Communicating Process Architectures 2002
James Pascoe, Peter Welch, Roger Loader and Vaidy Sunderam (Eds.)
IOS Press, 2002

Implementing a Distributed Algorithm for Detection of Local Knots and Cycles in Directed Graphs

Geraldo Pereira DE SOUZA and Gerson Henrique PFITSCHER
Department of Computer Science, University of Brasilia, Brazil
geraldo@acm.org, gerson@unb.br

Abstract. In general, most deadlocks take form of cycles (in database systems) and knots (in communication systems). Boukerche and Tropper have proposed a distributed algorithm to detect cycles and knots in generic graphs. Their algorithm has a message complexity of 2*m* vs. (at least) 4*m* for the Chandy and Misra algorithm, where *m* is the number of links in the graph, and requires *O* (*n log n*) bits of memory, where *n* is the number of nodes. We have implemented Boukerche´s algorithm. Our implementation of the algorithm is based on the construction of processes of the CSP model. The implementation was done using JCSP, an implementation of CSP for Java.

1 Introduction

Parallel simulations have received special attention and a great amount of resources for research due to the increasing demand for systems with good performance, including fault tolerance and search for more efficient algorithms to deal with issues related to communication and to handle problems like deadlocks. There is also the objective of taking advantage of resources available on the networks and on the Internet.

The detection of cycles and knots in directed graphs is an area of research in parallel simulations that has deserved many published works and proposed algorithms. In [1] Boukerche makes an analysis and presents the results of his search for an algorithm that solved the problem in a more efficient way. In his research Boukerche noticed that the proposed solutions handled cycles and knots independently, that is, there was not an algorithm that could detect in a single evocation if a given node was in a cycle or in a knot. Boukerche then proposed his algorithm for detection of cycles and knots in a single evocation. The utilization of the algorithm is important when cycles and knots can occur with a high frequency, when the size of the graph is big, and when it is necessary to know if a given node of the graph is in a cycle or in a knot.

According to Boukerche [1], most of the deadlocks take the form of cycles (in database systems) and knots (in communication systems). His algorithm has advantages over others such as the one by Chandy and Misra [20], requiring less communication: 2*m* against at least 4*m*, where *m* is the number of links in the graph. The algorithm needs *O* (*n log n*) bits of memory, where *n* is the number of nodes.

In a search for implementations of the algorithm and through contact with Boukerche (by e-mail) we were informed that he did not know of any completed implementation of his algorithm. Since the algorithm is inside our line of research, distributed computation, we

decided to apply part of the focus of our work, the development of distributed systems based on the CSP model, to solve the problem.

CSP, *Communicating Sequential Processes*, is a process algebra that permits the specification and formal modeling of concurrent applications based on the construction of processes that synchronously communicate through channels [6]. Languages such as Occam and ADA are based on CSP. There are also CASE tools such as Probe and FDR [10] that help in the verification of CSP models. Huang presented the use of CSP to solve problems related to graph theory by introducing an algorithm for detection of deadlocks using the communication principles of CSP [15].

Lately Java is being adopted for distributed and concurrent programming due to characteristics like platform independence, object orientation, safety and ease of use. In previous analysis we had already verified the viability of using Java for the development of efficient applications for distributed environments. In [12] we presented the implementation and execution of Java applications in Beowulf and heterogeneous cluster environments.

Two projects that support CSP for Java are the JCSP (*Communicating Sequential Processes for Java*) [7, 8, 9] and the CTJ (*Communicating Threads for Java*) [6, 16]. Between these two tools that initially satisfied our objectives, we chose JCSP because its library was more appropriate for the programming of the algorithm. The JCSP library presents CSP constructions for the Java language and supports parallel and sequential composition of processes. CTJ, the other library analyzed, is nowadays being oriented towards the development of real time A/D and D/A applications [6, 16].

The work is organized in the following way: In section 2 we introduce the CSP model; In 3 we introduce the JCSP library; In 4 we present Boukerche's distributed algorithm for detection of cycles and knots in directed graphs; In 5 we present the modeling of the algorithm using CSP notation; In 6 we present our implementation of the algorithm using the JCSP library; In 7 a simulation with `Dck` is presented and finally, in 7 we present our conclusions and suggestions for future work.

2 CSP Model

CSP was developed by Hoare to serve as a mathematical formalism capable of representing the functioning of a system through the composition of processes [2, 3]. It is based on two distinct types of basic objects: events and processes. In the CSP model processes communicate sequentially and through channels. Roscoe [5] and Schneider [4] included new functionalities to the CSP so that the model could handle new requirements like aspects of time (existent in an extension of CSP: TCSP[1]).

Table 1: summary of the main CSP constructions

Construction	Description
$P\ ;\ Q$	Sequential execution of *P* and *Q*
$a \rightarrow P$	Engage in event a, then become process P
$P\ {}_{A}\|_{B}\ Q$	Parallel execution of *P* and *Q* with synchronization
$P \mid\mid\mid Q$	Parallel execution of *P* and *Q* without synchronization
$P\ []\ Q$	P external choice Q
$P \sqcap Q$	P internal choice Q
$\mu X \bullet P$	Construction for loops
$c!v \rightarrow P$	Writing *v* to the channel of communication *c*
$c?v \rightarrow P$	Reading *v* from the channel of communication *c*

[1] Timed CSP

In the conventional CSP notation processes are represented by capital letters of the alphabet while lower case letters represent events. If *P* and *Q* are processes, the parallel execution of *P* and *Q* can be represented by $P||Q$, while the sequential execution is represented by $P;Q$.

If *P* is a process and *a* is an event, then $a \rightarrow P$ represents the construction of a process in which the occurrence of the event *a* assumes the state of *P*. Operations like Guard are also part of the CSP constructions [3, 4, 5].

Table 1 presents a summary of the main process constructions of CSP. In [2, 4] the authors present other constructions.

3 JCSP Library

The JCSP library implements the functionalities and operations of the CSP model for Java. Welch [7, 8, 9] used the CSP model to prove the correctness and viability of creating applications using the JCSP package instead of using conventional threads. According to him, threads in Java use synchronization primitives based on the concept of monitors, developed in the 70's. Taking into account that the semantics of threads in Java can lead to errors and to inconsistent situations due to its synchronization mechanisms, tools for automatic verification can and should be used to assist in the verification of models.

Tools can assist in the detection of problems such as deadlocks and livelocks and in proving the consistency between algorithms. Welch detected that the conventional multithreaded programming is arduous and hard to be debugged. Many mistakes, especially those of data corruption, have been detected in systems using the conventional thread model. Some errors only became apparent when JITs (*Just-in-Time*), accelerators of bytecode execution that act extremely fast in execution time, were used.

With all the mentioned problems, the libraries based on the mathematical formalism of CSP and the tools for verification of system consistence, such as FDR, Failures-Divergences-Refinement [10] are good alternatives for the development of concurrent and distributed applications. According to Hoare [3] there are two kinds of profitable systems: those that are obviously correct and those that are not obviously wrong. Unfortunately, it has been observed that it is very easy to create the second category of systems due to many factors. The use of tested libraries, such as the JCSP and CTJ, helps reducing the number of bugs originated from the lack of knowledge and experience of programmers.

3.1 Creation of Channels and Processes in JCSP

The types of JCSP communication channels with regard to the number of processes involved in the communication are:

- `One2OneChannel`: peer to peer communication;
- `Any2OneChannel`: communication any to one process;
- `One2AnyChannel`: communication one to any process;
- `Any2AnyChannel`: communication any to any process.

The interfaces for reading and writing data between processes are:

- `ChannelInput`: defines the `read()` method to input objects from the channel;
- `ChannelOutput`: defines the `write()` method to output objects to the channel.

3.2 Process Creation

JCSP has an interface called `CSProcess` that defines the basic functionalities for creating a process and the `run()` method that is executed in the `Sequential` and `Parallel` constructions. A process is in fact a Java object created from a class that implements the `CSProcess` interface. To implement Boukerche's algorithm we created a class called `DckNode` that implements `CSProcess` and defines its `run()` method. There is a biunivocal relation between the nodes of a graph that is submitted to the algorithm and the created processes.

In a graph with N nodes, N objects of type `DckNode` will be instantiated. The code below presents a summary of the `DckNode` class. Some of the attributes of a process (an object of type `DckNode`) are the `pid` (process identifier) of the process, the status of the process according to the algorithm, its `mode` (awake or asleep), `chIn` (the input channel of the process through which the process receives messages), `chOut[]` (the channel that links a process p_k and all its successors), `S` which records in each step the nodes that have sent messages of type `incomplete_search`, and the `parent` process identifier.

```
public class DckNode implements CSProcess {

  private int pid;
  private int status = -1;
  private int mode;
  private int numSuc;
  private StringBuffer S = new StringBuffer("");
  private ChannelInput chIn;
  private ChannelOutput[] chOut;
  public int parent;

  public DckNode (int pid,                    // constructor
                  ChannelInput chIn,
                  ChannelOutput[] chOut,
                  int[] suc) {
    this.pid = pid;
    this.chIn = chIn;
    this.chOut = chOut;
    mode = AWAKE;
    status = DckEnvelope.UNDEFINED;
    this.suc = suc;
  }

  public void run(){                          // main method
    if (pid == 0)
      ...  execution of the algorithm that initiates the search
    else
      ...  execution of the algorithm in process p[k]
  }

  ...  other methods

}
```

The `run()` method of each process p_k has its functionality defined according to the algorithm for a process p_k. A process receives a message by calling the method `read()` from its `chIn` attribute and sends messages to a process p_k calling the method `write()` from its `chOut[k]` attribute.

The implementation also defines the classes `DckEnvelope`, which serves as a basis for the creation of messages exchanged between processes, and `Main`, which is responsible for reading the graph, constructing processes according to the graph configuration and initializing its parallel execution. In Section 6 we will continue the description of the classes that implement the algorithm. Before we must explain and model the main construction of Boukerche's algorithm in the syntax of CSP.

4 Boukerche's Algorithm for Detection of Cycles and Knots in Directed Graphs

In [1], Boukerche presented his definition of cycles and knots in directed graphs. He adopted the model proposed by Chandy and Misra [20].

Consider a network (N, L) consisting of a set of nodes (or processes) and a set of communication links connecting the nodes in N. There is a correspondence between the network processes and the vertices of the graph $G = (V, E)$. A network process p_i is represented by a node i of the graph G. If there is an edge (i, j) or (j, i) that connects two nodes p_i and p_j, these nodes are considered neighbors. A process p_i is a successor of p_j if there is an edge from node i to node j in graph G, and is a predecessor of p_j if there is an edge from node j to node i. Now we can define cycles and knots in directed graphs.

A path P_{ij} is a sequence of nodes and edges from node i to node j. Node j is thus said to be reached from node i. A closed path (or circuit) P_{ii} is a cycle if there are no vertices in P_{ii} that occur more than once except for node i, that occurs exactly twice. If there is a path P_{ij} in the graph, then p_i is an ancestor of p_j and p_j is a descendent of p_i. Figure 1 presents an example of a graph in which a cycle occurs.

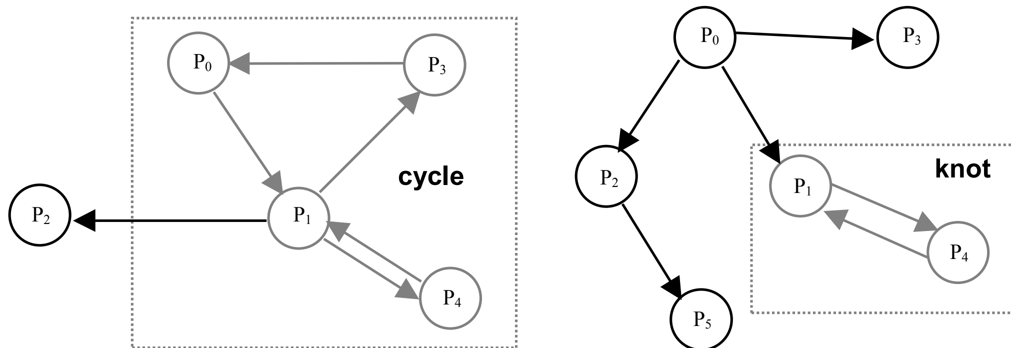

Figure 1: example of a cycle in a directed graph **Figure 2:** example of a cycle in a directed graph

A knot K in $G = (V, E)$ is a strongly connected sub-graph of G with no edges directed away from the subgraph K. A node i is a member of a knot K if it can be reached from all the nodes that are reached from the node i, figure 2.

4.1 Introduction to the Algorithm

In [1] Boukerche describes his algorithm for detection of cycles and knots in directed graphs and compares it with other existent ones. The algorithm is capable of detecting cycles and knots in generic graphs with a single evocation and with a minimal number of messages exchanged between the nodes.

According to Knapp [17], algorithms for detection of deadlocks can be classified in four categories: *path-pushing* [18], *edge-chasing* [19], *diffusing computation* [20] and *global state detection* [21, 22]. To these categories Boukerche added a new clustering technique,

developed by Cidon [13]. He argues that path-pushing and edge-chasing are appropriate only for the detection of cycles, while the other categories are appropriate for the detection of knots [1].

Boukerche analyzed other works related to the detection of cycles and knots. He concluded that some of the existent algorithms, such as those by Natarjan [14], Chandy and Misra [23] and Cidon [13], needed a high complexity of messages to reach his objectives. Other approaches like global state detection and algorithms using other communication principles were analyzed. All had the disadvantage of having a high message complexity [1].

Boukerche then based his algorithm on diffusing computation [20, 24], that requires a constant propagation of messages through the nodes of the graph during the search. He defined some criteria such as the inclusion of the message type INCOMPLETE_SEARCH, which allowed his algorithm to have a low message complexity. Another aspect was the imposition that only one node could initiate the search at each moment and that in the beginning of the search this node should only send messages to its successors. With this approach Boukerche was able to establish a smaller message complexity for his algorithm compared to the previous algorithms [13, 14, 20]. He also improved the message complexity of the distributed algorithm by Chandy and Misra [20] for detection of knots achieving a total number of messages of $2m$, which meant a reduction of at least $4m$, where m is the number of links of the graph. The amount of space required is $O(n \log n)$ bits of memory, n being the number of processes.

In his algorithm [1], Boukerche assumes that there is no loss of messages between nodes and that links never fail. In the links there is no message duplication [7] and messages obey the FIFO (*First In First Out*) policy, that is, the first message received by a process is the first to be processed. We maintained these principles in our implementation. The code listings below present the pseudocode of the algorithm of a process P_0, considering that process P_0 is the one that initiates the algorithm, and in a generic process P_k.

4.2 Algorithm in Process P_0

```
Step A1.

 Initial atributions:
   Mode_0 = Awake;
   Status_0 = Undefined;
   num_Suc_0 = Number of sucessors of  P_0;
   P_0 send <Request, P_0> to all sucessor;
   // Await for messages
   * Upon receiving a request <Request, P_i>


Step A2.

   if P_0 has sent a cycle message then
        S = {P_0} /* it is used to distinguish single cycle,
                    disjoint cycles */
   else
        S = ∅
   endif
   P_0 send <Reply, cycle, P_0, S> para P_k
   * Upon receiving <Reply, type, P_k, S>
```

Step A3.

```
   NumSuc0--
   S0k = S0k ∪S
   A3.1
         Status0 = status0 ⊕ type;
         /* type ⊕ cycle_only = cycle_only,
            cycle ⊕ leaf = cycle_only,
            cycle ⊕ incomplete_search = cycle,
            leaf ⊕ incomplete_search = leaf. */

   if numSuc0 = 0 then   /*Finish the computation */
         if status0 = cycle e S0k = ∅ then
               P0 is in a cycle
         else
         if status0 = cycle and all   elements of S0k are marked
                     P0 is in a knot
         else
                     P0 is not in cycle neither in a knot
         endif
         endif
   endif
```

4.3 Algorithm in a Process P_k (k>0)

Step B1.

```
 Initial attributes:
   modek = sleep;
   statusk = Undefined;
   num_Suck = 0;
   * Upon receiving a <Request, p>
```

Step B2.

```
   if Pk has already finished its search then
      send to p the same reply pk when it terminates its serach;
   endif
```

Step B3.

```
   if modek = awake then     // it's participating
     send <Reply, incomplete_search, pk, { pk} > to p
   else                      // it's not participating
      if pk has not sucessor  /* pk is a g-leaf */
        send <Reply, leaf, pk ,∅> to p
      else
        parentk = p
        modek = p
        num_Suck = Number of sucessor of pk
        send <Request, pk > to all sucessors of pk
      endif
    endif
   * Upon receiving a <Reply, type, pj, S>
```

```
Step B4.

  num_Suc--
  S_kj = S_kj ∪ S
  status_k = status_k ⊗ type

Step B5.

  if num_suc_k = 0 then /* it´s terminated its search */
    S_kj = S_kj ∪ S
    if pk has sent a message of type incomplete_search then
      S = S ∪ {p_k}
      if (status_k = leaf) or
         ((status_k = incomplete_search) and
          (all elements of S are marked)) then
        send <reply, leaf, p_k, ∅> to parent of k
      endif
    endif
  endif
  if (status_k = incomplete_search) and
     (S has at least one element not marked) then
    send <reply, incomplete_search, p_k, S> to k's parent
  endif
  if status_k = cycle_only then
    send <reply, cycle_only, p_k, ∅> to parent of k
  endif
  if status_k = cycle then
    if pk has received a message of type cycle then
      send <reply, cycle, p_k, S> to parent of k
    else
      if all elements of S_kj are marked then
        send <reply, cycle_only, p_k, S> to parent of k
      else
        send <reply, cycle, p_k, S> to parent of k
      endif
    endif
  endif
```

5 Modeling of the Algorithm

The nodes of a given directed graph G are referenced as 0 to m, therefore, $G = \{ k \mid 0 \leq k < m\}$. Node 0 is the node that initiates the search. The set of edges that connect each pair of nodes of G is defined by the set E. For each node $k \in G$ from node 0, the set of all adjacent and successor nodes of k is called $suc(k\}$ and is defined as:

$$suc(k) = \{j \in G \mid (k,j) \in G\}$$

The CSP definition describes each node of G as a CSP process and each edge of E as a CSP communication channel. In our implementation, for each two neighbor nodes k and j, there is a communication channel c_{kj} that permits that the messages travel from k to j and a channel c_{jk} so that messages can travel on the inverse path. Each channel c_{kj} has a complementary channel c_{jk} in the opposite direction.

Messages that pass through the communication channels can be of two types: request or reply. Reply messages can have the codes:

- cycle, when p_0 and p are in a same cycle;
- cycle_only, when p_0 and p are in a same cycle but are not part of a knot;
- incomplete_search, when the successors of p still haven't completed the search;
- leaf, when p is a leaf or a g-leaf.

We can then define the set of values (events) V of messages that can be sent through the communication channels E:

V = {request, reply, reply_cycle, reply_cycle_only, reply_incomplete_search, reply_leaf}

Node 0 starts the search sending a request message to all its successors and enters in wait mode waiting for the respective answers to be received. The node's situation will be computed after it receives the result of all its successors through reply messages. The alphabet of messages of node 0 is defined by:

$A_0 = \{c_{0l}.r \mid r \in V \wedge l \in suc(0)\}$

The alphabet for a node k, $k \neq 0$, is:

$A_k = \{c_{kl}.r \mid r \in V \wedge l \in suc(k)\} \cup \{c_{lk}.r \mid r \in v_{kl} \wedge l \in suc(k)\}$

The CSP process that detects cycles and knots will be called *DCK* (Detect Cycles and Knots). It receives a graph g as a parameter, over which the search will be performed. *DckNode* is the process that computes the output according to the situation of node 0 in the given graph.

DCK = in?g → out! $DckNode(g)$ → STOP

where g is the given graph.

The specification of *DckNode* is:

$DckNode = \|^{k \in G}_{Ak} DckNode(k)$

5.1 CSP Specification of Node/Process P_0:

```
DckNode0 = c0i!request → WAIT
WAIT = ck0?msg → PROCESS_MSG →
        if hasReceivedAllReply → RESULT
        else → WAIT
```

```
PROCESS_MSG =
  if msg = request       -- Request message received
    if  hasSentMsgCycle
      S = { p̄₀}
   else
     S = ∅
     c0i!reply.cycle     -- Send a cycle message
  else                   -- Reply received
    RESULT =
      if status = cycle_only
        print('P0 is in a cycle')
      else
        if stauts = cycle and S0j = ∅
          print('P0 is in a Cycle only!')
        else
          if staus = cycle and allElementsOfSkjAreMarked
            print('P0 is in a knot!')
          else
            print('P0 is not in a Knot neither in a Cycle')
```

5.2 CSP Description in a Node/Process P_k ($k>0$)

```
DckNodek = INICIAk → cjk?msg → PROCESS_MSGk → DckNodek

INITk = modek := sleep; statusk := UNDEFINED; numSuck := 0

PROCESS_MSGk =
if msg = request         -- Request message received
   PROCESS_REQUESTk
else
   PROCESS_REPLYk

PROCESS_REQUESTk =
   if pk has already finished is search
   ckp!msg.REPLY to parent p    -- Send a reply
   else
      if modek=awake
    ckp!msg.INCOMPLETE_SEARCH to parent p
     else
      if pk has not sucessor
         ckp!msg.LEAF to parent p
      else
     parentk = p;modek=awake;num_suck=Nbr of sucessor of pk
   ckp!msg.REQUEST to all sucessors of pk

PROCESS_REPLYk = num_suck--;
   Skj = Skj ∪ S
   statusk = statusk ⊗ type
   if num_Suc=0
   S = ∪ Skj
CONTINUE_REPLYk     // Process messages according the algorithm

END
```

6 Algorithm Implementation

Class `DckNode`, that serves as basis for the creation of the processes used in the algorithm, was presented in Section 3.2. The messages exchanged between processes are objects of the class `DckEnvelope`:

```
public class DckEnvelope {

   public int destino;            // receiver of the message
   public int remetente;          // forwarder of the message

   public StringBuffer conteudo;  // message content: set S

   public int type;               // type of the message

   public int codigo;             // message code

   public static final int REQUEST = 0;
   public static final int REPLY = 1;
   public static final int CYCLE = 2;
   public static final int CYCLE_ONLY = 3;
   public static final int LEAF = 4;
   public static final int INCOMPLETE_SEARCH = 5;
   public static final int UNDEFINED = 6;

   public DckEnvelope (int destino, int remetente,
                       int type, int codigo,
                       StringBuffer conteudo){
     this.codigo = codigo;
     this.destino = destino;
     this.remetente = remetente;
     this.type = type;
     this.codigo = codigo;
     this.conteudo = conteudo;
   }

   ...  other methods

}
```

Each message used in the algorithm will be an object of type `DckEnvelope`. The destination and source, type (request or reply), code (`CYCLE_ONLY`, `CYCLE`, `LEAF`, `INCOMPLETE_SEARCH` or `UNDEFINED`) and contents (set S) of the message should be informed at the time of construction of the object. The set S keeps track of the nodes that sent messages of type `INCOMPLETE_SEARCH`, that is, those that did not end the search in each execution step.

Class `DckNode`, which serves as basis for the creation of the objects (processes) that will represent the nodes of the graph were previously defined. Each node of the graph will be represented by an object of type `DckNode`. When execution starts, for a graph G with N nodes, N objects of type `DckNode` are created. The connection between the processes is made according to the graph configuration. The code listing below presents the construction for a graph with six nodes.

```
public class Main {

  public static void main (String args[]){

    int[][] sucPi = readGraph ();   // generate simulation graph
```

```
    // create the channels
    Any2OneChannel c[] = Any2OneChannel.create (sucPi.length);

    // create the array to the process
    CSProcess[] p = new CSProcess[sucPi.length];

    // alocate the process
    for (int i = 0; i < p.length; i++){
      p[i] = new DckNode (i, c[i], c, sucPi[i]);
    }

    // create the parallel construction of the process
    Parallel pp = new Parallel (p);

    // start the execution
    pp.run ();

  }

}
```

The construction `Any2OneChannel c[] = Any2OneChannel.create (...)` creates an array of channels to link the processes. Then, `CSProcess[] p = new CSProcess[P]` creates a process for each node of the graph. The construction `sucPi = readGraph()` returns a square matrix that keeps the configurations of the graph to be used in the simulation. Column 0 of the matrix contains the identification of all the k nodes of the graph, k [0 ... m-1]. Column j ($0 < j < m$, where m is the number of nodes of the graph) keeps the identification of the successor nodes of each node k of column 0. The construction:

```
for (int i = 0; i < p.length; i++) {
  p[i] = new DckNode (i, c[i], c, sucPi[i]);
}
```

creates the processes according to the generated matrix `sucPi` (of type `int[][]`). Each process receives its identification `pid`, a reference to its input channel, references to the input channels of all processes and the list of its successor processes. The construction `Parallel pp = new Parallel(p)` makes the parallel construction of the processes and `pp.run()` starts their parallel execution ($\|_{k[0..m]} P_k$).

An important construction in Boukerche's algorithm is the operation $status_k = status_k \otimes type$, which updates the status of a process p_k in each execution step of the algorithm. In our algorithm we created a method called `opSpecial`[2] (Special Operator), that implements the ⊗ operation [1].

In the listing of `DckNode`, for each `msgRec` message received (object of type `DckEnvelope`), the status attribute is updated using the `opSpecial` method with the following line of code:

```
status = opSpecial (status, msgRec.type);
```

The union of marked and unmarked elements of S in the algorithm is realized by the method called `unionMarked`, whose listing is presented below:

[2] Special Operator: $status_k = status_k \otimes type$

```
/** Execute the operation to update set S
 *  with elements marked and not marked.
 *  Ex: {p1 P2 P3} union marked {P1 P5} = {P1 P2 P3 P4}
 *  @param set Skj
 *  @param set S
 *  @return normalised set  */
public static StringBuffer unionMarked (StringBuffer skj,
                                        StringBuffer s) {
  String pk = null;          // {pk} union marked {/pk} = {/pk}
  if (s == null) {
    return skj;
  } else {
    StringTokenizer st = new StringTokenizer (s.toString ());
    while (st.hasMoreElements()) {
      pk = st.nextToken();
      int index = skj.toString().toUpperCase().
                  indexOf(pk.toUpperCase());
      if (index == -1) {
        skj.append(" "+ pk);  // insert pk in the final of skj
      } else {
        if ((pk.charAt(0) == 'P') &&
            (skj.charAt(index) != 'P')) {
          skj.replace(index, index+(pk.length()), pk);
        }
      }
    }
    return skj;
  }
}
```

The operation (method) that performs the search to verify if all the elements in S are marked is the following one:

```
/** Verify if all elements of S are marked */
private boolean isAllElementsSMarked(StringBuffer sb){
   return (sb.toString().indexOf("p") == -1);
}
```

To handle the elements of S that keep the information about the sent messages of type incomplete_search, Boukerche uses the concept of marked $\overline{p_k}$ and unmarked p_k elements [1]. In our implementation, an element p_k that is put in S can have a capital or lower case p. If it is a lower case p, it means that the element is unmarked. On the other hand, if it is a capital P, the element is marked. For example, a possible set of elements for S would be: *{P1 p3 P4}*. The elements are separated by single spaces. In the presented set *P1* and *P4* are marked while *p3* is unmarked.

7 Simulations

Figure 3 shows the first screen of `Dck`. The main parts are where the configuration of the graph is showed (text area) and can be edited, on the left side of the screen, and where the user can click on the buttons *Execute*[3] and *Result*[4], on the right side of the screen. Each line of the text area represents a node i of the graph and its successors. In this version, `Dck` shows the situation of the first node of the graph according to the algorithm. The node can be in a cycle, in a knot, in both or neither.

[3] Starts the execution to the current graph
[4] Shows the result of the last simulation

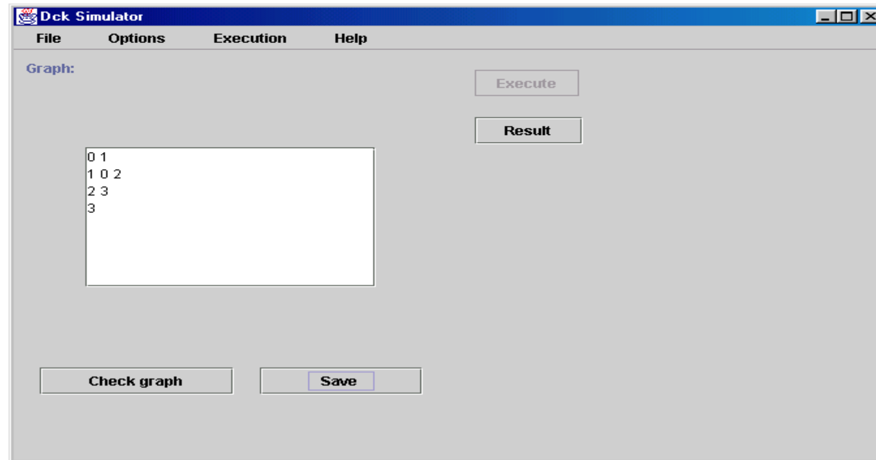

Figure 3: first screen of `Dck`

The function *Check graph* verifies if the graph is valid. After this operation the button *execute* is enabled. One click on the *Execute* button submits the graph to the algorithm. After the execution, the screen with the result is showed, as seen at figure 4.

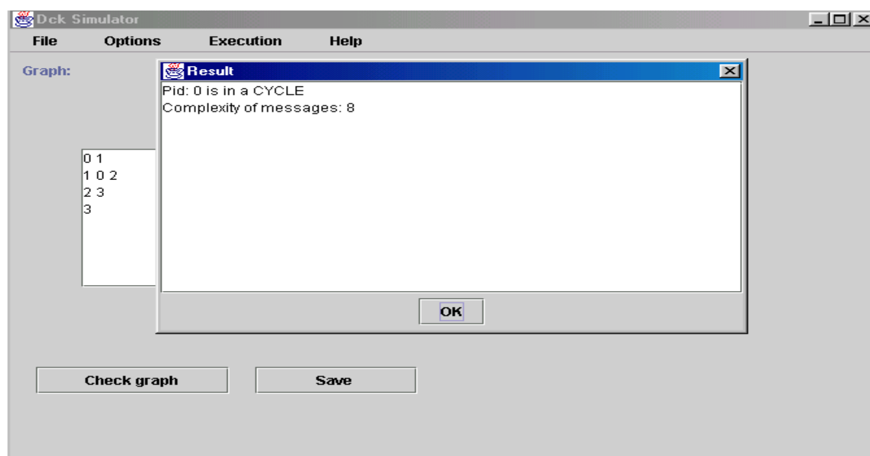

Figure 4: result of the simulation

The log (trace) of the simulation is sent to a log file. Some lines of the simulation log of the latest execution are shown below:

```
DBG 25/05/2002 17:56:38.310 DckMain Pid: 1 REPLY received from pid: 2
OPR 25/05/2002 17:56:38.310 DckMain Pid: 2 finished...
OPR 25/05/2002 17:56:38.310 DckMain Pid: 1 executing shutdown…
DBG 25/05/2002 17:56:38.310 DckMain Pid: 1 sending message to pid 0.
DBG 25/05/2002 17:56:38.310 DckMain Pid: 0 receiving message from pid 1.
DBG 25/05/2002 17:56:38.310 DckMain Pid: 1 message sent to pid: 0
OPR 25/05/2002 17:56:38.310 DckMain Pid: 0 executing shutdown
OPR 25/05/2002 17:56:38.310 DckMain Pid: 1 stopped...
DBG 25/05/2002 17:56:38.310 DckMain Pid: 0 status: 3
DBG 25/05/2002 17:56:38.310 DckMain Pid: 0 is in a CYCLE
OPR 25/05/2002 17:56:38.310 DckMain Pid: 0 complexity of messages: 8
OPR 25/05/2002 17:56:38.310 DckMain Pid: 0 stopped...
OPR 25/05/2002 17:56:38.310 DckMain ExecuþÕo realizada...
```

Generally, the following information is written for each message saved: *date*, *time*, *pid* of the process and the *text* of the messages. This option is very useful to the simulation with a graph full of nodes. Other options and features of `Dck` can be viewed in its user manual.

8 Results and Conclusion

In this work we presented an implementation of Boukerche's distributed algorithm for detection of cycles and knots in directed graphs [1]. The utilization of the algorithm is important in parallel simulations when cycles and knots can frequently occur, when the size of the graph is big and when it is necessary to know if a given node is in a cycle or in a knot. We could see that the message complexity of the implemented algorithm was 2m, m is the number of links in the graph, in accordance with the value stipulated by Boukerche.

The algorithm implementation was based on the construction of processes and channels existent in the CSP model [2, 3, 4, 5]. The algorithm was inlaid in a graphic application, called `Dck` (*Detection of Cycles and Knots*), which permits testing with generic graphs. The implementation presents an option for saving the execution logs in a file, which permits the accompaniment of the simulation of the algorithm execution. This is especially important for complex graphs with large quantities of nodes.

Two relevant aspects in the implementation of the algorithm were the use of the CSP notation for the modeling of the algorithm and the JCSP library for its implementation. With CSP it was possible to document and model the algorithm. After understanding the problem, we went on to its implementation. The use of the JCSP library permitted a faster and safer implementation because of the facilities provided by it, such a process creation, communication channels between processes and parallel construction of processes.

Works are being developed to make Java become a real time language [25, 26, 27, 28, 29, 30, 31]. The creation of a real time Java virtual machine will certainly be an important step to improve application performances. It is believed that Java programs will soon be executing with the same speed as C and C++ programs [29]. With this advent the Java platform will be also viable for the computation of real time applications and projects like JCSP and CTJ will gain more space in the development process.

For future work, we would like the inclusion of a module that permits the user to graphically edit a graph. Although the current version is not complicated, it can be laborious for large graphs.

Acknowledgments

This work is based on an algorithm proposed by Professors Azzedine Boukerche (University of North Texas) and Carl Tropper (McGill University, Montreal-Canada), whom we would like to thank for the attention. This research was partially supported by CNPq, Brazil.

References

[1] A. Boukerche, C. Tropper, "*A Distributed Graph Algorithm for the Detection of Local Cycles and Knots*". IEEE Transactions on Parallel and Distributed Systems, Vol. 9, No. 8, August 1998.
[2] C.A.R. Hoare, *"Communicating Sequential Processes"*. Prentice Hall, London, UK, 1985.
[3] C.A.R. Hoare, *"Communicating Sequential Processes"*. Communications of the ACM, Vol. 1 Number 8, pg 666-677, 1978.
[4] S. Schneider, "*Concurrent and Real-Time Systems: The CSP Approach*". Wiley, Chichester, 2000.
[5] A.W. Roscoe, "*The Theory and Practice of Concurrency"*. Prentice Hall, London, UK, 1998.
[6] G. Hilderink, A. Bakkers, and J. Broenink, *"A Distributed Real-Time Java System Based on CSP.,* III International Conference in Distributed and Real Time Computing using Object Oriented, IEEE 2000.

[7] P. Welch and J. Martin, *"A CSP Model for Java Multithreading*". Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE 2000), IEEE, 2000.
[8] JCSP Project, "*Communication Sequential Processes for Java*", University of Kent, UK, http://www.cs.ukc.ac.uk/projects/ofa/jcsp/.
[9] P. Welch, "*Process Oriented Design for Java Concurrency for All*". Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000), vol. 1, pages 51-57. CSREA Press, June 2000.
[10] Formal Systems Europe Ltda, http://www.formal.demon.co.uk/ .
[11] Sun Microsystems S/A. Url: http://www.java.sun.com.
[12] G. P. de Souza, G. H. PFITSCHER, e A. C. MELO. "*Distributed Computing based in Java on Beowulf and Heterogeneous Environment*". In: WPERFORMANCE/SBC 2002, Brazil, 2002.
[13] I. Cidon, "*An Efficient Distributed Knot Detection Algorithm,* ", IEEE Trans. Software Eng., vol. 15, no. 5, pp. 644-649, May 1989.
[14] N. Natarjan, *"A Distributed Scheme for Detecting Communication Deadlocks*", IEEE Trans. Software Eng., vol. 12. no. 4, pp. 531-537, Apr. 1986.
[15] S. Huang, *"A Distributed Deadlock Detection algorithm for CSP-Like Communication"*, ACM Transactions on Programming Languages and Systems, Vol. 12, No. 1, January 1990, Pages 102-122.
[16] G. Hilderink e J. Broenink, "*Communicating Threads for Java*" (CTJ), University of Twente, NL, Url: http://www.rt.el.utwente.nl/javapp.
[17] E. Knapp, "*Deadlock Detection in Distributed databases Systems*". ACM Computing Surveys, pp. 303-328, Dec. 1987.
[18] R. Obermark, "*Distributed Deadlock Detection Algorithm*". ACM Trans. Database Systems, vol. 7, no. 2, pp. 187-208, June de 1982.
[19] D. P. Mitchell and M. J. Merrit, *"A Distributed Algorithm for Deadlock Detection and Resolution".* Proc. ACM Symp. Principle of Distributed Computing, pp. 282-284, Aug. 1984.
[20] K. M. Chandy and J. Misra, "*A Distributed Graph Algorithm: Knot Detection*". ACM Transactions on Programming Languages and Systems, Vol. 4, Pages 678-686, October 1982.
[21] G. Bracha e S. Toueg, "*A Distributed Algorithm for Generalized Deadlock Detection*". Distributed Computing, 1987.
[22] A. D. Kshemkalyani and M. Singhal, "*Efficient Detection and Resolution of Generalized Distributed Deadlocks".,* IEEE Trans. Software Eng., vol. 20, no. 1, pp. 43-54, jan. 1994.
[23] K. M. Chandy, J. Misra and L. Hass, "*Distributed Deadlock*". ACM Trans. Computer Systems, vol. 1, no. 2, pp. 144-156, May 1989.
[24] E. W. Dikstra and C. S. Scholten, "*Termination Detection in Diffusing Computation*". Information Processing Letters, vol. 11, no. 1, pp. 1-4, Aug. 1980.
[25] W. Weinberg, "*Real-Time Java Implementation for Embedded Systems*". Real-Time Magazine, no. 1, pp. 43-49, Jan-march 1998.
[26] R. Helainhel and K Olukotun, "*Java as a Specification Language for Hardware-Software Systems*". ICCAD, IEEE, 1997.
[27] E. Yero, M. Henriques, J. Garcia and A. Leyva, *"JOINT: An Object Oriented Message Passing Interface for Parallel Programming in Java*". The International Conference and Exhibittion on High-Perfformance Computing and Networking HPCN, p636-p646, Europe, 2001.
[28] R. Veldema, T. Kieldmann and H. Bal, "*Optimizing Java-Specific Overheads: Java at the Spedd of C?*". The International Conference and Exhibittion on High-Performance Computing and Networking HPCN, p675-p684, Europe, 2001.
[29] E. Bertolissi and C. Preece, "*Java in Real-Time Applications*", IEEE Trans. in Nuclear Sciense, vol. 45, no. 4, part 1, pp. 1965-1972, Aug. 1998.
[30] D. Hardin, "*The Real-Time Specification for Java*", Dr. Dobb´s Journal, pp. 78-84, Feb. 2000.
[31] RTJEG, The Real Time for Java Expert Group (2000), "*The Real-Time Specification for Java*", Addison-Wesley, 2000.