throws
IOException {
    StringTokenizer ips = new StringTokenizer(conns);
    apt.connections = new Socket[Integer.parseInt(ips.nextToken())];
    apt.available = new int[apt.connections.length];
    apt.ostreams = new PrintWriter[apt.connections.length];
    apt.istreams = new BufferedReader[apt.connections.length];
    int i;
    apt.connections[0] = socket;
    apt.available[0] = 0;
    apt.ostreams[0] = output;
    apt.istreams[0] = input;
    for (i = 1; i < apt.connections.length; i++) {
      StringTokenizer ipnport = new StringTokenizer(ips.nextToken(), ":"); // Parse out
as <ip>:<port>:<id>
      String ip =  ipnport.nextToken();
      int port = Integer.parseInt(ipnport.nextToken());
      int ID = Integer.parseInt(ipnport.nextToken());
      if (ID >= ownID) {  // we've connected to all smaller nodes, time to connect to
bigger nodes.
        break;
      }

      listener.setSoTimeout(0);
      apt.connections[i] = listener.accept();
      apt.available[i] = 0;
      apt.ostreams[i]= new PrintWriter(new
OutputStreamWriter(apt.connections[i].getOutputStream()));
      apt.istreams[i]= new BufferedReader(new
InputStreamReader(apt.connections[i].getInputStream()));

    }
    while (ips.hasMoreTokens() && i < apt.connections.length) {
      StringTokenizer ipnport = new StringTokenizer(ips.nextToken(), ":"); // Parse out
as <ip>:<port>:<id>
      String ip =  ipnport.nextToken();
      int port = Integer.parseInt(ipnport.nextToken());
      System.out.flush();
      apt.connections[i] = new Socket(ip, port);
      apt.ostreams[i]= new PrintWriter(new
OutputStreamWriter(apt.connections[i].getOutputStream()));
      apt.istreams[i]= new BufferedReader(new
InputStreamReader(apt.connections[i].getInputStream()));
    }
  }
}
```

## Package wong.master.cooperMarzullo

```
/*
 * Created on Jun 19, 2003
 * Updated on May 23, 2012
 */
package wong.master.cooperMarzullo;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.lang.reflect.Constructor;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.StringTokenizer;

import wong.master.common.AbstractPossiblyTrue;
import wong.master.common.ProcessEventSequence;
import wong.master.common.TimeStamp;
import wong.master.predicates.GlobalStatePredicate;


/**
 * @author stephan
 * @author don
 * @version %I%, %G%
 */
public class PossiblyTrue extends AbstractPossiblyTrue {
//      inherited fields
//      protected ProcessEventSequence []     _eventSequences = null;
//      protected TimeStamp                                   _currentCGS = null;
  private static int ownID = 0;
  private static int localport = 0;
  private static Socket socket = null;
  private static ServerSocket listener = null;
  private static BufferedReader input = null;
  private static PrintWriter output = null;

  private ArrayList _currentGlobalStates = null;

  public PossiblyTrue( GlobalStatePredicate gsp )
  {
    super( gsp );
  }

  public static void main ( String [] args )
  {
        Runtime rt = Runtime.getRuntime();
        long time = System.currentTimeMillis();
        long freeMem = rt.freeMemory();
    try {
      ownID = Integer.parseInt(args[3]);
      localport = Integer.parseInt(args[2]);
      int foreignport = Integer.parseInt(args[1]);
      socket = new Socket(args[0], foreignport);
      listener = new ServerSocket(localport);
      input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
```

```
      output = new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));
      output.println(localport + ":" + ownID);
      output.flush();
      String additionalParams = input.readLine();
      StringTokenizer tok = new StringTokenizer(additionalParams);

      GlobalStatePredicate gsp = null;
      PossiblyTrue apt = null;
      try {
        Constructor predicateConstructor = Class.forName( tok.nextToken()
).getConstructors()[ 0 ];
        if ( predicateConstructor.getParameterTypes().length == 1 ) {
          gsp = (GlobalStatePredicate)predicateConstructor.newInstance( new Object [] {
tok.nextToken()} );
        } else {
          gsp = (GlobalStatePredicate)predicateConstructor.newInstance( new Object [] {}
);
        }

        Constructor algorithmConstructor = Class.forName( tok.nextToken()
).getConstructor( new Class[] {GlobalStatePredicate.class} );

        apt = (PossiblyTrue)algorithmConstructor.newInstance( new Object [] { gsp} );
      } catch (Exception e) {
        return;
      }

      int numberOfFiles = Integer.parseInt(input.readLine());
      String[] files = new String[numberOfFiles];
      for (int i = 0; i < numberOfFiles; i++) {
        File temp = File.createTempFile("trace"+i, ".trace");
        temp.deleteOnExit();
        files[i] = temp.getCanonicalPath();
        String tempLine = null;
        BufferedWriter out = new BufferedWriter(new FileWriter(temp));
        while (!(tempLine = input.readLine()).equals("+++")) {
          out.write(tempLine + "\n");
        }
        out.close();
      }

      String conns = input.readLine();

      // Calling APT to connect to everyone
      apt.connectAll(conns, apt, ownID, socket, input, output, listener);
      apt.compute( files, null, null, additionalParams);

      while (true) {
        ArrayList list = new ArrayList();
        String input = apt.istreams[0].readLine();
        if (input.equals("closing")) {
          apt.ostreams[0].println(apt._numCGS);
          apt.ostreams[0].flush();
          break;
        }
        StringTokenizer inputTok = new StringTokenizer(input);
        while (inputTok.hasMoreElements()) {
          TimeStamp temp = new TimeStamp();
          StringTokenizer tstok = new StringTokenizer(inputTok.nextToken(), ",");
          int tokenCount = tstok.countTokens();
          for (int i = 1; i <= tokenCount; i++) {
            temp.setClockValue(Integer.parseInt(tstok.nextToken()), i);
          }
          list.add(temp);
```

```
      }

      apt.ostreams[0].println(apt.applyPredicate(list));
      apt.ostreams[0].flush();
    }

  } catch (Exception e) {
    e.printStackTrace();
  }
  time = System.currentTimeMillis() - time;
  freeMem = rt.totalMemory() - rt.freeMemory();  // assumes memory growth during


  System.out.println( "-------------------------------------------");
  System.out.println( "\n\tExecution Time:\t" + time + " ms" );
  System.out.println( "\tMemory Used:\t" + freeMem + " bytes" );
}

/*
 * current = initial global state
 * level = 0
 * while ( !Predicate(each current state) )
 * {
 *           last = current;
 *           current = getAllReachableStates( last );
 * }
 * report possiblyTrue;
 */
protected void compute ()
{
  int numProcesses = _eventSequences.length;
  _currentGlobalStates = new ArrayList();
  _currentGlobalStates.add( initialGlobalState( numProcesses ) );
  ArrayList last = new ArrayList();
  int level = 0;
  super._currentCGS = applyPredicate(_currentGlobalStates);
  //ArrayList arrayListOfList = new ArrayList();
  for (int i = 0; i < connections.length; i++) {
    try {
      connections[i].setSoTimeout(1000);
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  while ( null == super._currentCGS ) {
    last.clear();
    last.addAll( _currentGlobalStates );
    getAllReachableGlobalStates( last );

    if ( _currentGlobalStates.size() == 0 ) {
      if ( TRACE ) {
        println( "No more global states to construct." );
      }
      break;
    }

    StringBuffer[] cgsSb = new StringBuffer[connections.length+1];
    ArrayList tempAL = new ArrayList();
    for (int i = 0; i < _currentGlobalStates.size(); i++) {
      if (cgsSb[i % cgsSb.length] == null) {
        cgsSb[i % cgsSb.length] = new StringBuffer();
      }
```

```
        if ((i % cgsSb.length) == 0) {  // Don't need to send this, just make it into an
arrayList.
          tempAL.add(_currentGlobalStates.get(i));
        } else {
          cgsSb[i % cgsSb.length].append(((TimeStamp)
(_currentGlobalStates.get(i))).toString() + " ");
        }
      }
      for (int i = 0; i < connections.length; i++) {  // cgsSb[0] should be empty
        if (cgsSb[i+1] == null) continue;
        ostreams[i].println(cgsSb[i+1]);
        ostreams[i].flush();
      }
      super._currentCGS = applyPredicate( tempAL);
      for (int i = 0; i < connections.length; i++) {
        try {
          if (cgsSb[i+1] != null) {
            String input = istreams[i].readLine();
            if (!input.equals("null") && (super._currentCGS == null)) {
              super._currentCGS = new TimeStamp();
              StringTokenizer tstok = new StringTokenizer(input, ",");
              int tokenCount = tstok.countTokens();
              for (int j = 1; j <= tokenCount; j++) {
                super._currentCGS.setClockValue(Integer.parseInt(tstok.nextToken()), j);
              }
            }
          }
        } catch (java.net.SocketTimeoutException e) {
          // no input from i
        } catch (IOException e) {
          e.printStackTrace();
        }
      }
    }
    for (int i = 0; i < connections.length; i++) {
      try {
        ostreams[i].println("closing");
        ostreams[i].flush();
        String temp = istreams[i].readLine();
        if (!temp.equals("null")) _numCGS += Integer.parseInt(temp);
        ostreams[i].close();
        istreams[i].close();
        connections[i].close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }

  private void getAllReachableGlobalStates( ArrayList last )
  {
    _currentGlobalStates.clear();

    Iterator stateIt = last.iterator();
    TimeStamp lastState = null;
    TimeStamp nextState = null;
    while ( stateIt.hasNext() ) {
      lastState = (TimeStamp)stateIt.next();
      if ( DEBUG ) {
        println( "\t--> Processing last global state " + lastState.toString() );
      }
      for ( int pid = 1; pid <= lastState.size(); pid++ ) {
        nextState = (TimeStamp)lastState.clone();
        nextState.setClockValue( nextState.getClockValue( pid ) + 1, pid );
```

```
        if ( DEBUG ) {
          println( "\t\tchecking next state " + nextState.toString() );
        }
        if ( isCGS( nextState ) ) {
          addCurrentCGS( nextState );
        } else {
          if ( DEBUG ) {
            println( "\t\tNot a Consistent Global State" );
          }
        }
      }
    }

    // drop events where state < global state
    dropEventsWithClockValueLessThan( globalStatesMinClockValues () );
  }

  private void addCurrentCGS( TimeStamp cgs )
  {
    boolean alreadyAdded = false;
    Iterator it = _currentGlobalStates.iterator();
    TimeStamp ts = null;
    while ( it.hasNext() ) {
      ts = (TimeStamp)it.next();
      if ( ts.equals( cgs ) ) {
        alreadyAdded = true;
      }
    }
    if ( !alreadyAdded ) {
      if ( TRACE ) {
        println( "\t*** Adding global state " + cgs.toString() );
      }
      _currentGlobalStates.add( cgs );
    }
  }


  // remove all events whose process clock is less than minClockValue
  private void dropEventsWithClockValueLessThan( TimeStamp minClockValues )
  {
    if ( minClockValues != null ) {
      ProcessEventSequence pes = null;
      TimeStamp ts = null;
      int minClockValue = 0;
      for ( int pid = 1; pid <= minClockValues.size(); pid ++ ) {
        pes = _eventSequences[ pid - 1 ];
        minClockValue = minClockValues.getClockValue( pid );
        ts = pes.getFirstEvent().getTimeStamp();
        int eventMinClockValue = ts.getClockValue( pid );
        if ( eventMinClockValue < minClockValue ) {
          if ( DEBUG ) {
            println( "\t\tDropping local state " + ts.toString() + " from process " + pid
);
          }
          pes.popEvent();
        }
      }
    }
  }

  // get the lowest clock value from each process in the set of all current global states
  private TimeStamp globalStatesMinClockValues ()
  {
    TimeStamp globalState = null;
```

```
    TimeStamp minClockValues = null;
    int currentValue = -1;
    int minValue = -1;
    Iterator it = _currentGlobalStates.iterator();
    if ( it.hasNext() ) {
      minClockValues = (TimeStamp)( (TimeStamp)it.next() ).clone();
      while ( it.hasNext() ) {
        globalState = (TimeStamp)it.next();
        for ( int pid = 1; pid <= globalState.size(); pid ++ ) {
          currentValue = globalState.getClockValue( pid );
          minValue = minClockValues.getClockValue( pid );
          if ( currentValue < minValue ) {
            minClockValues.setClockValue( currentValue, pid );
          }
        }
      }
    }
    return minClockValues;
  }

  private TimeStamp applyPredicate( ArrayList globalStates )
  {
    //
    TimeStamp cgs = null;
    ArrayList processStates = null;
    Iterator it = globalStates.iterator();
    while ( it.hasNext() ) {
      cgs = (TimeStamp)it.next();
      if ( applyPredicate( cgs ) ) {
        break;
      }
      cgs = null;
    }
    return cgs;
  }

}
```

## Package wong.master.garg

```
/*
 * Created on Jul 10, 2003
 * Updated on May 23, 2012
 */
package wong.master.garg;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.lang.reflect.Constructor;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.StringTokenizer;

import wong.master.common.AbstractPossiblyTrue;
import wong.master.common.ProcessEvent;
import wong.master.common.TimeStamp;
import wong.master.predicates.GlobalStatePredicate;

/**
 * @author stephan
 * @author don
 * @version %I%, %G%
 */
public class PossiblyTrue extends AbstractPossiblyTrue {
//      inherited fields
//      protected ProcessEventSequence []     _eventSequences = null;
//      protected TimeStamp                                   _currentCGS = null;

  private static int ownID = 0;
  private static int localport = 0;
  private static Socket socket = null;
  private static ServerSocket listener = null;
  private static BufferedReader input = null;
  private static PrintWriter output = null;
  private static int sent = 0 ;
  private static int checked = 0;

  public PossiblyTrue( GlobalStatePredicate gsp )
  {
    super( gsp );
  }


  public static void main ( String [] args )
  {
        Runtime rt = Runtime.getRuntime();
        long time = System.currentTimeMillis();
        long freeMem = rt.freeMemory();
    try {
      ownID = Integer.parseInt(args[3]);
      localport = Integer.parseInt(args[2]);
      int foreignport = Integer.parseInt(args[1]);
      socket = new Socket(args[0], foreignport);
      listener = new ServerSocket(localport);
      input = new BufferedReader(new InputStreamReader(socket.getInputStream()));
      output = new PrintWriter(new OutputStreamWriter(socket.getOutputStream()));
```

```
      output.println(localport + ":" + ownID);
      output.flush();
      String additionalParams = input.readLine();
      StringTokenizer tok = new StringTokenizer(additionalParams);

      GlobalStatePredicate gsp = null;
      PossiblyTrue apt = null;
      try {
        Constructor predicateConstructor = Class.forName( tok.nextToken()
).getConstructors()[ 0 ];
        if ( predicateConstructor.getParameterTypes().length == 1 ) {
          gsp = (GlobalStatePredicate)predicateConstructor.newInstance( new Object [] {
tok.nextToken()} );
        } else {
          gsp = (GlobalStatePredicate)predicateConstructor.newInstance( new Object [] {}
);
        }

        Constructor algorithmConstructor = Class.forName( tok.nextToken()
).getConstructor( new Class[] {GlobalStatePredicate.class} );

        apt = (PossiblyTrue)algorithmConstructor.newInstance( new Object [] { gsp} );
      } catch (Exception e) {
        return;
      }

      int numberOfFiles = Integer.parseInt(input.readLine());
      String[] files = new String[numberOfFiles];
      for (int i = 0; i < numberOfFiles; i++) {
        File temp = File.createTempFile("trace"+i, ".trace");
        temp.deleteOnExit();
        files[i] = temp.getCanonicalPath();
        String tempLine = null;
        BufferedWriter out = new BufferedWriter(new FileWriter(temp));
        while (!(tempLine = input.readLine()).equals("+++")) {
          out.write(tempLine + "\n");
        }
        out.close();
      }

      String conns = input.readLine();

      // Calling APT to connect to everyone
      apt.connectAll(conns, apt, ownID, socket, input, output, listener);
      apt.compute( files, null, null, additionalParams);
      socket.setSoTimeout(0);
      while (true) {
        ArrayList list = new ArrayList();
        String input = apt.istreams[0].readLine();
        if (input == null || input.equals("closing")) {
          apt.ostreams[0].println(apt._numCGS);
          apt.ostreams[0].flush();
          break;
        }
        StringTokenizer inputTok = new StringTokenizer(input);
        while (inputTok.hasMoreElements()) {
          TimeStamp temp = new TimeStamp();
          StringTokenizer tstok = new StringTokenizer(inputTok.nextToken(), ",");
          int tokenCount = tstok.countTokens();
          for (int i = 1; i <= tokenCount; i++) {
            temp.setClockValue(Integer.parseInt(tstok.nextToken()), i);
          }
          checked++;
          if (apt.applyPredicate(temp)) {
```

```
          apt.ostreams[0].println(temp.toString());
        } else {
          apt.ostreams[0].println(false);
        }
        apt.ostreams[0].flush();
      }
    }

  } catch (java.net.SocketException e) {
  } catch (Exception e) {
    e.printStackTrace();
  }
  time = System.currentTimeMillis() - time;
  freeMem = rt.totalMemory() - rt.freeMemory();  // assumes memory growth during


  System.out.println( "-------------------------------------------");
  System.out.println( "\n\tExecution Time:\t" + time + " ms" );
  System.out.println( "\tMemory Used:\t" + freeMem + " bytes" );
}

/*
 * var
 *    G: array[0...n] of integer initially for all i: G[i]=0; //current CGS
 *    K: array[0...n] of integer; // K=leastConsistent(succ(G,k))
 *    m: array[0...n] of integer //m[i] equals the number of events at Pi
 *
 * while( G <= m )
 * {
 *           if( B( G ) ) then return G;
 *           if (G == m ) then return null;
 *           boolean found := false;
 *           int k:=m;
 *           while( !found )
 *           {
 *                  if(G[k] != m[k]) then //next event on Pk exists
 *                  {
 *                         e := next event on P[k] after G[k]
 *                         boolean enabled := true;
 *                         for j :=1 to n, j != k do
 *                         {
 *                                if( e.v[j] > G[j] ) then
 *                                       enabled = false;
 *                         }
 *                         found := enabled;
 *                         if( !found ) k := k - 1;
 *                  }
 *                  else k := k - 1; //try the smaller process
 *           }
 *           G[k] := G[k + 1]; // advance on Pi
 *           for j := k + 1 to n do // reset higher numbered processes to 0
 *                  G[j] := 0;
 *
 *           K := G; // initialize K to G
 *           for i := 1 to n do // compute K := leastConsistent(G)
 *                  for j := 1 to n do
 *                         K[j] = max( K[j], G[i].v[j] );
 *           G := K;
 * }
 */
protected void compute()
{
  _currentCGS = initialGlobalState( _eventSequences.length );
  TimeStamp leastCGS = null;
```

```
    TimeStamp maxEvents = getMaxEvents();
    ProcessEvent e = null;
    int numProcesses = _eventSequences.length;

    while ( !_currentCGS.greaterThan( maxEvents ) ) {
      int k = numProcesses;
      boolean found = false;
      if ( TRACE ) {
        println( "--> current CGS: " + _currentCGS.toString() );
      }

      if (sent >= 4 && sent < 5) {
        if ( applyPredicate( _currentCGS) ) {
          checkForValues();
          break;
        }
        ++sent;
      } else {
        if (sent >= 5) {
          found = checkForValues();
          sent = 0;
        }
        int output = sent/2;
        ostreams[output].println(_currentCGS.toString());
        ostreams[output].flush();
        ++sent;
      }
      if (found) break;  // break out since one of the other threads found correct
result.
      if ( _currentCGS.equals( maxEvents ) ) {
        if ( TRACE ) {
          println( "\t\tcurrent CGS equals maxEvents" );
        }
        _currentCGS = null;
        break;
      }
      while ( !found ) {
        if ( _currentCGS.getClockValue( k ) != maxEvents.getClockValue( k ) ) {
          e = _eventSequences[ k - 1 ].getEventForClockValue( k,
_currentCGS.getClockValue( k ) + 1 );
          if ( DEBUG ) {
            println( "\t\tevaluating event " + e.getTimeStamp().toString() + " from
process " + k );
          }
          boolean enabled = true;
          for ( int pid = 1; pid <= numProcesses; pid++ ) {
            if ( pid == k ) {
              continue;
            }
            if ( e.getTimeStamp().getClockValue( pid ) > _currentCGS.getClockValue( pid )
) {
              if ( DEBUG ) {
                println( "currentCGS: " + _currentCGS.toString() + ", pid: " + pid );
                println( "\t\tevent clock for process " + pid + ": " +
e.getTimeStamp().getClockValue( pid ) );
                println( "\t\tcurrent CGS event clock for process " + pid + ": " +
_currentCGS.getClockValue( pid ) );
                println( "\t\t--> event not enabled" );
              }
              enabled = false;
              break;
            }
          }
          found = enabled;
```

```
          if ( !found ) {
            k--;
          } else {
            if ( DEBUG ) {
              println( "\t\tevent is enabled" );
            }
          }
        } else {
          k--;
        }
      }

      _currentCGS.setClockValue( _currentCGS.getClockValue( k ) + 1, k );
      for ( int pid = k + 1; pid <= numProcesses; pid++ ) {
        _currentCGS.setClockValue( 0, pid );
      }
      if ( DEBUG ) {
        println( "\t\tpossible CGS: " + _currentCGS.toString() );
      }

      leastCGS = _currentCGS;
      int leastCGSVal = -1;
      int currentCGSVal = -1;
      for ( int i = 1; i <= numProcesses; i++ ) {
        for ( int j = 1; j <= numProcesses; j++ ) {
          leastCGSVal = leastCGS.getClockValue( j );
          currentCGSVal = _eventSequences [ i - 1 ].getEventAt(
_currentCGS.getClockValue( i ) ).getTimeStamp().getClockValue( j );
          leastCGS.setClockValue( leastCGSVal >= currentCGSVal ? leastCGSVal :
currentCGSVal, j );
        }
      }
      if ( DEBUG ) {
        println( "\t\tleast CGS: " + leastCGS.toString() );
      }
      _currentCGS = leastCGS;
    }
    if (_currentCGS == null) checkForValues(); // check the other threads one mroe time

  }


  private TimeStamp getMaxEvents()
  {
    TimeStamp maxEvents = new TimeStamp();
    TimeStamp ts = null;
    int pid = 0;
    for ( int ndx = 0; ndx < _eventSequences.length; ndx ++ ) {
      ts = _eventSequences[ ndx ].getLastEvent().getTimeStamp();
      pid = ndx + 1;
      maxEvents.setClockValue( ts.getClockValue( pid ), pid );
    }
    return maxEvents;
  }

  private boolean checkForValues() {
    boolean found = false;
    for (int i = 0; i < 2 && !found; i++) {
      try {
        connections[i].setSoTimeout(1000);
        String streamInput = null;
        for (int j = 0; j < 2; j++) {  // j is the number of things we sent, read that
many response back.
          streamInput = istreams[i].readLine();
```

```
          _numCGS++;
          if (!streamInput.equals("false")) {
            StringTokenizer inputTok = new StringTokenizer(streamInput);
            TimeStamp temp = new TimeStamp();
            StringTokenizer tstok = new StringTokenizer(inputTok.nextToken(), ",");
            int tokenCount = tstok.countTokens();
            for (int x = 1; x <= tokenCount; x++) {
              temp.setClockValue(Integer.parseInt(tstok.nextToken()), x);
            }
            _currentCGS = temp;
            found = true;
            break;
          }

        }
      } catch (Exception exc) {
      }
    }
    return found;
  }
}
```

## Package wong.master.predicates

## Package wong.master.util

## Package wong.master.util.generator

## Package wong.master

```
/*
 * Created on Jul 16, 2003
 * Updated on May 23, 2012
 */
package wong.master;

import wong.master.predicates.GlobalStatePredicate;
import wong.master.util.Properties;
import wong.master.common.AbstractPossiblyTrue;
import java.awt.*;
import java.awt.event.*;
import java.io.IOException;
import java.io.PipedInputStream;
import java.io.PipedOutputStream;
import java.io.PrintStream;
import java.lang.reflect.Constructor;
import javax.swing.event.*;
import javax.swing.*;
import java.util.StringTokenizer;

/**
 * @author stephan
 * @author don wong
 * @version %I%, %G%
 */
public class PossiblyTrue {
  private static final String PROPERTIES  = "wong/master/PossiblyTrue.properties";

  private Properties _props               = null;
  private JLabel     _algorithmsLabel     = null;
  private JComboBox  _algorithms          = null;
  private JLabel     _traceFilesLabel     = null;
  private JTextField _traceFiles          = null;
  private JLabel     _predicatesLabel     = null;
  private JComboBox  _predicates          = null;
  private JLabel     _predicateParamLabel = null;
  private JTextField _predicateParam      = null;
  private JLabel     _numberOfProcsLabel  = null;
  private JTextField _numberOfProcsParam  = null;
  private JLabel     _portLabel           = null;
  private JTextField _portParam           = null;
  private JLabel     _debugLabel          = null;
  private JCheckBox  _debug               = null;
  private JLabel     _traceLabel          = null;
  private JCheckBox  _trace               = null;
  private JButton    _compute             = null;
  private JTextField _stdout              = null;
  private JLabel     _status              = null;

  private PipedInputStream  _piOut        = null;
  private PipedInputStream  _piErr        = null;
  private PipedOutputStream   _poOut      = null;
  private PipedOutputStream _poErr        = null;
  private JTextArea       _console        = null;



  public static void main(String[] args)
  {
    PossiblyTrue gui = new PossiblyTrue();
    gui.run();
  }
```

```
private void run()
{
  JFrame frame = buildUI();
  frame.setVisible(true);
}

private JFrame buildUI()
{
  try {
    UIManager.setLookAndFeel( UIManager.getSystemLookAndFeelClassName() );
  } catch (Exception e) {
    e.printStackTrace();
  }

  JFrame frame = new JFrame ( "Possibly True" );
  JPanel pane = new JPanel();
  pane.setBorder( BorderFactory.createEtchedBorder() );
  pane.setLayout( new BoxLayout( pane, BoxLayout.Y_AXIS ) );

  _algorithmsLabel = new JLabel( "Algorithm: " );
  _algorithms = new JComboBox();
  _algorithms.addItem( makeObj( "wong.master.alagarVenkatesan.PossiblyTrue" ) );
  _algorithms.addItem( makeObj( "wong.master.cooperMarzullo.PossiblyTrue" ) );
  _algorithms.addItem( makeObj( "wong.master.garg.PossiblyTrue" ) );

  _traceFilesLabel = new JLabel ( "Trace Files: " );
  _traceFiles = new JTextField( 25 );

  _predicatesLabel = new JLabel( "Predicate: " );
  _predicates = new JComboBox();
  _predicates.addItem( makeObj( "wong.master.predicates.IntegerAveragePredicate" ) );
  _predicates.addItem( makeObj( "wong.master.predicates.FalsePredicate" ) );
  _predicateParamLabel = new JLabel( "Parameter: " );
  _predicateParam = new JTextField( 3 );
  _numberOfProcsLabel = new JLabel( "Processes: " );
  _numberOfProcsParam = new JTextField( 3 );
  _portLabel = new JLabel( "Port: " );
  _portParam = new JTextField( 3 );

  // Set up System.out
  try {
    _piOut = new PipedInputStream();
    _poOut = new PipedOutputStream(_piOut);
    System.setOut(new PrintStream(_poOut, true));

    // Set up System.err
    _piErr = new PipedInputStream();
    _poErr = new PipedOutputStream(_piErr);
    System.setErr(new PrintStream(_poErr, true));
  } catch ( IOException ioe ) {
  }

  // Add a scrolling text area
  _console = new JTextArea();
  _console.setEditable(false);
  _console.setRows(20);
  _console.setColumns(50);

  // Create reader threads
  new ReaderThread(_piOut).start();
  new ReaderThread(_piErr).start();
```

```
_traceLabel = new JLabel( "Trace " );
_trace = new JCheckBox();
_debugLabel = new JLabel( "Debug " );
_debug = new JCheckBox();
_debug.addChangeListener( new ChangeListener ()
                          {
                            public void stateChanged( ChangeEvent e )
                            {
                              if ( _debug.isSelected() ) {
                                _trace.setSelected( true );
                                _trace.setEnabled( false );
                              } else {
                                _trace.setSelected( false );
                                _trace.setEnabled( true );
                              }
                            }
                          }
                        );
_compute = new JButton( "Compute" );
_compute.addActionListener( new ActionListener ()
                            {
                              public void actionPerformed ( ActionEvent e )
                              {
                                compute();
                              }
                            }
                          );

JPanel algorithmsPanel = new JPanel();
algorithmsPanel.add( _algorithmsLabel );
algorithmsPanel.add( _algorithms );

JPanel traceFilesPanel = new JPanel();
traceFilesPanel.add( _traceFilesLabel );
traceFilesPanel.add( _traceFiles );

JPanel predicatesPanel = new JPanel();
predicatesPanel.add( _predicates );
predicatesPanel.add( _predicateParamLabel );
predicatesPanel.add( _predicateParam );
predicatesPanel.add( _numberOfProcsLabel );
predicatesPanel.add( _numberOfProcsParam );
predicatesPanel.add( _portLabel );
predicatesPanel.add( _portParam );
predicatesPanel.setBorder( BorderFactory.createTitledBorder( "Predicate") );

JPanel settingsPanel = new JPanel();
settingsPanel.add( _traceLabel );
settingsPanel.add( _trace );
settingsPanel.add( _debugLabel );
settingsPanel.add( _debug );

JPanel buttonPanel = new JPanel();
buttonPanel.add( _compute );

pane.add( algorithmsPanel );
pane.add( traceFilesPanel );
pane.add( predicatesPanel );
pane.add( settingsPanel );
pane.add( buttonPanel );
pane.add(new JScrollPane(_console), BorderLayout.CENTER);

pane.setPreferredSize( new Dimension ( 575, 550 ) );
frame.getContentPane().add( pane, BorderLayout.CENTER );
```

```
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.pack();
    return frame;
  }

  private Object makeObj( final String item )
  {
    return new Object(){ public String toString(){ return item;}};
  }

  private void compute ()
  {
    _console.setText( null );

    String algorithm = _algorithms.getSelectedItem().toString();
    String predicate = _predicates.getSelectedItem().toString();
    String parameter = _predicateParam.getText();
    String processes = _numberOfProcsParam.getText();
    String ports = _portParam.getText();
    System.setProperty( "wong.master.TRACE", _trace.isSelected() ? "true" : "false" );
    System.setProperty( "wong.master.DEBUG", _debug.isSelected() ? "true" : "false" );
    if (processes.equals("")) processes = "2";
    if (ports.equals("")) ports="2200";

    try {
      GlobalStatePredicate gsp = null;
      Constructor predicateConstructor = Class.forName( predicate ).getConstructors()[ 0
];
      if ( predicateConstructor.getParameterTypes().length == 1 ) {
        gsp = (GlobalStatePredicate)predicateConstructor.newInstance( new Object [] {
parameter} );
      } else {
        gsp = (GlobalStatePredicate)predicateConstructor.newInstance( new Object [] {} );
      }

      Constructor algorithmConstructor = Class.forName( algorithm ).getConstructor( new
Class[] {GlobalStatePredicate.class} );

      AbstractPossiblyTrue apt = (AbstractPossiblyTrue)algorithmConstructor.newInstance(
new Object [] { gsp} );

      StringTokenizer traceFiles = new StringTokenizer( _traceFiles.getText(), " " );
      int numArgs = traceFiles.countTokens();

      String [] args = new String [ numArgs ];
      int ndx = 0;

      while ( traceFiles.hasMoreTokens() ) {
        args[ ndx ] = traceFiles.nextToken();
        ndx++;
      }

      apt.compute( args, processes, ports, predicate + " " + parameter + " " + algorithm
);

    } catch ( Exception e ) {
      System.err.println( e.getMessage() );
      e.printStackTrace( System.err );
    }
  }

  class ReaderThread extends Thread {
    PipedInputStream pi;
```

```
    ReaderThread(PipedInputStream pi)
    {
      this.pi = pi;
    }

    byte[] buf = new byte[2056];
    int len = 0;

    public void run()
    {
      try {
        while (true) {
          len = pi.read(buf);
          if (len == -1) {
            break;
          }
          _console.append( new String( buf, 0, len ) );
          _console.setCaretPosition(_console.getDocument().getLength());
        }
      } catch (IOException e) {
      }
    }
  }
}
```

## REFERENCES


1. 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.
2. R. Cooper and K. Marzullo. Consistent Detection of Global Predicates. In Proc. of the Workshop on Parallel and Distributed Debugging, pages 163-173, Santa Cruz, CA, May 1991.
3. V. Garg. Enumerating Global States of a Distributed Computation. In International Conference on Parallel and Distributed Computing Systems, 2003
4. V. Garg. Elements of Distributed Computing, Wiley and Sons, New York, 2002.
5. S. Lips. Global Predicate Detection Algorithms. August 2003.
6. The Java Programming Language,

   http://www.oracle.com/technetwork/java/javase/downloads/index.html
7. Eclipse Integrated Development Environment, http://www.eclipse.org/
8. Java VisualVM, http://visualvm.java.net/