Some comments on "Deadlock Detection is Cheap" in SIGMOD Record Jan 83

W.N. Chin
Dept. of Computer Science
University of Manchester
Manchester M13 9PL
U.K.

The deadlock detection scheme suggested by Agraval, Carey and DeWitt appears to be a rather elegant and efficient method for overcoming one aspect of the deadlock problem (ie detection) which arises from most concurrency control techniques in database systems. It needs to be pointed out however that the algorithm suggested for cycle-detection in the Periodic Deadlock Detection case has been based upon incorrect assumptions.

It was mentioned that the Detect-Cycle function (in Periodic Deadlock Detection) would be anologous to the Chk-Cycle function (in Continuous Deadlock Detection), where a directed walk in the waits-for graph starting from vertex, v, will terminate at a root (if no cycle exists in that part of the sub-graph) or will again reach v (in the case of a cycle). This assumption however is incorrect. The algorithm for Detect-Cycle needs to be substantially different from Chk-Cycle.

In Periodic Deadlock Detection, the vertex first chosen need not necessarily be involved in a cycle for that part of the sub-graph. The directed walk will thus not necessarily return to its initial vertex, v. This is best illustrated by part of a possible waits-for graph in Fig. 1, where Ta, being a possible initial vertex to be chosen for cycle-detection, is not part of the cycle that has been formed further up the sub-graph.

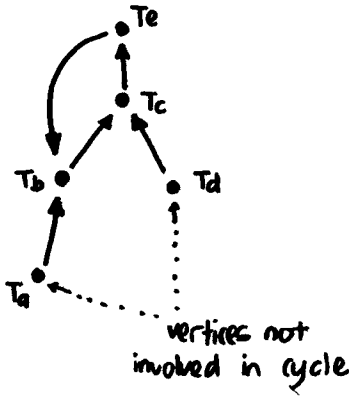

Fig. 1 : Part of a waits-for graph

Two possible algorithms will now be suggested for the Detect-Cycle function. The first one make use of a counter while the second requires an additional boolean in each of the nodes present in the waits-for graph.

The first algorithm relies on the fact that in any acyclic graph, its maximum path length before it terminates at a root should not exceed the total number of nodes in the graph. By the converse logic, if a directed walk takes us through a path exceeding the total number of nodes in the graph, it must have encountered a cycle. (The graph could not have been acyclic)

```
Detect-Cycle-1 (v : transaction£);

   CONST N = total_no_of_tran;

   VAR count : INTEGER;
       ancestor : transaction£;

   BEGIN

   ancestor:=v;
   count    :=0;

   LOOP

      {  IF ancestor=null THEN return(ok)
         ELSE { tran[ancestor].visited := true;
                count := count + 1;
                IF count>N THEN return(deadlock)
                ELSE ancestor := tran[ancestor].waitingfor
              }
      }     -- end LOOP


   END;
```

The second algorithm employs an additional boolean (called passed) in each of the transaction nodes to mark the current subgraph traversed. Should a directed walk take us back to a node with passed marked, then a cycle has been detected. Notice that the boolean visited could not be used as a substitute for the above purpose.

```
Detect-Cycle-2 (v : transaction£);

   VAR state   :  (carryon,ok,deadlock);
       t,ancestor : transaction£;
       finish :  BOOLEAN;


   BEGIN

   ancestor:=v;
   state:=carryon;

   WHILE state=carryon DO

      {  IF ancestor=null THEN state:=ok
         ELSE IF tran[ancestor].passed THEN state:=deadlock
         ELSE IF tran[ancestor].visited THEN state:=ok
         ELSE BEGIN
                tran[ancestor].visited:=true;
                tran[ancestor].passed:=true;
                ancestor:=tran[ancestor].waitingfor;
              END
      }


   t:=v; finish:=false;

   REPEAT

     IF t=null THEN finish:=true
     ELSE IF tran[t].passed=false THEN finish:=true
     ELSE BEGIN
            tran[t].passed:=false;
            t:=tran[t].waitingfor
          END

   UNTIL finish;      -- loop to clear away passed marked
                      -- in the current traversal

   return(state);

   END;
```