数理解析研究所講究録
第236巻 1975年 131-152

Correctness of Co-operating Sequential Programs

Nobuo Saito
(Electrotechnical Laboratory)

1. Introduction

A modern computer system is usually operated in a multi-programming or a multi-processing environment. In such a system, a number of sequential programs are concurrently executed, either as a user program or as a control program of an operating system. Some of these sequential programs constitute one group for the purpose of carrying on a given task or a job. Exchanging control and information data, they co-operate with each other to accomplish their purpose. Programs of this group are called co-operating sequential programs[1]. Since there are many independent control flows in this kind of programs, it is very difficult to write correct programs.

Various methods for proving correctness of a given program have been developed. It originates with Floyd[2], and most of these methods aim to prove assertions about programs with single control flow.

This paper proposes one approach to extend these methods so that assertions about co-operating sequential programs can be proved. We will introduce an execution graph which is constructed as a direct product of given sequential programs. Each node of an execution graph is considered to represent a state of given co-operating sequential programs, and each arc is considered to represent a state transition which occurs in one computation of given programs. We place some assertions at each node of an execution graph. Assume that, for any arc in an execution graph, assertions, which are placed at its start

node with the function of the corresponding state transition, implies assertions placed at its end node. Then, we can conclude that assertions placed at any node will be satisfied provided that a computation gets to this node.

There are many problems to be considered when using an execution graph. These problems arise because of multiple control flows in co-operating sequential programs.

Usually several kinds of synchronizing primitives are used in order to synchronize the flows of control among several sequential programs. Various kinds of synchronizing primitives have been proposed, and we should choose such kinds of primitives that are suitable for a formal treatment.

Since an execution graph is mechanically constructed as a direct product of the member of given sequential programs, some of its nodes and arcs may not be realized in any computation because of the logical combination of execution conditions. Therefore, we should determine the realizability of nodes and arcs so that we can neglect all the unrealizable nodes and arcs in the proving process.

Two kinds of anomalies of the dynamic behavior may appear in the execution of co-operating sequential programs. One is a deadlock, and the other is an effective deadlock. They will be defined by using an execution graph. Both of them are caused by the logical structure of given sequential programs, and we should carefully construct programs so that neither deadlocks nor effective deadlocks exist.

This paper will discuss several aspects of the above mentioned problems with relation to the use of an execution graph in the verification of given co-operating sequential programs.

2. Execution Graph

Consider a set of sequential programs each of which is represented by a flow chart. Given a flow chart F_C, its flow graph is transformed from F_C by assigning a node to each branch of F_C and connecting two nodes with an arc provided that there is a common box between the corresponding two branches in F_C. An example of a flow graph is shown in Figure 1.

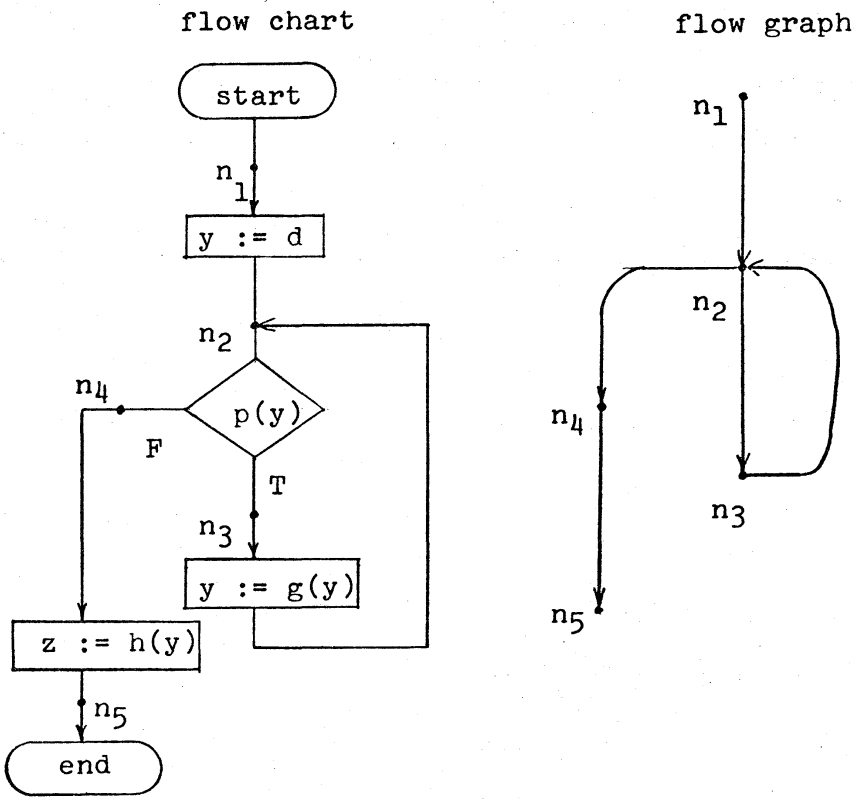


FIG. 1 flow chart and flow graph

Let $S = (p_1, p_2, \cdots, p_M)$ be a set which consists of M sequential programs. Each p_i ($1 \leqq i \leqq M$) has its flow graph with the following elements.

n_i: node set (including u_i nodes),

a_i: arc set (including v_i arcs).

$\forall \alpha \in a_i$, $\alpha : \nu \rightarrow \nu'$ ($\nu , \nu' \in n_i$)

where ν is called a start node of α ,

and ν' is called an end node of α .

Definition 2.1 (execution graph)

For a given set S of sequential programs, its execution graph E_S is defined as a directed graph which has the following elements.

N: node set (including $\prod_{i=1}^{M} u_i$ nodes)

Each element of N is labeled by an M-tuple $\langle \nu_1, \nu_2, \cdots, \nu_M \rangle$, where for all i ($1 \leqq i \leqq M$) ν_i belongs to n_i.

A: arc set

Each element of A should satisfy the following two conditions.

Condition 1

$$\forall a \in A, \quad a: \langle \nu_1, \nu_2, \cdots, \nu_M \rangle \rightarrow \langle \nu_1', \nu_2', \cdots, \nu_M' \rangle$$

$$\Longleftrightarrow \quad \exists_1 k \ (1 \leqq k \leqq M)$$

$$\nu_k \neq \nu_k'$$

$$\nu_j = \nu_j' \qquad (1 \leqq j \leqq M \text{ except } k)$$

$\langle \nu_1, \nu_2, \cdots, \nu_M \rangle$ is called a start node of a, and $\langle \nu_1', \nu_2', \cdots, \nu_M' \rangle$ is called an end node of a.

Condition 2

$$\forall a \in A , \quad a: \langle \nu_1, \nu_2, \cdots, \nu_M \rangle \rightarrow \langle \nu_1', \nu_2', \cdots, \nu_M' \rangle$$

$$\Longleftrightarrow \quad \text{if } \nu_k \neq \nu_k' , \text{ then}$$

$$\exists \alpha \in a_k \quad \alpha : \nu_k \rightarrow \nu_k' \quad (\nu_k , \nu_k' \in n_k)$$

Definition 2.2

A flow graph of each p_i of S has an initial node $i_i \in n_i$, and a set of terminal nodes $t_i \in n_i$. For an execution graph E_S of S, its initial node and terminal nodes are defined as follows.

$\langle \nu_1, \nu_2, \cdots, \nu_M \rangle \in N$ is an initial node of $E_S \Leftrightarrow \forall k\ (1 \leq k \leq M)\ \nu_k = i_k$

$\langle \nu_1', \nu_2', \cdots, \nu_M' \rangle \in N$ is a terminal node of $E_S \Leftrightarrow \forall k\ (1 \leq k \leq M)\ \nu_k' \in t_k$

Some control programs of an operating system may continue its execution indefinitely, and they have neither initial node nor terminal node. In this case, an initial node and a terminal node are not defined in an execution graph. An example of an execution graph is given in Figure 2.

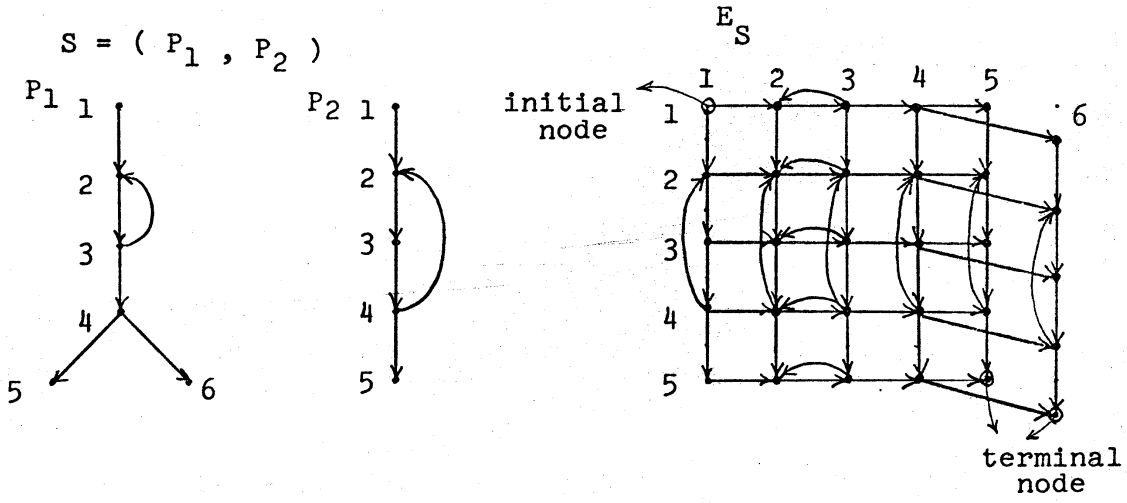


FIG. 2 Example of an execution graph

Let's consider the meaning of an execution graph E_S of co-operating sequential programs $S = (p_1, p_2, \cdots p_M)$. Assume that the execution of each statement in S should be an indivisible operation. Since an execution of a statement is assumed not to be disturbed by other statements executed concurrently, it is guaranteed that the semantics of a statement in co-operating sequential programs is just the same as the semantics defined in a single sequential program. Given this assumption, a node of E_S represents an instance of the combinations of the intermediate execution locations of all the sequential programs of S, i.e. it represents one execution state for a computation. The node set N of E_S contains all the possible combinations. An arc of E_S represents a single execution of a statement in one of the sequential programs of S, and a directed path between an initial and a terminal node defines a linear representation of an execution sequence for one possible computation in S.

Proposition 2.1 (Verification Theorem)

Consider an execution graph E_S of given co-operating sequential programs S. For each node $n \in N$, assign a key assertion Q_n. Each arc $a \in A$ is given its semantics S_a according to the corresponding single statement.

If for any arc $a: j \rightarrow k$ $(a \in A)$, the verification condition

$$Q_j \wedge S_a \supset Q_k$$

is proved, the key assertion Q_n assigned to a node $n \in N$ will be satisfied provided that there exists a computation which gets to this node.

The verification theorem is an extension of Floyd's method, and we should consider the special features of co-operating sequential programs when applying this theorem.

3. Synchronizing Primitives

In co-operating sequential programs, several kinds of "primitives" are usually used so as to synchronize timing of the execution of specific statements. The word "primitive" is a synonym of an executive macro statement or of a supervisor macro statement. The operating system implements functions of primitives without disturbing users' behaviors, and so primitives can be used as ordinary statements. In the formal treatment of such a primitive, we consider only its function, without taking account of its implementation method.

In this paper, the semaphore system proposed by Dijkstra[1] is selected to use as a synchronizing primitive. One reason is that the function of the semaphore system covers most of the functions of other synchronizing primitives, and the other reason is that the semaphore system is suitable for the formal analysis.

In the semaphore system, we can define any number of special purpose integer variables called semaphore variables. Two primitives, P-operation and V-operation are prepared as operations to semaphore variables. Each of these primitives takes a semaphore variable s as its argument. Their function is as follows.

P-operation : P(s)

(1) Decrease s by one.

(2) If s is non-negative, continue the computation.

(3) If s is negative, hang-up the computation and lie down in the queue of the variable s.

V-operation : V(s)

(1) Increase s by one.

(2) If s is non-positive, pick up one waiting program in the queue and activate it.

Since they are primitive operations, their executions are indivisible. After an initial value is given to a semaphore variable s, a user is allowed to access s only through P(s) and V(s) statements.

An example of co-operating sequential programs with P- and V-operations is given in the following.

Example 3.1

```
begin  semaphore  s;  s := 1;
  parbegin
    programl: begin
              labell:  P(s) ;
                       critical section 1 ;
                       V(s) ;
                       remainder of programl ;
                       go to labell
              end ;
```

```
program2:  begin
           label2:  P(s) ;
                    critical section 2 ;
                    V(s) ;
                    remainder of program2 ;
                    go to label2
           end
  parend
end ;
```

In the above example, a semaphore variable s is used so that the executions of critical sections exclude each other in time.

Habermann proved that the following relation always holds for the semaphore system [3].

Theorem 3.1

The effect of executing P(s) and V(s) is equivalent to the rule that the relation

$$ne(s) = \min [\, np(s),\ C(s) + nv(s) \,] \qquad (1)$$

is invariant for execution of P(s) and V(s),

where

np(s): how many times P(s) was executed;

nv(s): how many times V(s) was executed;

ne(s): how many times P(s) was passed, i.e. how many times a program was enabled to continue its computation;

C(s): an initial value of a semaphore variable s.

4. Realizable and Unrealizable Nodes and Arcs

4-1 For Co-operating Sequential Programs with P- and V-operations

When co-operating sequential programs are written by using P- and V-operations, the relation (1) given in Theorem 3.1 can be applied to determine the realizability of nodes and arcs. In order to know if the relation (1) holds, we need to count np(s), nv(s) and ne(s) in given programs.

When each of the sequential programs is a straight line program (i.e. a program without any conditional branch), it is easy to determine the above mentioned values.

Example 4.1

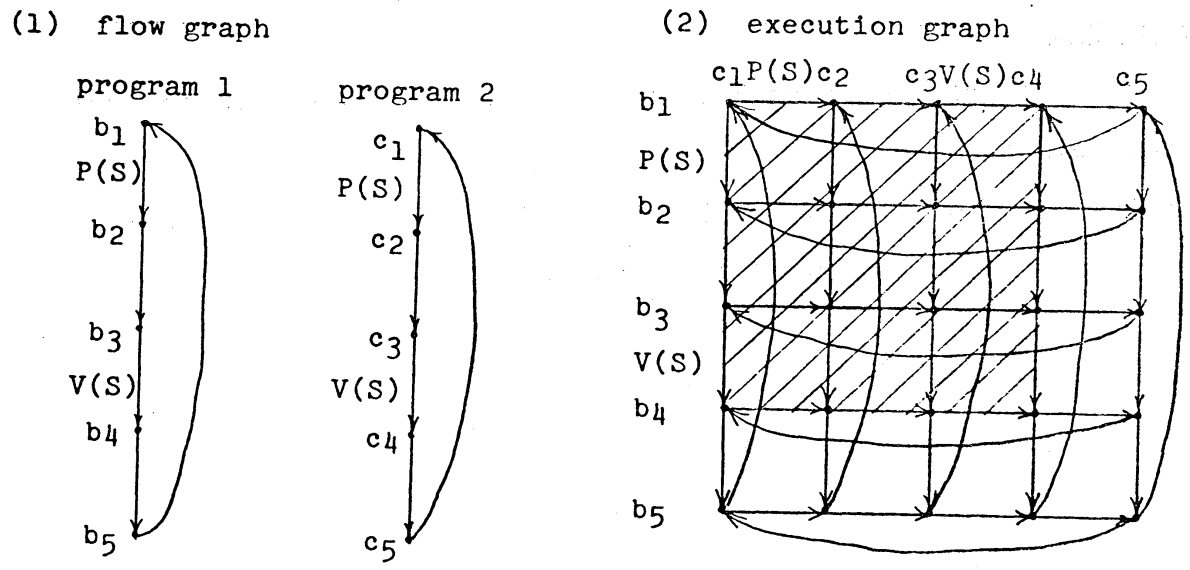


FIG. 3 Unrealizable nodes and arcs of an execution graph for a mutual exclusion problem

Consider the program given in Example 3.1. The flow graphs for program1 and program2 are shown in Figure 3-(1), and its execution graph is shown in Figure 3-(2).

Consider a node $< b_2, c_2 >$ in the execution graph. Assume that program1 executed its loop n times and program2, m times. Then,

$C(s) = 1$;

$np(s) = n + m + 2$;

$nv(s) = n + m$;

$ne(s) = n + m + 2$;

hold. Since

$$ne(s) \neq \min [\, np(s), C(s) + nv(s) \,] ,$$

the relation (1) does not hold, and the node $< b_2, c_2 >$ is not realizable.

Applying the same method to the other nodes, we will easily come to the conclusion that the nodes and arcs in an interior part of a shaded square in Figure 3-(2) cannot be realized in any computation. (Note that the nodes and arcs on the edges of the square are realizable.)

When a sequential program includes several conditional branches, it is rather difficult to determine the values of np(s), nv(s) and ne(s). In this kind of program, the case analysis may be useful for calculating such values.

4-2 For Co-operating Sequential Programs without P- and V-operations

Consider co-operating sequential programs which do not have any P- and V-operation. Each of the sequential programs may have several conditional branches, and each branch exit is associated with an execution condition. Since the conjunction of the execution conditions for branch exits of the sequential programs is not always true, there may be some unrealizable nodes and arcs in its execution graph.

Values of variables which appear in a predicate of a conditional branch of a sequential program may be varied by any other sequential program. Therefore, an execution condition for a branch exit must carefully be determined.

In order to describe such an execution condition, we will introduce the following modality [4].

Definition 4.1

(1) modal operator $\Box$:

$\Box p$: it is always true that p.

(2) modal operator $\Diamond$:

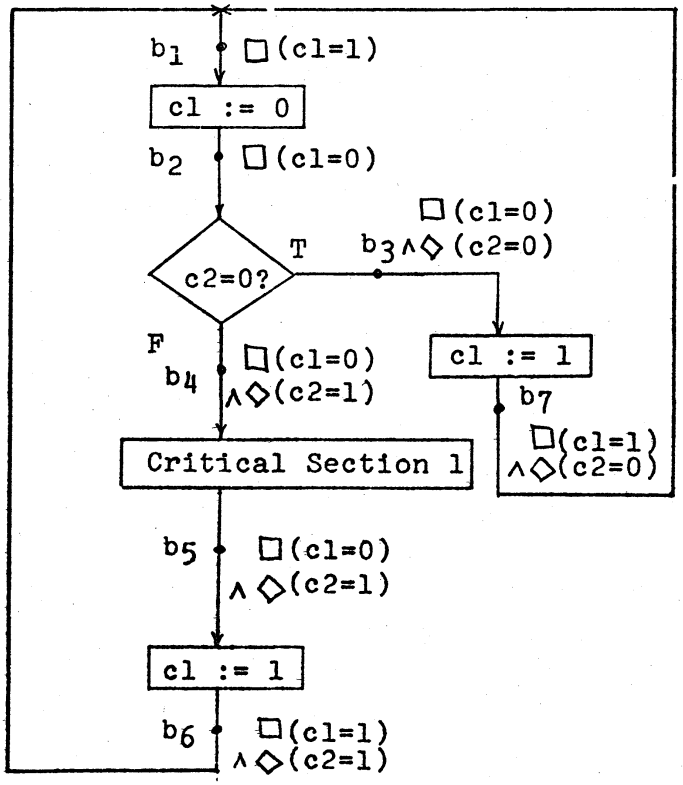
$\Diamond p$: it was true some time in the past that p.

Now let's consider the following example, and use the above modal operators to determine the realizability of nodes and arcs.

Example 4.2

The co-operating sequential programs shown in Figure 4 are a solution to the mutual exclusion problem. (Strictly speaking, it is not a correct solution.) They do not use any P- and V-operation. The value of c1 is varied only in P1, and the value of c2 is varied only in P2. For each branch, its execution condition with modal

P1:



P2:

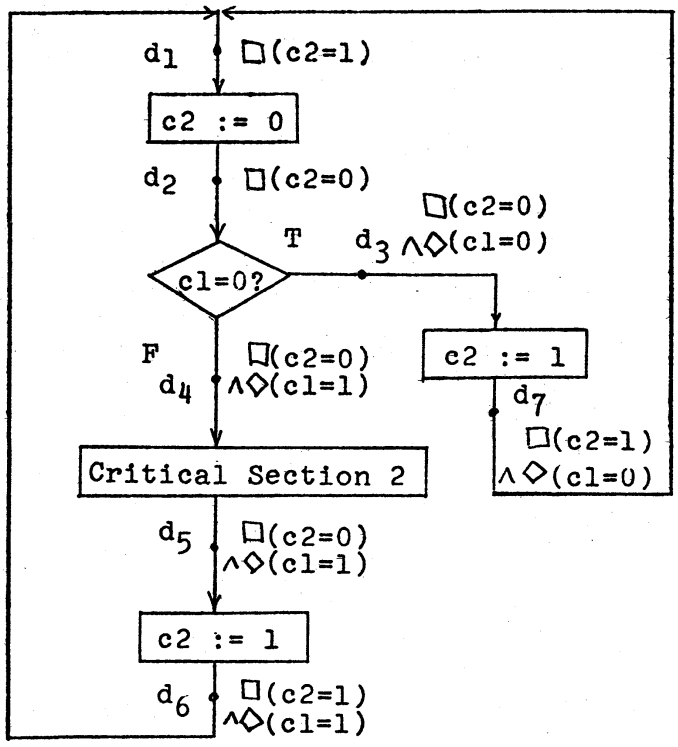


FIG.4 Mutual exclusion problem

operators is assigned by the following rules.

Rule 1

$\Box$ -operator is associated with a predicate on a variable whose value is varied only in my program.

Rule 2

$\Diamond$ -operator is associated with a predicate on a variable whose value is varied in your program.

In fact, an execution condition is a conjunction of predicates on several variables. When several branches join, an execution condition after the junction point is given by a disjunction of the execution conditions before the junction point. Since $\Diamond p \vee \Diamond \sim p = \Diamond(p \vee \sim p) = \Diamond T$ holds, the execution condition for b_1 is $\Box(c1=1)$, and that for d_1 is $\Box(c2=1)$.

A node in the execution graph of this co-operating sequential programs is given by $< b_i, d_j >$. In order to determine the realizability of a node $< b_i, d_j >$, we should investigate if the conjunction of both execution conditions for b_i and d_j can be satisfied. When determining the realizability of a node, we concurrently determine the realizability of arcs incident to or from this node. The following steps are applied to this example.

Step 1

Replace $(\Box p \wedge \Diamond p)$ by $\Box p$.

Step 2

A node with $(\Box p \wedge \Box q)$ can be realized if and only if

$$p, q \not\vdash \text{False} .$$

An arc incident to this node can be realized if and only if a node of the other end is realizable.

Step 3

A node with $(\Box p \wedge \Diamond \sim p) \wedge \Box q$ can be realized if and only if the following conditions hold.

1) $p, q \not\vdash$ False ;

2) Assume that b_i is associated with $\Box p$. Then there is at least one branch b_k with $\Box \sim p$, and from b_k to b_i there is a path which has no intermediate branch either with $\Box \sim q$ or $\Diamond \sim q$.

An arc incident to a node which is determined to be realizable in this step can be realized if and only if a statement corresponding to this arc can be executed without contradicting to $\Box q$.

Step 4

If both b_i and d_j are conditional branch exits, a node with $(\Box p \wedge \Diamond \sim p) \wedge (\Box q \wedge \Diamond \sim q)$ cannot be realized. An arc incident to or from this node cannot be realized either.

If both of them are not conditional branch exits, take the nearest conditional branch exits on the backward path from them, and investigate the realizability for these branch exits.

The followings are the results of the application of the above steps to several nodes.

$\langle b_4, d_4 \rangle$: unrealizable (by the first part of Step 4)

$\langle b_3, d_3 \rangle$: realizable (by Step 1 and Step 2)

$\langle b_3, d_4 \rangle$: realizable (by Step 1 and Step 3. You will find a path $b_1 \rightarrow b_2 \rightarrow b_3$.)

$\langle b_7, d_7 \rangle$: realizable (by the last part of Step 4. Instead of $\langle b_7, d_7 \rangle$, you may consider a pair $\langle b_3, d_3 \rangle$)

5. Deadlocks and Effective Deadlocks

Two kinds of anomalies, deadlocks and effective deadlocks, may exist in the dynamic behavior of co-operating sequential programs. These anomalies have much to do with a resource allocation problem in an operating system. This section will briefly discuss them with relation to an execution graph.

5-1 Deadlocks

It is easily understood that there is no deadlock danger in co-operating sequential programs without P- and V-operations. Since the pass of a P-operation in a program may be delayed until another program activates it through a V-operation, it is possible that some delayed P-operations will never be activated.

A deadlock is defined by using an execution graph as follows.

Definition 5.1 (deadlocks)

For given co-operating sequential programs S, a node n of its execution graph E_S is in a deadlock if and only if

(1) n is realizable ;

(2) there is at least one subset $S' \subseteq S$, and in an execution graph $E_{S'}$ of S' the node which corresponds to n has no realizable arc starting from it.

Example 5.1

Consider the following co-operating sequential programs.

```
begin
  semaphore  A,B ;
       A := B := 1;
  parbegin
    P1:  begin
           P(A) ;
           P(B) ;
           critical section 1 ;
           V(B) ;
           V(A) ;
           remainder of P1
         end ;
    P2:  begin
           P(B) ;
           P(A) ;
           critical section 2 ;
           V(A) ;
           V(B) ;
           remainder of P2
         end
  parend
end ;
```

The execution graph of this example is shown in Figure 5. The interior of the shaded part of this execution graph cannot be realized. Although a node $\langle b_2, c_2 \rangle$ is realizable, it has no arc starting from it. Then, $\langle b_2, c_2 \rangle$ is in a deadlock situation.

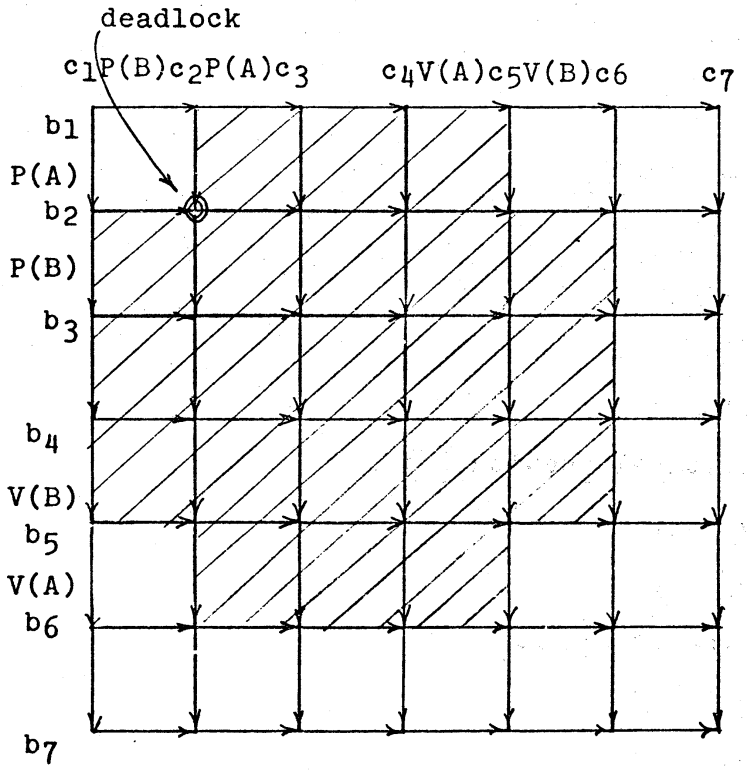


FIG. 5 An example of deadlock

5-2 Effective Deadlocks

Even if there is no danger of deadlocks, it is not assured that a computation will be done successfully. In an execution graph, it is possible that there is an infinite loop in which the objectives of some programs will never be finished.

Definition 5.2 (effective deadlocks)

Assume that each p_i of co-operating sequential programs $S = (p_1, p_2, \ldots, p_M)$ has its objective nodes. Assume also that there is no program whose progress is indefinitely delayed for lack of processors.

A directed cycle consisted of realizable nodes and arcs in an execution graph E_S is in an effective deadlock if and only if

(1) the execution along this cycle continues indefinitely;

(2) there is at least one program p_j whose objective node o_j will not appear as j-th element of the nodes on this cycle;

(3) this cycle has at least one node from which an arc starts to a node with o_j as its j-th element.

Example 5.2

Consider programs of Example 4.2, and slightly modify them as is shown in Figure 6. Its execution graph with realizable nodes and arcs is given in Fifure 7.

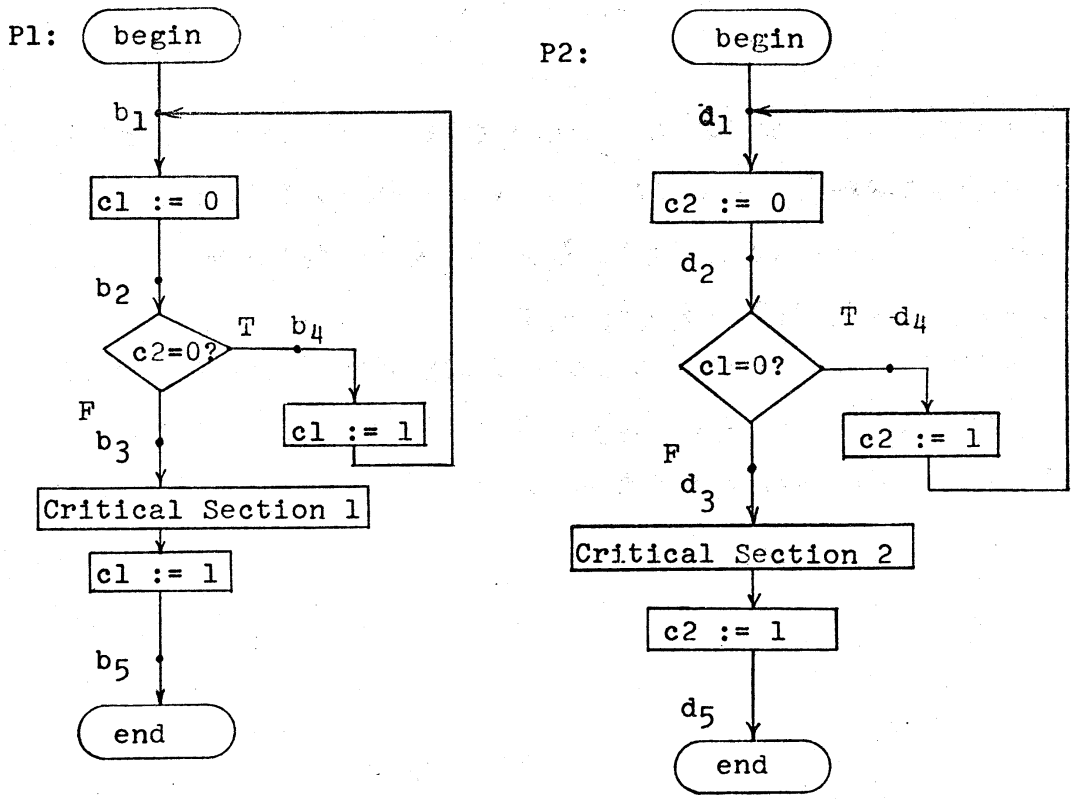


FIG. 6 Mutual exclusion problem

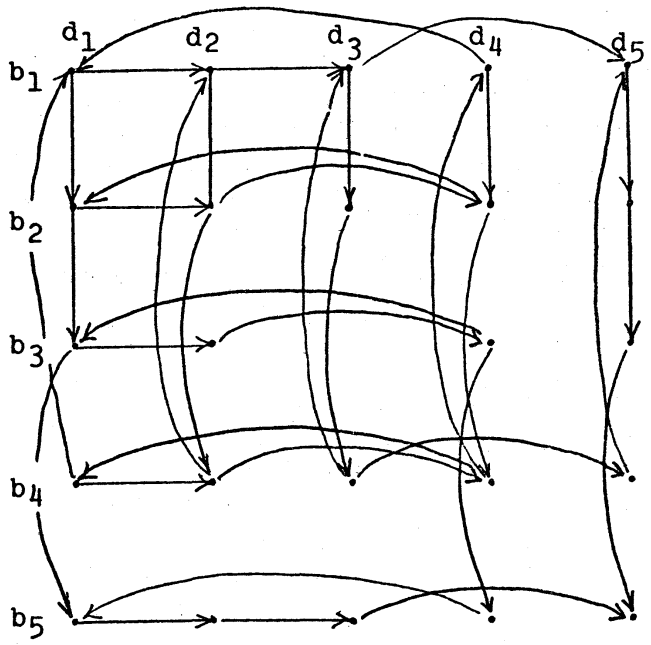


FIG. 7 Execution graph of FIG.6

The objective node of P1 is b_3, and that of P2 is d_3. In this execution graph, there is a directed cycle given as follows.

$$\langle b_1,d_1\rangle \rightarrow \langle b_1,d_2\rangle \rightarrow \langle b_2,d_2\rangle \rightarrow \langle b_2,d_4\rangle \rightarrow$$
$$\langle b_4,d_4\rangle \rightarrow \langle b_4,d_1\rangle \rightarrow \langle b_1,d_1\rangle$$

In this cycle, d_3 does not appear, although there is a realizable arc between $\langle b_1,d_2\rangle$ and $\langle b_1,d_3\rangle$. Therefore, this cycle is considered to be in an effective deadlock.

6. Verification Steps Revised

The verification of correctness of given co-operating sequential programs S by using its execution graph is given by the following steps.

Step 1

Construct a direct product of the flow graphs of given programs S.

Step 2

Remove all the unrealizable nodes and arcs from the direct product constructed in Step 1, and let it be called E_S.

Step 3

Check whether there is a danger of deadlocks or effective deadlocks in E_S. If there is a deadlock or an effective deadlock, S is not correct.

Step 4

Apply the verification theorem to E_S.

7. Concluding Remarks

An execution graph was proposed to be used in the verification of given co-operating sequential programs. It is constructed as a direct product of given sequential programs, and the number of its nodes and arcs tends to become large. The construction algorithm of an execution graph, however, is so simple that it can easily be mechanized on a computer.

An execution graph is defined for co-operating sequential programs with fixed number of control flows. This can also be defined for a more general parallel programs with variable number of control flows.

Modal operators were introduced in Section 4 in order to determine the realizability of nodes and arcs in an execution graph. It should be investigated whether these operators can directly be applied to key assertions used in the verification theorem.

References

[1] Dijkstra,E.W. Co-operating Sequential Processes, Programming Languages, edited by Genuys,F., Academic Press, New York, 1968

[2] Floyd, R.W. Assigning Meanings to Programs, Proc. of a Symposium in Applied Mathematics, Vol.19, AMS, 1967

[3] Habermann, A.N. Synchronization of Communicating Processes, Proc. of 3rd ACM Symposium on Operating Systems Principles, Oct. 1971

[4] Snyder, D.P. Modal Logic and Its Application, Van Nostrand Reinhold Co., New York, 1971