Synthesis of Mutual Exclusion Solutions Based on Binary Semaphores

R. T. JACOB AND I. P. PAGE

Abstract—A graphical form of the mutual exclusion problem is considered in which each vertex represents a process and each edge represents a mutual exclusion constraint between the critical sections of the processes associated with its endpoints. An edge semaphore solution for mutual exclusion problems is defined, and those graphs which are edge solvable are characterized in terms of both a forbidden subgraph and a graph grammar. Finally, an efficient algorithm is given which generates the entry and exit sections for all processes in an edge-solvable problem.

Index Terms—Binary semaphore, concurrent sequential processes, deadlock, edge semaphore, graph grammar, mutual exclusion.

I. INTRODUCTION

In a computer system with several active processes sharing common resources, a means of mutual exclusion is required to ensure that only one process at a time is able to use a resource or modify shared information. A critical section of a process is a code fragment that accesses shared resources or modifies shared storage. The original mutual exclusion problem is to encode the critical sections of all processes, or to surround their critical sections with code, in such a way that,

1) when a process is in its critical section, no other process may enter its critical section;
2) when no process is in its critical section, any process wanting to enter its critical section must be allowed to do so;
3) when several processes compete to enter their critical sections at the same time, one of them must be allowed to enter its critical section within a finite time; and
4) there must always be at least one process able to proceed.

Well-known examples of mutual exclusion problems are the 'readers and writers problem' and Dijkstra's 'five dining philosophers problem.'

In this paper, we discuss a graphical method for describing generalized mutual exclusion problems. With this method, a set of processes is represented by an undirected graph. Each vertex of the graph corresponds to a process with a single critical section, and the presence of an edge between two vertices indicates a mutual exclusion constraint between the critical sections of the associated processes. In our generalized version of the problem, several processes may be in their critical sections simultaneously, provided there are no constraints between any of their critical sections.

We consider solutions to mutual exclusion problems that involve coding the processes with entry and exit sections inserted before and after each critical section. By making the restriction that each entry and exit section be a single call to one of the synchronization primitives, Lipton [1] showed that Dijkstra's P and V primitives can solve only complete graphs. Henderson and Zalcestein [2] found that the PVchunk primitives can solve a larger class of mutual exclusion graphs, but not all. In each of these studies, the entry and exit sections consist of single kernel calls. This restriction is not practical in a systems programming environment, so it is pertinent to ask what problems can be solved using more complex entry and exit sections.

Here, we investigate a method of synthesizing semaphore solutions for a class of graphs which we call "edge-solvable" graphs. This class of graphs arises from our observation that, in the literature, almost all semaphore solutions to mutual exclusion problems fall into one of two classes. In the first, the semaphores may be pictured as residing on the edges of a graph. We call these edge semaphore solutions. In the other, the semaphores may be pictured as residing at the vertices of the graph. These solutions are invariably simulations of monitors using semaphores. Hence, we call them monitor semaphore solutions. Monitor solutions are beyond the scope of this paper. Interested readers should consult [3] for a full discussion.

In Sections IV and V, we characterize those mutual exclusion graphs that have edge semaphore solutions and, in Section VII, give an algorithm for finding solutions for these graphs.

A semaphore is called a binary semaphore if it may take only the two values 0 and 1 and is called a weak semaphore if it obeys the rule that, when a V(x) operation is done while there is any process suspended on x, one such process will be chosen nondeterministically to continue. Only weak binary semaphores are assumed in this work. Since weak semaphores are generally prone to starvation, the solutions given here do not avoid the starvation problem. This difficulty can be remedied by imposing various queueing disciplines on the system. A discussion of this problem can be found in [3].
Although in recent years many authors have rejected global synchronization implemented with shared variables in favor of distributed control implemented with messages and remote procedure calls, it is still the case that many systems programming environments provide little more than semaphores in a Pascal-like high-level language setting. There are excellent distributed solutions in the literature to the problems addressed here \cite{4, 5}, and we do not claim to better them. Our aim is to explore the range of problems that can be solved with a certain set of primitives.

II. BASIC DEFINITIONS

A mutual exclusion graph \((V, E)\) is a finite, connected, undirected graph. Each vertex in \(V\) is uniquely labeled with the name of an associated process. For each pair of vertices in \(V\), the set \(E\) contains an edge connecting the two vertices if and only if there is a mutual exclusion constraint between the processes associated with them.

Throughout the remainder of this paper, the term ‘‘graph’’ will be used in place of the phrase ‘‘mutual exclusion graph.’’ In addition, the distinction between a vertex and its associated process will be blurred, so that we will often say, for example, that ‘‘vertex \(v\) is critical’’ when we really mean ‘‘the process \(v\) is in its critical section.’’ Several standard graph theory terms are not defined here. See \cite{6} for definitions.

In a graph, \(N(p)\) denotes the set of neighbors of a vertex \(p\):
\[ N(p) = \{ t \in V : (p, t) \in E \}. \]

We say that two vertices \(p\) and \(q\) are similar if \(N(p) = N(q)\). A graph is connected if and only if \(N(p) = N(q)\). If two vertices are similar, but not strongly similar, we call them weakly similar.

A simple synchronization block consists of a single \(P\) or \(V\) operation. A compound synchronization block consists of a sequence of instructions of the form
\[ P(m); \text{test and update sequence}; V(m) \]
where the semaphore \(m\) does not appear within the block’s test and update sequence. In a compound synchronization block, the semaphore \(m\) is called the gate semaphore of the synchronization block.

In a compound synchronization block, the test and update sequence is typically an ‘‘if’’ or ‘‘case’’ statement that tests the state of the system and, if necessary, executes a \(P\) or \(V\) operation on one or more semaphores different from the gate semaphore \(m\).

Two particularly important synchronization blocks are the \(PP\) and \(VV\) macros \cite{7}, whose encoding is shown below.

\[ c = 0 : \text{integer}; \]
\[ m = 1, s = 1 : \text{semaphore}; \]

\[ PP(c,s,m): P(m); \\
   c := c + 1 \\
   \text{if } c = 1 \text{ then } P(s); \]
\[ V(m); \]

\[ VV(c,s,m): P(m); \\
   c := c - 1; \\
   \text{if } c = 0 \text{ then } V(s); \]
\[ V(m); \]

An entry or exit section is a sequence of synchronization blocks. A process threads its entry section when it wants to enter its critical section and its exit section upon leaving its critical section.

In this paper, we will assume that all processes have the same form:
\[ \text{repeat} \\
   \text{noncritical section} \\
   \text{entry section} \\
   \text{critical section} \\
   \text{exit section} \\
   \text{until done}. \]

A solution to a problem is a coding of the processes that implements the required mutual exclusion constraints. It is necessary not only that the processes exclude each other in the manner indicated by the graph, but also that no process be blocked unless a graph neighbor is in its critical section. We also require that the code for solutions obey the following rules.

1) There are two disjoint sets \(\text{Sem}\) and \(\text{St}\) of variables with the property that a variable occurs in the entry or exit section of a process if and only if it is in \(\text{Sem} \cup \text{St}\). Furthermore, no variable in \(\text{Sem} \cup \text{St}\) may appear in the critical or noncritical section of any process. The elements of the set \(\text{Sem}\) must be all weak binary semaphores. The elements of the set \(\text{St}\), called state variables, must all be integer variables.

2) The first statement executed by a process entering a compound synchronization block must be a \(P\) operation on its gate semaphore, and the last statement executed in a compound synchronization block must be a \(V\) operation on the gate semaphore.

Condition 2) and the definition of entry and exit sections together imply that the synchronization blocks of the entry or exit section of a process are linearly ordered so that, if \(B_i\) and \(B_j\) are synchronization blocks in the entry (exit) section of process \(p\) such that \(i < j\), then any time \(p\) threads its entry (exit) section it must thread both \(B_i\) and \(B_j\), and must enter \(B_i\) before \(B_j\).

A semaphore solution to a mutual exclusion problem is called an edge semaphore solution when, for each pair of processes \(p\) and \(q\) which are graph neighbors, there are a semaphore \(x\) and synchronization blocks \(S_p\) in \(p\)’s entry section and \(S_q\) in \(q\)’s entry section such that, whenever \(u\) is either one of \(p\) or \(q\) and \(v\) is the other, then

1) when \(u\) leaves \(S_u\), the value of \(x\) must be 0 and must remain 0 until \(u\) enters its critical section;

2) the semaphore \(x\) appears in no other synchronization block of the entry section of \(u\) or \(v\); and

3) if \(v\) has threaded \(S_v\), but has not yet reached the end of its critical section when \(u\) enters \(S_u\), either \(v\) leaves its critical section while \(u\) is in \(S_u\) or \(u\) becomes blocked on some semaphore in \(S_v\).
The semaphore $x$ is called a $v$-synchronizer of the process $u$, and the block $S_x$ is called a $v$-synchronization block of $u$.

A graph is said to be edge solvable whenever it is the graph of a mutual exclusion problem that has an edge semaphore solution.

III. PRINCIPLES FOR SOLUTIONS

Following Dijkstra [8], we make the following five assumptions:

1) Two graph neighbors are never critical at the same time.
2) If a process is in its entry or exit section, it must either halt on a semaphore or exit that section with finite time (the progress rule). In particular, two processes must not be able reach a kind of livelock situation, in which each can proceed, but neither is willing to commit itself until the other does.
3) The behavior of a process while in its noncritical section can have no effect on the mutual exclusion protocol. In particular, if a graph has a solution and a node is deleted from the graph, the solution must be correct for the remainder of that graph.
4) Deadlocks are not allowed.
5) There is no priviledged process which is specially coded or given priority when necessary to avoid deadlocks or other problems. In Dijkstra’s solutions, this rule results in identical encodings for the entry and exit sections of all processes. In the generalized problem discussed here, identical encoding for all processes is impossible. However, our solution is symmetric in the sense that, if two processes are similar, then they have identical entry and exit encodings, and if two processes are symmetric in the sense that there is an isomorphism of the graph onto itself that takes one process onto the other, then their entry and exit encodings are similar in the sense that there is a one-to-one mapping of $\text{Sem} \cup \text{St}$ onto itself that changes one encoding into the other.

In addition, we impose the following conditions.

6) Only weak binary semaphores may be used.
7) A process shall be blocked or remain blocked only if a graph neighbor is in its critical section. While not explicitly stated by Dijkstra, this condition is implicit in his solutions.
8) Solutions involving busy waiting are not allowed. Again, this condition is implicit in Dijkstra’s solutions since one motivation for the introduction of semaphores was the avoidance of busy waiting.
9) If $t_1$ and $t_2$ are times when all processes are in their noncritical sections and $z \in \text{Sem} \cup \text{St}$, then the value of $z$ at time $t_1$ equals the value of $z$ at time $t_2$. In particular, the system has no memory of which processes have been in their critical sections or of how many times they have been there.

IV. THE FORBIDDEN SUBSYSTEM

A graph $G' = (V', E')$ is said to be a subsystem of a graph $G = (V, E)$ if $V' \subset V$ and $E' = \{(p, q) : p, q \in V' \text{ and } (p, q) \in E\}$.

In this section, we will show that, for any graph $G$, if the graph $F$ shown in Fig. 1 is a subsystem of $G$, then $G$ is not edge solvable.

Lemma 1: In any edge semaphore solution for the graph $T$ of Fig. 2, the first synchronization block $B$ in $q$’s entry section that contains either a $p$- or an $r$-synchronizer must contain both a $p$- and an $r$-synchronizer.

Proof: Suppose that the first synchronization block $B$ of $q$’s entry section that contains either a $p$- or an $r$-synchronizer contains a $p$-synchronizer, but no $r$-synchronizer. Suppose all processes are in their noncritical sections. Let process $q$ thread its entry code to the point where it is about to enter its first synchronization block that contains an $r$-synchronizer. Process $q$ has already threaded $B$, and some $p$-synchronizer has the value 0. Now, let process $r$ thread its entry code and enter its critical section. Process $q$ continues and becomes blocked on some semaphore in its next synchronization block. If process $p$ now attempts to thread its entry code, it must at some point do a $P$ operation on a semaphore with value 0 and become blocked. This violates our stipulation that no process be blocked unless a graph neighbor is critical. We have a similar contradiction when process $q$’s first synchronization block that contains a $p$- or $r$-synchronizer contains an $r$-synchronizer, but no $p$-synchronizer. Therefore, $q$’s first synchronization block $B$ that contains either a $p$- or an $r$-synchronizer must contain both a $p$-synchronizer and an $r$-synchronizer.

End of proof

Lemma 2: The synchronization block $B$ of Lemma 1 can have no gate semaphore in common with any synchronization block in the entry sections of $p$ and $r$.

Proof: Suppose that some synchronization block in $p$’s entry section has the same gate semaphore $m$ as $B$. Suppose all processes are in their noncritical sections. Let process $r$ thread its entry code and enter its critical section. Now, let process $q$ begin to thread its entry code and become blocked on some semaphore in $B$. The semaphore $m$ must now be 0. If process $p$ now attempts to thread its entry code, it must become blocked on $m$ or some semaphore that precedes $m$. This violates our stipulation that no process be blocked unless a graph neighbor is critical. We have a similar contradiction if some synchronization block in $r$’s entry section has a gate semaphore in common with $B$. Therefore, $B$ shares no gate semaphore with any synchronization block in the entry sections of $p$ and $r$.

End of proof

Lemma 3: If, when all processes are in their noncritical sections, process $q$ begins to thread its entry code, then the first $p$- or $r$-synchronizer $x$ it encounters is both a $p$- and an $r$-synchronizer.

Proof: If all processes are noncritical when process $q$ begins to thread its entry code, it must encounter both a first $p$-synchronizer $x$ and a first $r$-synchronizer $y$ in block $B$. Suppose that the semaphore $x$ is different from the semaphore $y$ and that all processes are in their noncritical sections. Let process $q$ thread its entry code until it reaches block $B$. By the progress rule, in block $B$, process $q$ must
reach a point where one of x and y is 0, say x, and q is about to do a P operation on the other, y. Let process r now attempt to enter its critical section. Since none of the synchronization blocks in r’s entry section has a gate semaphore in common with B and q has not done a P operation on any of its r-synchronizers, r successfully enters its critical section. The semaphore y now is 0. Process q now continues and becomes blocked on y. If process p now attempts to thread its entry section, it must become blocked on x or some semaphore that precedes x. This violates our stipulation that no process be blocked unless a graph neighbor is critical. We have a similar contradiction in the case where y comes before x. Therefore, x and y must be the same semaphore.

**End of proof**

**Lemma 4:** The semaphore x of Lemma 3 must be contained in the first q-synchronization blocks of the entry sections of both p and r.

**Proof:** Suppose that x is not contained in the first q-synchronization block of p’s entry code and that all processes are in their noncritical sections. Let process q thread its entry code until it arrives at the point where it is about to do a P operation on the semaphore x. Let process p now thread its entry code until it arrives at the point where it is about to do a P operation on its first q-synchronization z. Process q now does a P operation on x. Process p continues, does a P operation on z, and goes on to the point where it becomes blocked on x. Process q now continues and becomes blocked on z or some semaphore that precedes z. This violates our stipulation that all solutions be deadlock free. We get a similar contradiction if we assume that x is not in the first q-synchronization block of r’s entry section.

**End of proof**

**Lemma 5:** The first q-synchronization blocks of the entry sections of p and r must have a common gate semaphore.

**Proof:** If not, suppose that all processes are in their noncritical sections. Let process q thread its entry code until it arrives at the point where it is about to do a P operation on the semaphore x. Now, let process p thread its entry code until it arrives at the point where it is about to do a P operation on x. Process r now threads its entry code and enters its critical section. The value of x must still be 1 since, otherwise, if p proceeded it would be blocked. Process q now continues, does a P operation on x, and becomes blocked on some r-synchronizer. The value of x is now 0. Process p continues and becomes blocked on x. This violates our stipulation that no process be blocked unless a graph neighbor is critical.

**End of proof**

**Theorem 1:** The graph F is not edge solvable.

**Proof:** Suppose there is a solution for F. By the lemmas, processes p and q have a common q-synchronizer x which is contained in q-synchronization blocks with a common gate semaphore $m_p$, and q and s have a common r-synchronizer y which is contained in r-synchronization blocks with a common gate semaphore $m_q$. Furthermore, x and y must be in the first synchronization block of q that contains the relevant types of semaphores.

Suppose all processes are in their noncritical sections. Let p enter its critical section. The semaphore x now has the value 0. Let q try to enter its critical section. It must become blocked on some semaphore in its synchronization block that contains x. The semaphore $m_p$ now has the value 0. If s now tries to enter its critical section, it must become blocked on some semaphore. This violates our stipulation that no process be blocked unless some graph neighbor is critical. Therefore, there is no edge semaphore solution for F.

**End of proof**

**Corollary to Theorem 1:** If G is any graph having F as a subsystem, then G is not edge solvable.

**Proof:** Any subsystem of an edge-solvable graph is edge solvable. Hence, no edge-solvable graph can contain F.

**End of proof**

**V. BUILDING GRAPHS**

In this section, we prove that a graph is edge solvable if and only if it does not contain F. In the process, we show implicitly how every edge-solvable graph can be built up from a trivial graph in a simple regular way. In Section VII, we will present an explicit algorithm for decomposing an edge-solvable graph to obtain an edge solution for it that, in effect, uses this method in reverse.

**Theorem 2:** Every complete multipartite graph is edge solvable.

**Proof:** Suppose G is a complete multipartite graph with partitions $V_1, \ldots, V_n$. Let $m_1, \ldots, m_n$ and x denote binary semaphores with initial value 1. Let $c_1, \ldots, c_n$ denote integer state variables with initial value 0. Encode the entry section of each process in $V_i$ ($1 \leq i \leq n$) as $PPV(c_i, x, m_i)$ and its exit section as $VV(c_i, x, m_i)$. The resulting system is easily seen to be correct.

**End of proof**

We call an edge semaphore solution for a graph G a PPVV solution, if for every vertex p of G,

1) the entry section of process p contains only P and PP synchronization blocks, and its exit section contains only V and VV synchronization blocks;
2) no semaphore is contained in two different synchronization blocks of p’s entry section, or in two different synchronization blocks of p’s exit section;
3) P(x) is a synchronization block in p’s entry section if and only if $V(x)$ is a synchronization block in p’s exit section; and
4) $PPV(c, x, m)$ is a synchronization block in p’s entry section if and only if $VVV(c, x, m)$ is a synchronization block in p’s exit section.
Theorem 3: If \( G = (V, E) \) has a PPVV solution, \( p \in V, q \) is a vertex not in \( V, V' = V \cup \{q\} \), and \( E' = E \cup \{(q, r) : (p, r) \in E\} \), then \( G' = (V', E') \) also has a PPVV solution.

Proof: If, in the PPVV solution for \( G \), the entry section for \( p \) contains a synchronization block of the form \( P(x), \) let \( m \) be a semaphore and \( c \) be an integer state variable not used in the solution for \( G \). Replace the block \( P(x) \) by the block \( PP(c, x, m) \) and replace the block \( V(x) \) in \( p \)'s exit section by \( VV(c, x, m) \). In \( G' \), let both \( p \) and \( q \) have these modified entry and exit sections, and let all other vertices have the same entry and exit code they have in the solution for \( G \). The result is a PPVV solution for \( G' \).

End of proof.

Theorem 4: If \( G = (V, E) \) has a PPVV solution, \( S \) is a set of similar vertices of \( G \), and \( E' = E \cup \{(p, q) : p, q \in S\} \), then \( G' = (V, E') \) also has a PPVV solution.

Proof: Let \( x \) be a semaphore not used in the PPVV solution for \( G \). For each vertex \( p \) in \( S \), add \( P(x) \) after the last synchronization block in the entry section for \( p \), and \( V(x) \) before the first synchronization block in the exit section for \( p \). In \( G' \), let all elements of \( S \) have these modified entry and exit sections, and let all other vertices have the same entry and exit sections they have in the solution of \( G \). The result is a PPVV solution for \( G' \).

End of proof.

Theorem 5: Suppose that \( G \) is a graph, \( F \) is not a subsystem of \( G \), and \( G \) contains two similar vertices. If \( G' \) is a graph having \( G \) as a subsystem, containing exactly one vertex not in \( G \) and not having \( F \) as a subsystem, then \( G' \) contains two similar vertices.

Proof: Suppose that \( p \) is the vertex of \( G' \) that is not in \( G \) and that \( q \) and \( r \) are similar vertices of \( G \). If \( p \) is joined to both \( q \) and \( r \), or if it is joined to neither of them, then \( p \) and \( r \) are similar in \( G' \). Suppose, then, that \( p \) is joined to \( q \), but not to \( r \) in \( G' \). Either \( a \) \( q \) and \( r \) are joined in \( G' \) or \( b \) they are not.

Case \( a \): The vertices \( q \) and \( r \) are joined in \( G' \). Suppose that \( G' \) does not contain two similar vertices. There must be a vertex \( s \) that is joined to \( r \), but not to \( p \). Note that \( s \) must also be joined to \( q \) since \( q \) and \( r \) are similar in \( G \). See Fig. 3(a).

Since \( r \) and \( s \) are not similar in \( G' \), there is a vertex \( t_1 \) that is joined to \( r \) (and hence to \( q \)), but not to \( s \). The vertices \( t_1 \) and \( p \) cannot be joined because, if they were, then the subsystem \( p_1s \) of \( G' \) would be isomorphic to \( F \). (For the rest of the proof, a subsystem of \( G' \) isomorphic to the forbidden subsystem \( F \) will be called a "forbidden subsystem.") See Fig. 3(b).

Since \( s \) and \( t_1 \) are not similar in \( G' \), there is a vertex \( t_2 \) that is joined to one of them, but not to the other. Without loss of generality, we may assume that \( t_1 \) and \( t_2 \) are joined. The vertex \( t_2 \) must be joined to \( q \) and \( r \), but not to \( p \). Since \( t_2 \) and \( t_1 \) are not similar in \( G' \), there is a vertex \( t_3 \) that is joined to one of them (we may assume \( t_2 \)), but not to the other. See Fig. 3(c).

We will finish the proof of Case \( a \) by induction. Suppose that \( n > 1 \) and that \( t_1, t_2, \ldots, t_n \) is a sequence of vertices such that
1) for all \( i \) \((1 \leq i \leq n)\), \( t_i \) is joined to \( q \) and \( r \), but not to \( p \) or \( s \);
2) for all \( i \) \((1 \leq i \leq n)\), \( t_i \) is joined to \( t_{i-1} \) and to all \( t_j \) \((j \text{ even}, 1 < j < i)\), but to no other \( t_k \) \((1 \leq k < i)\);
3) if \( n \) is even, then there is a vertex \( t_{n+1} \) that is joined to \( t_n \), but not to \( t_{n-1} \); and
4) if \( n \) is odd, then there is a vertex \( t_{n+1} \) that is joined to \( t_n \), but not to \( t_{n-2} \).

It is easy to see that \( t_{n+1} \) cannot be joined to \( s \), but must be joined to \( q \) and \( r \), and cannot be joined to \( p \). Note that if \( n \) is odd and greater than 2, then \( t_{n+1} \) must be joined to \( t_{n-1} \) since \( t_{n+1} t_{n-1} t_{n-2} \) would be a forbidden subsystem otherwise. However, for no \( n \) can \( t_{n+1} \) be joined to any \( t_i \) \((i \text{ odd}, 1 \leq i < n - 2)\) since \( t_{n+1} t_{n-1} t_{n-2} \) would be a forbidden subsystem when \( n \) is even and \( t_{n+1} t_{n-1} t_{n-2} \) would be such a subsystem when \( n \) is odd. Furthermore, \( t_{n+1} \) must be joined to \( t_i \) \((i \text{ even}, 1 < i < n - 1)\) since \( t_{n+1} t_{n-1} t_{n-2} \) would be a forbidden subsystem otherwise. In the subgraph of \( G' \) constructed so far, \( t_n \) and \( t_{n+1} \) are similar if \( n \) is even, and \( t_{n-1} \) and \( t_{n+1} \) are similar if \( n \) is odd. Hence, if \( n \) is even, there is a vertex \( t_{n+2} \) that is joined to one of \( t_n \) and \( t_{n+1} \), say \( t_{n+1} \), but not to the other, and if \( n \) is odd, there is a vertex \( t_{n+2} \) that is joined to one of \( t_{n-1} \) and \( t_{n+1} \), say \( t_{n+1} \), but not to the other.

Now, if \( n = 2 \), the induction hypothesis is satisfied. Hence, there exists an infinite sequence \( t_1, t_2, \ldots \) of dis-
tinct vertices of \( G' \). This is impossible. Therefore, \( G' \) must contain two similar vertices.

*Case b)*: The vertices \( q \) and \( r \) are not joined in \( G' \). Let \( H \) be the graph that results from adding the edge \((q, r)\) to \( G \) and \( H' \) be the graph that results from adding that edge to \( G' \). The vertices \( q \) and \( r \) are similar in \( H \), and neither \( H \) nor \( H' \) contains \( F \) as a subsystem. Since the graphs \( H \) and \( H' \) satisfy the conditions of Case a), \( H' \) must contain two similar vertices.

From the proof of Case a), \( p \) and \( r \) are similar in \( H' \), \( r \) and \( s \) are similar in \( H' \), or two vertices \( u \) and \( v \) different from \( q \) and \( r \) are similar in \( H' \). In the first instance, \( p \) and \( q \) must be similar in \( G' \); the second is impossible since \( G' \) would contain the forbidden subsystem \( pqrst \); and in the third, \( u \) and \( v \) are similar in \( G' \). In any case, \( G' \) contains two similar vertices.

**Corollary to Theorem 5**: Every nontrivial graph that does not contain the subsystem \( F \) has two similar vertices.

**Proof**: Note that the unique graph of order 2 has two similar vertices and use induction on the order of the graph.

**Theorem 6**: If \( G \) is a graph, then the following three statements are equivalent.
1. \( G \) is \( PPVV \) solvable.
2. \( G \) is edge solvable.
3. \( G \) does not contain the subsystem \( F \).

**Proof**:
1) \( \Rightarrow \) 2): If \( G \) is \( PPVV \) solvable, then it is obviously edge solvable.
2) \( \Rightarrow \) 3): By the Corollary to Theorem 1, if \( G \) is edge solvable, then it does not contain \( F \) as a subsystem.
3) \( \Rightarrow \) 1): We will prove by induction that every graph that does not contain \( F \) as a subsystem is \( PPVV \) solvable.

Note that the unique graphs of orders 1 and 2 do not contain the subsystem \( F \) and are \( PPVV \) solvable.

Now suppose that every graph of order \( n \) \((n > 1)\) that does not contain the system \( F \) is \( PPVV \) solvable and that \( G \) is a graph of order \( n + 1 \) that does not contain the subsystem \( F \). By the Corollary to Theorem 5, \( G \) contains two similar vertices \( p \) and \( q \). Let \( G' \) be the graph obtained by removing the vertex \( q \) from \( G \). \( G' \) does not contain the subsystem \( F \), so by hypothesis, it is \( PPVV \) solvable. By Theorems 3 and 4, when \( q \) is added to \( G' \) to obtain the graph \( G \), the result is also a \( PPVV \)-solvable graph.

End of proof

VI. GRAPH BUILDING RULES

Theorems 3 and 4 give us two useful graph transformation rules.

**Rule 1**—The Copy Rule: If \( G = (V, E) \) is a graph, \( p \in V \) and \( q \notin V \), build the new graph \( G' = (V', E') \) from \( G \) such that \( V' = V \cup \{ q \} \) and \( E' = E \cup \{ (q, r) : (p, r) \in E \} \). In the new graph \( G' \), \( N(q) = N(p) \).

**Rule 2**—The Join Rule: If \( G = (V, E) \) is a graph and \( S \) is a set of similar vertices of \( G \), build the new graph \( G' = (V', E') \) from \( G \) such that \( V' = V \cup S \) and \( E' = E \cup \{ (q, r) : (p, r) \in E \} \). In the new graph \( G' \), \( N(q) = N(p) \).

Rules 1 and 2 are almost the production rules for a graph grammar for the set of all edge-solvable graphs. However, the join rule is not in acceptable form since it contains, in effect, infinitely many separate rules. We may replace the join rule by the join2 rule, given below, to obtain an acceptable grammar.

**Rule 3**—The Join2 Rule: If \( G = (V, E) \) is a graph and \( p, q \in V \) such that \( N(p) = N(q) \), build the new graph \( G' = (V', E') \) from \( G \) such that \( V' = V \cup \{ (p, q) \} \).

It is easy to see that any graph that can be derived from the trivial graph using the copy and join rules may also be derived from that graph using the copy and join2 rules. See Fig. 4 for an example of the application of these production rules.

Another particularly important rule is the following.

**Rule 4**—The Copy-Join Rule: If \( G = (V, E) \) is a graph, \( p \in V \), and \( q_1, \ldots, q_n \notin V \), build the new graph \( G' = (V', E') \) such that \( V' = V \cup \{ q_1, \ldots, q_n \} \) and \( E' = E \cup \{ (p, q_i) : 1 \leq i \leq n \} \cup \{ (q_i, q_j) : 1 \leq i < j \leq n \} \cup \{ (q_i, r) : (p, r) \in E \text{ and } 1 \leq i \leq n \} \).

The rules copy and copy-join may also be used to derive all edge-solvable graphs from the trivial one. We will see in the next section that this pair of rules is particularly important in the decomposition of edge-solvable graphs.

VII. GRAPH DECOMPOSITION AND CODE GENERATION

Every edge-solvable graph always contains a pair of similar vertices, which may or may not be joined. In either case, in our solution the associated processes will have the same coding in their entry and exit sections. We use the reverse of the copy and copy-join rules to decompose the graph recursively. For example, if \( p \) and \( q \) are strongly similar vertices in \( G \), we can use the reverse of the copy rule to eliminate \( q \) and its adjoining edges. If \( p \) and \( q \) are weakly similar, we can use the reverse of the copy-join rule to eliminate \( q \) and its adjoining edges. Often, large groups of similar vertices may be eliminated in this fashion. To formalize the algorithm, we present a detailed procedure which generates a descriptor tree from the
A. The Decomposition Algorithm

Note that similarity is an equivalence relation on the vertices of G. Each equivalence class is a maximal collection of either strongly similar vertices (a strong similarity class) or weakly similar vertices (a weak similarity class).

In constructing G from the trivial graph, each strong similarity class can be built from one of its members by repeated applications of the copy rule, and each weak similarity class can be built from one of its members by repeated applications of the copy-join rule. The decomposition algorithm applies these rules in reverse to collapse equivalence classes to single nodes called supernodes. The process is applied recursively until a single vertex remains or there are no remaining similar vertices. In the latter case, the original graph (and the remaining collapsed graph) contains the forbidden subsystem F and cannot be solved by edge semaphores.

If S = {p₁, p₂, ⋯, pₙ} is an equivalence class of nodes, then the descriptor string for S is (p₁ p₂ ⋯ + pₙ) if S is a strong similarity class and is (p₁ | p₂ ⋯ pₙ) if S is a weak similarity class.

Descriptor strings have the following properties:

1) (p + q) = (q + p)
2) (p|q) = (q|p)
3) (p + (q + r)) = (p + q + r)
4) (p|(q|r)) = (p|q|r).

Note that the operators + and | do not have the distributive property.

In the algorithm below, if G = (V, E) is a graph, A denotes the adjacency matrix for G (Aᵢⱼ = 1 iff (i, j) ∈ E) and I denotes the identity matrix. Note that the members of a strong similarity class have identical rows in A, while the members of a weak similarity class have identical rows in A + I.

begin {Stage 1}
while (|V| > 1) AND (reduction is still possible) do
begin
for each nontrivial similarity class S do
begin
  collapse S to a supernode labeled by the descriptor string for S;
  |V| := |V| − |S| + 1;
end;
end;
begin {Stage 2}
while there is a child node c with the same type as its parent p do
begin
  remove the child c, and make all former children of c new children of p
end {Stage 2};
end {Stage 1};

The third stage of the algorithm generates the code for the entry and exit sections of all processes. For a substring of the form (X₁ | X₂ ⋯ | Xₙ), we choose a semaphore x, add P(x) to the entry sections of X₁, X₂, ⋯, Xₙ, and add V(x) to their exit sections.

For a substring of the form (X₁ + X₂ + ⋯ + Xₙ), the PP and VV primitives are used. The members of the class {X₁, X₂, ⋯, Xₙ} share a state variable c and a gate semaphore m, but use a synchronizer x inherited from their parent. PP(c, x, m) is added to their entry sections, and VV(c, x, m) is added to their exit sections.

In the Stage 3 algorithm, "#" denotes concatenation.

begin {Stage 3}
if root node is a leaf node then exit;
begin
for i = 0 to depth(descriptor tree) − 1 do
begin
  for each interior node X in level i do
  begin
    if type(X) = "|" then begin
      let x be a semaphore variable not used already;

    end;
  end;
end;
end {Stage 3};
entry-code (X) := entry-code(X) \# "P(x);"
exit-code(X) := "V(x);" \# exit-code(X)
end
else begin {type of node is "+"}
    let x be the semaphore in the last line of
    entry-code(X);
    let c be a state variable not already used;
    let m be a semaphore variable not already used;
    remove the last line "P(x);" from entry-
    code(X);
    remove the first line "V(x);" from exit-
    code(X);
    entry-code(X) := entry-code(X) \# "PP(c,x,m);";
    exit-code(X) := "VV(c,x,m);" \# exit-code(X)
end;
for each Y such that Y is a child of X do
begin
    entry-code(Y) := entry-code(X);
    exit-code(Y) := exit-code(X)
end
end {Stage 3};


**R. T. Jacob** received the B.A. and M.A. degrees in mathematics from the University of Texas, Austin, the M.S. degree in computer science from the University of Texas at Dallas, Richardson, and the Ph.D. degree in mathematics from Emory University, Atlanta, GA.

He has been on the faculty of the University of New Orleans, New Orleans, LA, and the University of Dallas, Irving, TX. He is presently an Assistant Professor of Computer Sciences at North Texas State University, Denton. His research interests include operating systems, distributed computing, and computer architecture.

**I. P. Page** received an honors degree in electrical engineering and the Ph.D. degree in computer science from Brunel University.

He worked in industry at ICL Research and Development Laboratories and Elliott Automation Radar Systems and then returned to Brunel University as a faculty member, staying ten years. He joined the University of Texas at Dallas, Richardson, in 1981. He has visited India twice as a United Nations ITU Visiting Expert in digital signal processing. He also spent several months at the NOAA Space Environment Laboratory in Boulder, CO, as a Visiting Scientist, developing software for its high-frequency ionospheric sounding equipment. He has published papers mostly on operating systems. His research interests include operating systems, computer architecture, and computer graphics.