To achieve complete generality, an implementation of the building and splitting process described in this study would have to be able to recognize arbitrarily complex conditions and resolve arbitrarily complex conjunctions of conditions and state information. A completely general implementation would, for example, not only have to recognize that $x > 5$ satisfies the condition $x \neq 3$ but not the condition $x \neq 6$, but also be able to deal effectively with conditions such as $x < y + z$. At present, SPLIT only handles conditions of the forms $x = c$ and $x \neq c$ where c is an integer or literal constant. This has been sufficient to deal with all branches on critical variables encountered in programs analyzed so far.

3.4 Folding

The initial state graphs produced by BUILD can be quite large; for example, the graphs representing operating system programs written in CDC6600 peripheral processor assembly language averaged about one node for every three lines of executable source code, and about 1.5 arcs per node. (These ratios will probably vary depending on the source language and nature of the program.) The splitting algorithm greatly increases the number of nodes. Thus both to conserve memory and to reduce the amount of work to be done by subsequent portions of TRACE, it is imperative to fold the graph as much as possible at each stage of the process. The main folding routine, FOLD, is in the main overlay of TRACE so that it can be called at any time. FOLD incorporates the two folding rules presented in Chapter II. When it is called, it considers one by one each node in the whole graph and attempts to apply the two folding rules repeatedly until no further combinations can be made with the node at hand. In addition, ϵ-arcs from a node to itself and one of a pair of identical

arcs between the same two nodes may be eliminated. When two nodes are combined, location labels and information regarding the value of variables attached to either of them are attached to the combined node, with duplications eliminated. Nodes connected by a conditional arc are never combined.

If any combinations were made in a pass through the whole graph, then FOLD again applies the rules to every node, and the process is repeated until a pass has been made in which no new combinations occurred.

FOLD is applied to the graph as soon as BUILD has finished and is usually able to cut its size in half. During the execution of SPLIT, FOLD may be called whenever memory gets crowded and is always called when SPLIT has finished. It has been found that at this point even very large object programs will be represented by graphs of less than 50 nodes. It therefore becomes practical in CLEANUP to apply folding rules with more sophisticated criteria for combining states:

(3) Any two nodes whose sets of outgoing arcs are identical, with respect to both labels and successor nodes, may be combined.

(4) If there is a closed path, no matter how long, consisting entirely of ϵ-arcs from a node back to itself, then all the nodes along this path may be combined into one node. (Notice that this is a generalization of rule 2; in rule 2 the length of the path is just two.)

When all four of the folding rules have been applied to every node in the graph without any new combinations occurring, CLEANUP outputs the result as the final graph and may also carry out the comparison algorithm if the user has supplied a prototype.

3.5 Actual Uses

Despite its experimental nature, TRACE has been put to several real uses on the UT2 operating system. It was run on six of the system programs which access the Job Status Table to verify that they all adhere to the semaphore protocol for reserving, accessing, and releasing the table. It was also used on a few programs to verify that they adhered to a similar protocol for reserving and releasing data channels. TRACE can also be used in its present form to verify that programs adhere to protocols for reserving disks and Extended Core Storage, for requesting half-tracks on these devices, etc.

As a matter of fact, using TRACE on these system programs revealed two clearcut protocol violations. It was discovered that program 1SJ could reserve the Job Status Table and then terminate without releasing it and that program PPR could reserve the system channel and then abort without releasing it. Unfortunately for the prestige of TRACE, each of these paths was already known to a system programmer, each could be taken only when the program had detected extreme error conditions, and neither path had ever been known to be taken under normal system conditions. Thus the protocol violations were not considered to be "important," and neither was corrected. Nevertheless, the discovery of real sequencing errors in real operating system programs by TRACE in the hands of a user not intimately acquainted with that operating system is a strong argument for the practicality of the analysis method presented in this study.

TRACE was also used to analyze portions of a different operating system, a locally modified version of Scope 3.3 for a CDC6600 at a Mobil Oil installation in Dallas. Only minor changes were necessary to enable TRACE to deal with these programs written in a different version of COMPASS. These consisted

mostly of adding executive routines to BUILD to respond to previously unencountered instructions. This experience should help justify the claim that the verification method is essentially language-independent.

TRACE has also proven to be very useful in simply revealing the actual (as opposed to apparent or flowchart) structure and ordering properties of programs for human inspection. In these cases, the final step, comparison with a prototype, is omitted, and the objective is just the final, split, and folded state graph of the object program. Such state graphs could be a valuable part of the documentation of a program. In one instance, a Computation Center programmer who had just acquired maintenance responsibility for a set of difficult programs from a departed colleague asked that TRACE be run on them simply to help him understand what they did under various circumstances and values of certain variables.

The state graphs produced by TRACE are also of use in connection with a major performance measurement and evaluation project currently in progress on the UT2 operating system. This project includes an event driven system trace [1, 17, 22]. The trace is used to produce directed graphs representing sequences of actions actually performed by programs including system programs. Some 50 or 60 different actions are detected, although not all of these will be performed by any one program. All of the actions recorded by the event trace can also be detected by TRACE in the source code of the same system programs. Thus the two directed graphs of a program's actions produced by these two different methods can be compared in order to help verify each method. Comparison of the graph produced from the source code with that from the program's actual behavior also serves to identify sections of code which are seldom or never used as well as heavily used sections which could be most profitably optimized. Finally, the directed graphs produced by TRACE have even

been used to pre-set the event trace graphs so that the event records could be used simply to put frequency information on the arcs of a graph which was already built rather than having to produce one.

Table 3.1 summarizes some information about eight representative executions of TRACE on UT2 system programs. These programs are all peripheral processor programs and all are important components of the UT2 operating system for the CDC6600. 1AJ, 1SJ, 1RJ, and 1TD are the primary overlays of the programs which advance, suspend, and resume jobs and of the tape driver, respectively. PFM and PPR are the permanent file manager and peripheral processor resident, respectively.

TABLE 3.1

SOME EXECUTIONS OF PROGRAM TRACE

Name	Lines	Modifications	Nodes	C P Time
1AJ	2541	25	42	32
			14	15
1RJ	784	15	16	6
			10	3
1SJ	619	6	11	3
PFM	1254	18	5	18
PPR	572	16	15	2
1TD	456	1	28	6

The first column of Table 3.1 gives the name of the object program. The second column gives the total number of lines in the program's source listing, of which perhaps 25% are comments so that, for example, 1AJ has 1800-1900 lines of executable code. The next three columns give the number of hand modifications which had to be made to the original source code before TRACE could be

run on the program, the number of nodes in the final folded state graph produced by TRACE, and the number of seconds of central processor time required by TRACE for the run. Some programs were analyzed more than once for different purposes. The first runs listed for 1AJ and 1RJ were for the event trace project so that BUILD was given a long list of "interesting actions," while the second run on each was to verify adherence to the Job Status Table reservation protocol so that only three actions in addition to the usual control statements were designated "interesting." In both cases the larger list of interesting actions given to BUILD naturally resulted in larger graphs.

3.6 Conclusions

TRACE was never intended to be a production program but rather an experiment, the purpose of which was to demonstrate that the method of program analysis presented here is practical. The two kinds of uses to which the program has been put, displaying the sequencing structure of many real programs for human analysis and actually verifying that some programs adhered, or failed to adhere, to a desired protocol, indicate that it is indeed practical, especially in view of TRACE's modest time and space requirements. TRACE required surprisingly little execution time. As can be seen in Table 3.1, running TRACE from start to final verification required about one second of CDC6600 central processor time for every 75 lines of executable source code in the object program.

Memory space is more of a problem. Most of the memory required by TRACE is for the arrays in which the state graph is stored. Various sizes for these arrays resulted in versions of TRACE requiring from 42000_8 to 54000_8 words of central memory on the 6600. 50000_8 words was sufficient for most of

the programs in the UT2 operating system including 1AJ which was about 1800 executable lines long.

The size of the graphs varies greatly from stage to stage in the process, but they are usually largest during the splitting step. FOLD is usually called several times during the execution of SPLIT, so it was not possible to get accurate figures on the largest size which the graphs might reach while being thus expanded. However, it is certainly possible that, if it were not constantly folded, a state graph could grow to have more states than there were lines of code in the program it represents.

The size of a state graph, both during its manipulation and the final version, is less dependent on the length of the object program than on its structural complexity; the number and depth of nested subroutine calls, the number of critical switching variables, and the range of values they assume all contribute directly to the size of the state graph. In its only major failure, the largest version of TRACE ran out of memory on 1ED, a program of about 1100 lines, many subroutines, and three switching variables with ranges of two, four, and eight. This particular program could probably be handled by simply increasing the array sizes of TRACE still further, but there will certainly always be programs whose state graphs will exceed the capacity of any given analysis program. It should be mentioned that using auxiliary memory to store the state graph would be very expensive in time and might not be practical since both the splitting and folding algorithms traverse the whole graph, possibly several times.

The most serious problems with implementing this analysis method are the various additions and modifications which must be made by hand to the source code by someone well acquainted with the method and its implementation before the automatic processing can begin. In analyzing the operating system

programs, an average of about one addition or change for every 50 lines of executable source code was necessary. Some of these, such as re-writing table jumps, are strictly shortcomings of the particular implementation program and are not necessary features of the general method. Other problems, such as recognizing actions which are not performed by a single line of code, fall into a grey area; it is probably possible to write recognition routines to detect any desired actions, but it may be more convenient to mark some of them in the source code by hand. Finally, at least some of the critical variables in object programs will have to be identified by the user. Since the number of critical variables is a crucial factor in the size of the state graphs, it is impractical to automatically designate as critical all branch variables, and no techniques are known at present for easily determining which variables actually affect the order in which a given set of actions are performed. Thus for the present the analysis method presented here will have to be thought of as an interactive one which can be largely, but not completely, automated.

CHAPTER IV

ANALYZING SYSTEMS OF PROCESSES

4.1 Introduction

In the previous chapters, state graph techniques have been applied only to individual processes. In this chapter procedures based on state graphs will be developed to prove statements about and analyze systems of processes running in parallel and interacting with each other. Although the primary interest of this study is the proof of statements about systems of computer programs, including operating system components, running in a multiprogrammed environment, the techniques are general and should apply equally well to other systems such as communication networks or factory assembly lines.

The use of state graphs to analyze systems of processes is not new. In particular, Gilbert and Chandler [8] and Bredt [3] both employ state graphs to examine the mutual exclusion problem, although using a method less general than the one presented here. Gilbert and Chandler present a method of using state graphs to detect potential deadlock, and both papers use them to check for permanent blockage. Neither paper addresses the problems of representing or dealing with system state graphs in limited space or time. In this chapter, the concept of the state graph of a system of processes will be developed, and then methods will be presented for expressing and verifying properties of such graphs, along with a discussion of the practicality of the methods. It should be admitted at the outset that the analysis methods to be presented in this chapter will be useful mainly on very small systems or on abstract systems.

4.2 System State Graphs

Suppose that there are N processes, possibly different, executing in parallel at possibly different rates. It is assumed that each of the processes can be analyzed using the techniques of the previous chapters so that each can be represented by a folded final state graph which is finite and presumably small. A system of such processes could then be represented simply by the N individual graphs as in Figure 4.1.

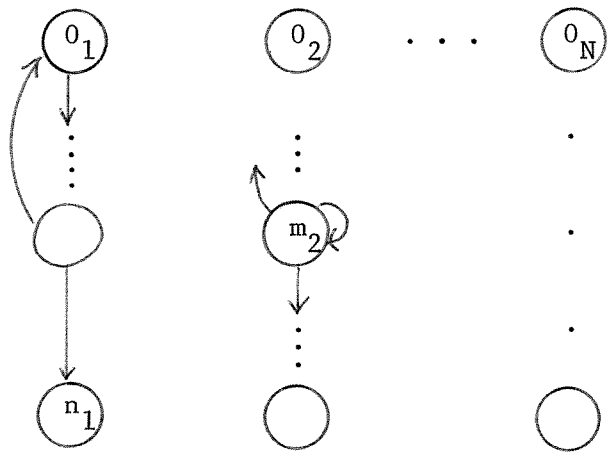

Figure 4.1. A System of Parallel Processes.

The node 0_i represents the starting state of the i^{th} process, m_i the m^{th} node of the i^{th} process, etc. As before, it is assumed that each process has a unique starting state and at least one final state. It is also assumed that each process is self-contained, i.e., that there are no arcs of the form $n_i \rightarrow m_j$, $i \neq j$.

It is not necessary to the methods of this study that the processes ever interact in any way, but if they do not, then analysis of the system reduces to analysis of the individual processes. The more interesting cases arise when the processes communicate through common store or interact by sharing or competing for other resources. In this case a notation is needed

to represent the state of the whole system at any given instant. The state of the resources of the system could be represented by a vector, V, containing the values in common (global) memory cells, and for other shared resources such as processors perhaps a number or tuple representing status or ownership. The construction of V will depend on the nature or application of the analysis. In fact, the construction of V is quite similar to the problem of designating critical variables discussed in the previous two chapters; we want to include in V only those common variables which cannot be ignored without invalidating the analysis, and like the previous problem, the question of which variables to put in V may only be solvable by successive attempts to analyze the graph. It is necessary to assume that each member of V takes on only a finite range of values. This is a restriction on the method.

One could in theory construct the state graph, G, of the whole system. This graph would have nodes labeled (i, j, ..., k, V_ℓ) corresponding to the state of the system when process 1 was in state i, represented by the i^{th} node in its graph, process 2 in state j, ..., process N in state k, and the resources in configuration V_ℓ. If one process, in performing an action, carried the system to a new state, then G would have an arc from the previous node to the new, labeled with the action performed by that process. G would be a subgraph of the graph C formed by the Cartesian product of all the individual graphs and the vector V. While G would be much smaller than C because of shared resource interlocks and branches on values of common variables, G would still be formidable for most real systems. In general, it is impractical to represent the whole state graph. Algorithms will be developed which require at most the folded state graphs of the individual processes as in Figure 4.1 and an enumeration of the nodes of G. However, it is this whole state graph, G,

which is being used as the model of the system, and except when discussing algorithms for automatic analysis, G will be referred to freely.

To the graphs of Figure 4.1, add N pointers, one for each process, pointing to the node representing the state which that process is in. When process i performs an action which carries it to a next state, the i^{th} pointer is advanced to the appropriate next node. Any program for automatically analyzing systems of processes could more easily deal with a representation of the relatively small graphs of Figure 4.1, advancing pointers along it and acting as a simulator, keeping up with the values of common variables or the status of other resources as individual processes changed them. Thus the state of the system being analyzed can be represented at all times by an ordered (N+1)-tuple, $(i,j,\ldots,k,V_\ell)$, indicating that the first process is in state i and the first pointer is at the i^{th} node in its graph, etc. It can be seen that this pointer representation results in the same labels for states of G as before. Storing all of G explicitly, on the one hand, and trying to retain the same information by adding a simulator apparatus to the individual process graphs on the other are theoretically equivalent but represent a space-time tradeoff; in the first case the connections between states are quickly available at some storage cost, and while the second method requires less memory, the neighbors of a given state are not immediately obvious and must be computed.

We define a <u>legal transition</u> of a system of processes from one system state to another to be the advancement of one of the N pointers from its present node along one arc to a next node provided that any condition attached to that arc is TRUE. Formally, there exists a legal transition from a state

$A = (a_1,\ldots,a_{i-1}, a_i, a_{i+1},\ldots,a_N, V)$ to a state

$B = (a_1,\ldots,a_{i-1}, b_i, a_{i+1},\ldots,a_N, V')$ if and only if:

(1) the graph of the i^{th} process has an arc from its node a to its node b, and

(2) the condition, if any, on that arc is TRUE when the system is in state A, and

(3) the action on that arc transforms the resource vector V to V' (where, of course, it may be that V' ≡ V).

State B will be called a <u>successor</u> of state A; A is a <u>predecessor</u> of B. There is a <u>path</u> from a state s_1 to a state s_n if there is a sequence of states $s_1,\ldots,s_{n-1}$, s_n such that there is a legal transition from s_{i-1} to s_i for $1 < i \leq n$. A state is <u>reachable</u> if there is a path from the system starting state to it. $s_0 = (0,\ldots,0,V_0)$, the state where all processes are in their starting states, will be called the <u>system starting state,</u> which without loss of generality can be assumed to be unique. A <u>system final state</u> is any state where all processes are in a final state.

At any given instant when a system is in state s, there will typically be several "next states" for the system; any one of several processes could be the next to act, and each action would in general carry the system to a different state. Typically, we have no way of knowing which process will perform its next action first, and systems appear to be nondeterministic because of this uncertainty of order of operations. The graph model being described accurately reflects this uncertainty property; there will typically be several successors to any state s in G. This is because by the definition of legal transition there will be as many legal transitions from s as there are processes which can possibly act. In cases where it matters which process goes first, this is a conservative strategy for modelling the system because no assumptions are made about the order; instead, all possible traces are created including the one where the "wrong" process acts first. However, it should

be noted that there is an implicit assumption in this model: In agreement with past treatments of this problem (see, for example Dijkstra [6], p. 53, and Gilbert and Chandler [8], p. 429) it is assumed that if two or more processes are each about to perform an action, one or another of them actually acts first, even if we cannot know which, but that there is never actual simultaneity. In the case of single processor hardware, this is equivalent to assuming that race conditions are always resolved. However, in multiprocessor systems there might be actual simultaneity, and there is no generally satisfactory way of reflecting this property with the present model. However, if the two actions do not affect the same resource, it probably does not matter which we say occurred first, while if they do, then there is a race condition which is presumably resolved in some order in any real multiprocessor system.

It has been mentioned that the graphs of the individual processes in the system are to be built by the methods of the previous chapters. If these graphs are to be used in the analysis of a system of processes, then obviously the interactions of the processes are of interest. For this reason it is assumed that actions by the processes which may affect their interaction were designated as "interesting" and appear on the arcs of the graphs. Specifically, it is assumed that any processes which test or set the value of any of the variables in the common vector V have such actions explicitly labeled on the arcs of the graphs being used to analyze the system. Some of these arcs may have special kinds of labels indicating that they may not be eliminated by folding but are later to be considered to be null (ϵ) for purposes of comparison with a prototype graph. While the graph manipulation techniques described in the previous chapters are certainly also applicable to the system graph G, in practice there will probably not be much opportunity to use them.

Because the individual process graphs from which G is constructed have already been thoroughly folded, there will be little or no folding possible on G; and because account was taken of actions concerning the system's critical (common) variables as the states of G were generated, splitting will not be necessary.

A common feature of multi-process systems is that processes may spawn child- or successor-processes; that is, one process may generate or initiate a second process which runs in parallel with the first. This child-process may then have the same status as the other processes in the system, using the common store, competing for resources, and even spawning children of its own. In terms of the "fork" and "join" notation of Conway [5], this initiation corresponds to a "fork." Regardless of how such process generation actually occurs in the system being analyzed, this feature can be modeled using the notation already developed. The graph of the potential child process is included in the system of graphs from the beginning, and its node pointer is initially at node 0. At this first node the only outgoing arc has as a condition a certain value of an enabling variable, say e, in common store. At the appropriate point in the parent process, e is set to the value which gives the child a legal transition. "Joins" can be modeled in a similar fashion.

Processes may have more than one child, and lineages may be of indefinite length, but it is necessary that at least one copy of the graphs of <u>all</u> potential processes be included in the original graph system. Further, it is necessary that the <u>total number</u> of such processes, either initially active or potential, be finite and known in advance so that the proper number of pointers can be provided. This is definitely a limitation on the method, but fortunately an acceptable one for most applications. In particular, the number of processors, even virtual ones, and hence the number of active processes, is bounded on

most computer systems; job- or task-tables and queues are of finite length. However, systems of recursive processes cannot, in general, be modeled using this method.

4.3 Analyzing Sequencing Properties of Systems

We are now ready to develop methods for proving properties of or statements about such systems of processes. The most general, although not the only, class of properties which systems can be proved to have are those which can be expressed in terms of a prototype graph. The verification method will be similar to the method used earlier to verify that individual processes had properties expressed by prototypes.

As before, the user will have to provide a prototype graph, P, which expresses the desired ordering relationship on the actions performed by the system of processes. Each node in the prototype graph will represent a state of the whole system. If the ordering rule is of the type "some (any) process must do A before some (any) process does B," then the prototypes will look just like those for individual processes. However, if it matters at any point that an action be performed by a specific process, then the action-labels on the arcs of the prototype will have to be subscripted with the identity of that process. As an example, let us consider the mutual exclusion problem. Suppose that two processes each have a critical section and that we wish to verify that they cannot both be executing this section simultaneously. By denoting two actions as "interesting," namely, "begin critical section," BC, and "end critical section," EC, we can state the desired order-of-operation rule as: "once either process has done a BC, that same process must do an EC before the other can do a BC." The prototype graph expressing this

property is given in Figure 4.2.

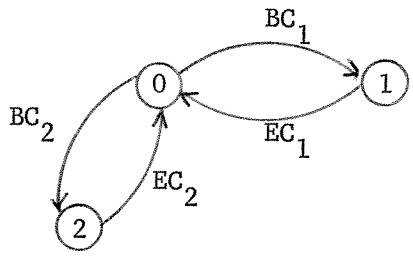

Figure 4.2. Prototype Graph for Mutual Exclusion of Two Processes with Critical Section

Of course, if the processes had more than one critical section governing the use of different resources, then different arc labels would be used to denote entry and exit for each critical section, and the prototype would be more complex. If P is a primitive in a given system, then in some applications "P" and "BC" might be equivalent, but in general the action of reserving a resource and beginning the program segment where it is used are not equivalent, so two different arc labels may be needed.

The system comparison procedure will be slightly more involved than the comparison algorithm of Chapter II because in generating legal successors the procedure must also be somewhat like a simulator, keeping up with values of V and evaluating conditions on the arcs of the individual graphs, and because the arcs in the prototype may specify actions by a particular process. The system comparison algorithm presented below is written assuming that arc labels in the prototype are subscripted as in the example in Figure 4.2; if this is not the case, the algorithm can be made slightly simpler.

System Comparison Algorithm

1. Set up a list, L, of pairs (s,p) where s is a node of G and p is a node of P. Initialize L to $\{(s_0,p_0)\}$, the starting nodes of G and P, respectively.

2. If there exist s, p, x, i, s', and p' such that (s,p) is in L, $s \overset{x}{\Rightarrow} s'$ is in G and the arc traversed to carry G from s to s' was in the i^{th} process graph, and $p \overset{x_i}{\rightarrow} p'$ is in P, but (s',p') is not in L, add (s',p') to L and repeat this step.

3. If there is a member (s,p) of L such that s is final but p is not, report an error, otherwise report success.

The above algorithm will now be applied to a system consisting of two processes embodying a purported solution to the mutual exclusion problem. This "solution" is flawed and is not seriously proposed but is constructed solely for illustrative purposes. Suppose that a system designer has decided to insure that two processes are never in their critical sections simultaneously by instituting an interlock consisting of setting and testing a single common variable w, initialized to 0. When a process wants to enter its critical section it must first increment w by 1 and then test it against 1. If $w \leq 1$, the process may proceed through C; if $w > 1$ then the other process is in its critical section and the first process must wait. As a process finishes its critical section it decrements w. Figure 4.3 shows the graph for such a system.

Notice that since both processes have 6 states and since w can take on 3 values, 0, 1, and 2, there are theoretically $6 \times 6 \times 3 = 108$ possible states of this system. However, as we shall see, only 32 states are actually reachable

and need be considered. Such dramatic differences in size between C, the Cartesian product graph, and G the actual state graph are to be expected in real systems and help make state graph analysis practical.

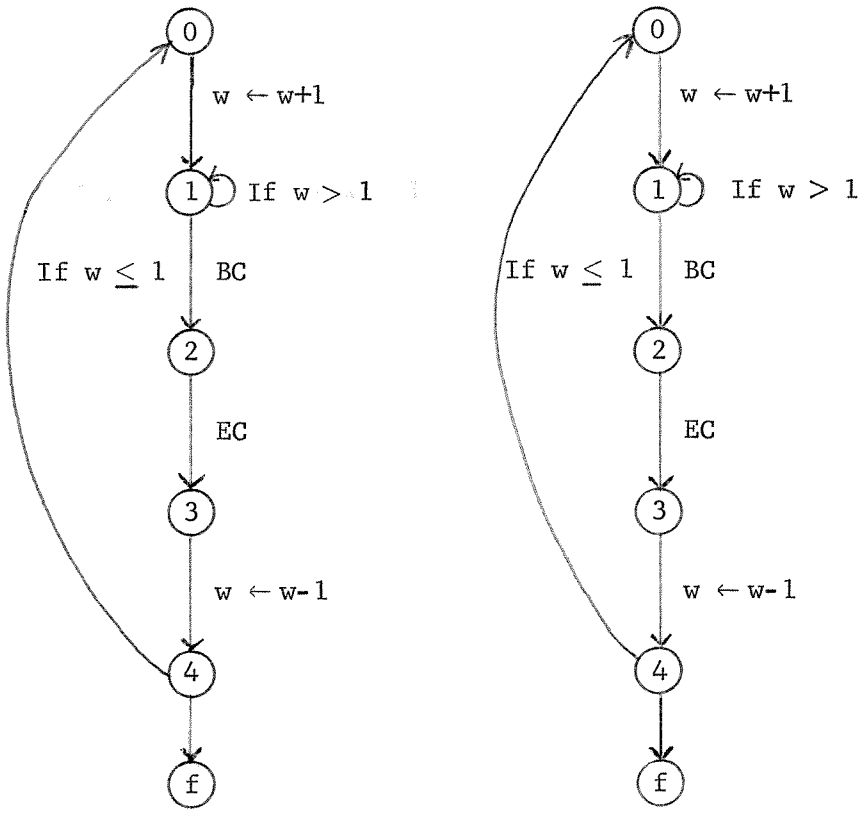

Figure 4.3. Two Parallel Processes with Interlock

Let us use the system comparison algorithm with this system and the prototype of Figure 4.2 in order to prove that the processes cannot both be in their critical sections at the same time. The initial state of G is (0,0,0) which we shall call 0 for short, and the initial state of P is 0, so the first pair in L is ((0,0,0),0). There are two successors to 0 in G, 1 = (1,0,1) and 2 = (0,1,1). In both cases, the arc from node 0 to node 1 in the individual process graph is ϵ for the purposes of this algorithm since

only BC and EC appear in the prototype. Thus ((1,0,1),0) and ((0,1,1),0) are the next two pairs in L. The execution of the algorithm is straightforward and results in the list L which is presented in Table 4.1. The final state of G, (F,F,0) is paired only with 0, the final state of P, so the comparison is successful. Thus the given interlock scheme involving w will indeed prevent the two processes from being in their critical sections simultaneously..

Along with each pair, (s,p), Table 4.1 also lists all of the legal successors of s. This column is presented here for expository purposes and will also be referred to later.

The proof that the algorithm does correctly determine whether or not all the traces in the system graph G are also traces in P is essentially the same as that given for the algorithm in Chapter II and will not be presented in detail. However, one point requires further consideration.

It can be seen that the sequencing analysis method for systems presented in this chapter consists basically of viewing the system as a single process and then using the method of Chapter II. This is no small step, since two processes each with a certain property can together form a system which does not have this property, and vice versa. A case in point is termination. Two processes can have the property that there is a path from every state to a final state when running alone and yet deadlock when running together in parallel. Conversely, two cooperating processes might be designed which manipulated resources common to the two of them in such a way that they progressed nicely together but neither alone could ever reach a final state without enabling actions by the other. One implication of this observation is that there are no reasonable assumptions which can be made about individual process graphs which will guarantee that the system graph will have the property

TABLE 4.1

EXAMPLE OF USE OF THE SYSTEM COMPARISON ALGORITHM

Pairs in L			Successors of s
s		p	
0	(0,0,0)	0	1, 2
1	(1,0,0)	0	3, 4
2	(0,1,1)	0	3, 5
3	(1,1,2)	0	None
4	(2,0,1)	1	6, 7
5	0 2 1	2	8, 9
6	2 1 2	1	10
7	3 0 1	0	10, 11
8	1 2 2	2	12
9	0 3 1	0	12, 13
10	3 1 2	0	14
11	4 0 0	0	0, 14, 15
12	1 3 2	0	16
13	0 4 0	0	0, 16, 17
14	4 1 1	0	2, 18, 19
15	f 0 0	0	19
16	1 4 1	0	1, 20, 21
17	0,4 0	0	20
18	4 2 1	2	5, 22, 23
19	f 1 1	0	22
20	1 f 1	0	24
21	2 4 1	1	4, 24, 25
22	f 2 1	2	26
23	4 3 1	0	9, 26, 27
24	2 f 1	1	28
25	3 4 1	0	7, 27, 28
26	f 3 1	0	29
27	4 4 0	0	11, 13, 29, 30
28	3 f 1	0	30
29	f 4 0	0	15, 31
30	4 f 0	0	17, 31
31	(f,f,0)	0	None

that there is a path from every state to a final state. Recall that this property was discussed in Chapter II. The remarks made there apply here also; a user of the system comparison algorithm can simply assume that the system state graph G has the property, or he can attempt to prove that it does, or he can apply the algorithm to graphs which do not, with the understanding that the algorithm is then comparing sequences of arc labels in G and P rather than elements of $\mathcal{L}(G)$ and $\mathcal{L}(P)$. Certainly the best alternative, if it is possible, is to prove this feature of G or to identify the system states which cannot reach a final state. This problem will now be considered in detail.

4.4 Deadlock

The whole subject of systems reaching their final states, states in which each process is in a final state, is very important in computer operating system theory. As it turns out, a system's state graph as defined here so far is also a natural model for investigating these properties of systems. One feature for which we would like to analyze systems of processes operating in parallel is the possibility of deadlock.

Deadlock has been defined in various ways in the literature, usually in terms of system resources. For example, Haberman [10] says it exists when "cooperating processes prevent their mutual progress even though no single one requests more resources than are available." Holt's [13] definition is: "the resources of the system have been allocated among certain processes in such a way that it is impossible to grant additional requests to these processes."

The concept of system resource is not strictly necessary to the concept of deadlock. One thing that is necessary is that processes in the system not be completely independent, i.e., that they interact, and that the condition

exists wherein one process may have to halt, or block, while waiting for another. That for which it must wait, even if only a message, may be considered to be a resource and may be considered to be "owned" by the system or a process if necessary or desirable for the unification of a theory or to aid analysis. For a thorough discussion of the ownership problem see Howard [15]. Howard and Holt [13] have both observed that one can always define blockages (and thus deadlock) in terms of resources by, if necessary, inventing resources.

Here deadlock will be defined solely in terms of sequences of legal transitions of the states of a system of processes. The idea is to define a set of circumstances in which two or more processes are blocked waiting for each other and in which there are no state transitions available to the other processes, if any, which will unblock them.

First we define "waiting" in terms of state graphs: Process i is waiting for process j, $i \neq j$, $P_i \lessdot P_j$, if

(1) P_i cannot make any non-ϵ transitions because the conditions leading to the non-ϵ arcs are FALSE (to allow for the case where P_i is in a testing loop, the definition will allow P_i some ϵ transitions provided that the only traces P_i can generate are ϵ^*) and

(2) there is on a path between P_j's present state and a reachable state some action which will make the condition on P_i's non-ϵ arc TRUE. P_j need not be able to actually take the path if some condition along it is also FALSE; the definition only requires that the path exist.

The subtle ownership problems which must be dealt with in other definitions of blockage appear here, too, even though no explicit mention is made of resources or ownership. Suppose that there are several processes any one

of which can perform an action which unblocks P_i. Then for whom is P_i waiting? In this case we will simply say that $P_i \lessdot P_j$ is true of each P_j which can change the value of the condition on P_i's blocked arc. But the situation can be even more complicated. P_i might be waiting for all of a set of processes to act, either because they are all necessary to change one condition or because P_i has a compound condition on its arc. In fact, since the conditions on arcs can be arbitrary logical expressions, wait relations among processors can be arbitrarily complex. In the definitions which follow, the simplifying assumption will be made that no more than one process will ever have to act to change the condition on an arc of any other process.

We shall now define a condition which corresponds to "circular wait" in other definitions of deadlock but which is more general.

A non-empty subset D of processes is a knot if for all P_i which are members of D,

(1) if $P_i \lessdot P_j$ then P_j is also in D, and

(2) there exists a P_k in D, different from P_i, such that $P_i \lessdot P_k$.

Deadlock can now be defined in terms of system state graphs: A system is in a deadlock state (or deadlocked) if some subset of its processes form a knot. A system is said to have a potential deadlock if one of its reachable states is a deadlock state.

It is easy to see that this state graph definition of deadlock includes all the cases described by the well-known four conditions of Coffman, Elphik, and Shoshani [4]:

(1) Tasks [processes] claim exclusive control of the resources they require ("mutual exclusion" condition).

(2) Tasks hold resources already allocated to them while waiting for additional resources ("wait for" condition).