implement mutual exclusion on a table J using semaphores.

```
          .
          .
      JAP     LOC1        noncritical branch to LOC1
      (P)     J           reserve table J
      ENTA    1
      STA     SW          SW ←1
      JMP     LOC2        unconditional jump to LOC2
          .
          .               (uninteresting code)
          .
LOC1  STZ     SW          SW ←0
          .
LOC2      .               (uninteresting code)
          .
      LDA     SW
      JAZ     LOC3        branch to LOC3 if SW = 0
      (V)     J           release table J
LOC3      .
          .
```

Figure 2.4. Segment of Source Code

The building techniques described so far result in the graph segment of Figure 2.5.

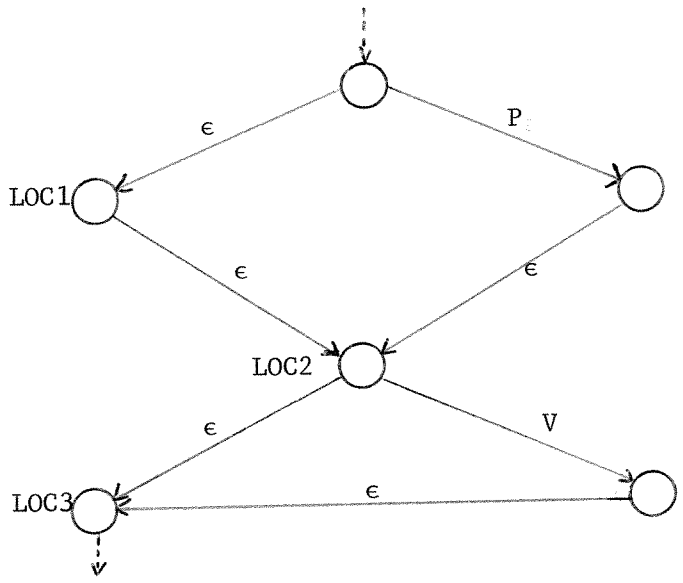

Figure 2.5. Segment of Initial State Graph (Incorrect)

2.6 Splitting

In the example above, it is immediately obvious that something is wrong. When this graph is compared to the prototype graph of Figure 2.1, it appears that there are several illegal traces. Apparently the program can reserve the table and not release it or release it without having reserved it since the traces Pεε and εεVε are both present in the state graph. But a careful examination of the source code shows that these crimes will never actually be committed and that the program is in fact correct in its use of P and V.

This example illustrates the necessity of expanding the initial state graph to make needed distinctions based on the values of program variables. In the example the problem is, of course, that no account was taken of the switch variable SW in building the graph. The graph does not have enough states. The initial state graph produced directly from the program's source code is not at all a complete state graph; it does not have a different node for each combination of variable values. Many combinations are lumped together in one state. In fact, the only distinction between states in this initial graph is in the value of the program counter; no distinctions based on values of other variables have been made. As the example shows, distinctions between states based on other program variables may be needed. Yet not all distinctions can be kept because the full state graph is unmanageably large.

The solution to the problem is to designate certain variables to be critical. The initial state graph will be expanded to reflect distinctions between states based on different values of these critical variables, while other variables will continue to be ignored. This designation can be selective; variables can be ignored in some parts of a program and considered

critical in others. The definition of critical variables is pragmatic and may be after-the-fact: a variable is critical if failure to consider it so results in spurious illegal traces in the state graph.

Two modifications to the building process are necessary in order to deal with critical variables. Whenever a variable which has been designated to be critical acquires a new value, this fact must be noted and information about its value attached to the current node of the graph. Whenever a branch is made on the value of a critical variable, the conditions under which each path is taken must be attached to the outgoing arcs. If SW had been designated critical in the example of Figure 2.4, then the initial state graph should be that of Figure 2.6.

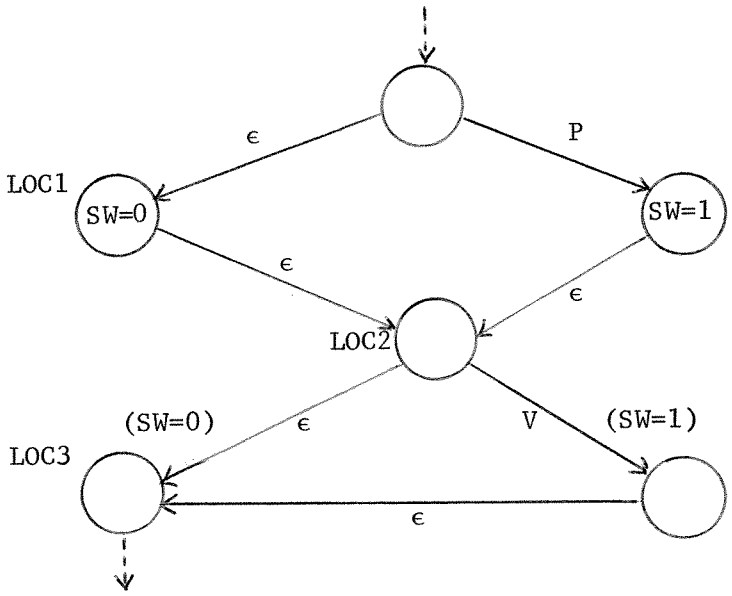

Figure 2.6. Segment of Initial State Graph (Corrected)

The graph now contains all the necessary information, and it remains only to expand the graph to include the additional states. This expansion is accomplished by splitting. In splitting, one node is split into two or more nodes, with all arcs and all information attached to the node retained.

This process begins at the node from which there are conditional arcs and propagates backward along the directed graph to nodes at which the critical variable acquired known values. At these points arcs with conditions contrary to the information on the node are deleted. Thus two or more copies of the whole graph segment between the variable being set and its subsequent use are created corresponding to the (presumably) different sequences of actions performed as the critical variable takes on its range of values. An implementation of this splitting process is described in Chapter III. The use of this technique implies that any variable to be designated critical must be known to take on only a finite range of values. This is a limitation on the method, but fortunately, critical variables usually obey the restriction.

To continue with the example: applying the splitting process to the graph of Figure 2.6, beginning at the node labeled LOC2, the result is the graph of Figure 2.8. The graph of Figure 2.7 shows an intermediate stage of the process.

There are now two nodes representing the state of the machine when the program is at the statement labeled "LOC2" corresponding to the two different paths by which the program might get there, i.e., corresponding to the two different values of the variable SW. The state graph no longer looks so much like a flowchart, but the retention of labels and variable values with each node allows the user to see what each node corresponds to in the behavior of his program.

The identification of critical variables is the only major part of the methods described in this chapter which has not been automated. The technique currently in use is for the human user to look over the source code and to identify (by insertions into the text of the code) the intervals, if any, in which certain variables must be considered critical. In practice the identity

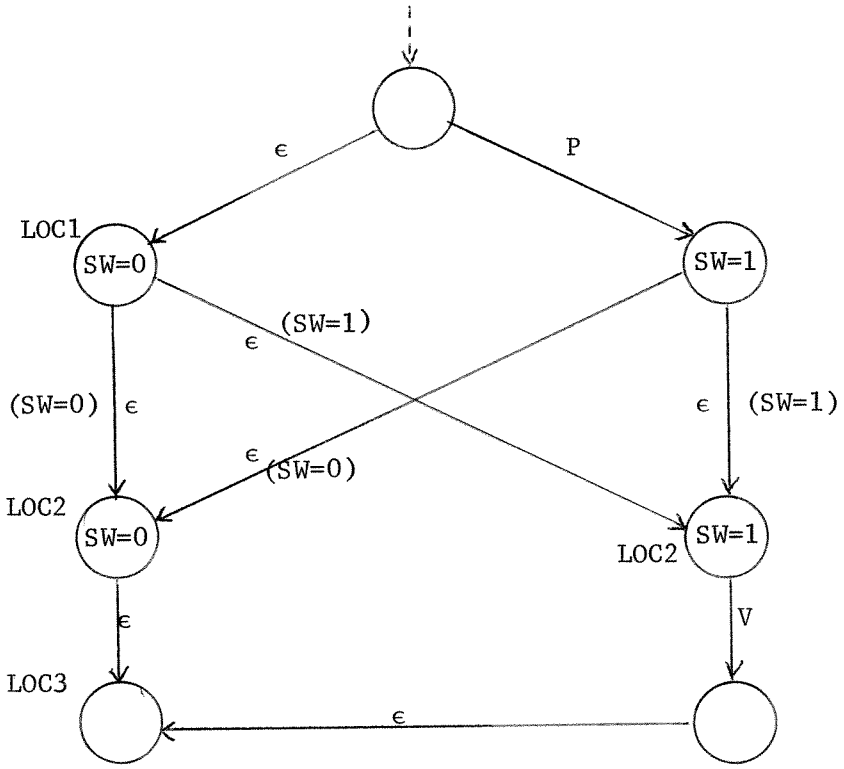

Figure 2.7. Graph Segment During Splitting

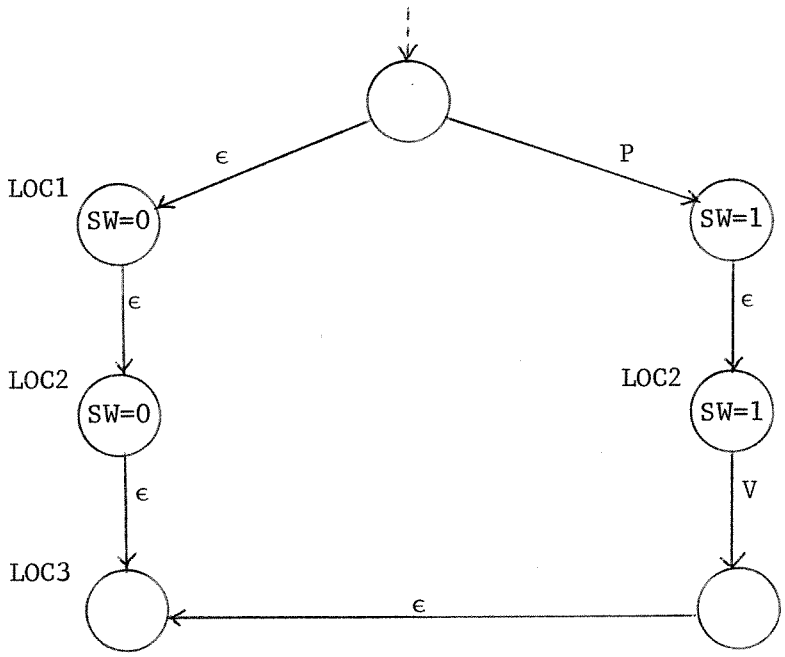

Figure 2.8. Graph Segment After Splitting

of critical variables is usually obvious; furthermore, the failure to identify one is not disasterous. As the example illustrates, failure to note a critical variable can result only in the presence of spurious illegal traces in the state graph, and the attempt to find the corresponding illegal path in the program will bring the omitted variable to the user's attention.

One class of critical variables can be identified automatically, namely the locations to which the program returns from subroutines. Associated with each subroutine encountered in the source code is a generated unique variable which is automatically designated critical. When building the initial state graph, each call to the subroutine results in setting the value of this variable, and the return statements in the subroutine are treated as branches on its value. The splitting process then results in the insertion of a copy of the subroutine's graph at each point in the program's graph where the subroutine was called. The finiteness restriction on critical variables implies that recursive subroutine calls cannot be handled.

One other problem in connection with critical variables arises when they do not acquire their values by explicit assignment statements but rather are set by input statements or are initialized externally. In this case the splitting process as described so far cannot terminate since no nodes will be found at which a critical variable has a known value. The most useful solution is to modify the splitting procedure slightly so that it stops at the starting node and at relevant input nodes, simply leaving the two (or more) outgoing arcs with conditions attached. This will result in the creation of two or more copies of the graph segment between this node and the point where the branch occurred. Following one or more of these paths should yield an illegal trace (else the variable was not really critical) and thus the explicit conditions on the arcs will tell the user under exactly which initial

values of the critical variable his program will run correctly.

2.7 Folding

It has been mentioned that the full state graphs of most real programs are unmanageably large. The number of nodes in the full state graph of a program may be thought of as the product of the sizes of the ranges of all memory cells and registers which the program uses or may use. Thus the state graphs of many programs will have more nodes than the number of memory cells actually available on the largest computer, even if we ignore the problems created by the facts that the ranges of some variables may not be known and that since programs may be run on virtual machines even the number of memory cells used cannot be determined *a priori*.

In general, the problem of the size of state graphs is solved by discarding unnecessary information and ignoring the distinction between two states when the distinction is irrelevant or of no use to the analysis. The technique for compressing state graphs is called *folding*. Folding consists quite simply of combining two or more nodes into one node, preserving all the arcs into or out of all the original nodes by having them all go into or out of the one new node.

The initial state graphs of programs can be large, and the splitting process can greatly increase their size. The comparison algorithm must deal with each node of the state graph at least once and possibly many times. Thus in order to save both memory and processing time, it is necessary to keep the graph as folded as possible at all times. In fact, folding is the crucial technique which makes practical the analysis methods presented in this study.

Folding can be thought of as a homomorphic mapping from the set of

nodes of a graph onto a subset of the nodes which preserves the connectivity properties of the graph. If G is the original graph and H is the folded graph, then a folding is a mapping, f, from the nodes of G to the nodes of H such that if $s \xrightarrow{x} s'$ then $f(s) \xrightarrow{x} f(s')$. For a thorough discussion of graph homomorphisms see Hedetniemi [11].

In practice, folding can almost always be defined in terms of ignoring a particular variable or set of variables; if the only distinction between the states represented by two nodes is the value of x, then folding those two nodes together is equivalent to ignoring the value of x. Notice that the source code listing of an assembly language program is an extremely folded version of its own state graph in which every variable except the program counter is ignored. Each line of executable code labels the transition from one state to another, i.e., from one value of the program counter to another. The initial state graph built from the source code is similarly folded. The problem in the example in which the value of SW was ignored was that the initial graph was too folded.

It is of crucial importance that since folding preserves individual arcs with their labels, it preserves both paths and traces. That is, if the trace xy...z existed in a graph G, and G is folded to H, then the trace xy...z will also be present in H. A proof that folding preserves traces can be found in Howard [15].

This property of folding allows us to deal with folded versions of the state graph of a program when searching for sequences of operations in violation of some rule; if an illegal trace does not exist in the folded graph, it did not exist in the original graph. Unfortunately, the converse of this is not true; that is, folding will generally add traces which were not in the original graph. This is undesirable only if the added traces are illegal,

that is, if they were not allowed by the rule to which we are trying to prove that the program adheres. Thus the general strategy in dealing with state graphs must be to fold them as much as possible without introducing spurious illegal traces.

The only restriction of folding is that it must not introduce into the graph illegal traces, i.e., traces not present in the prototype. But this criterion is difficult to apply directly since constant reference to the prototype would be inconvenient and time consuming and since the comparison algorithm cannot even be applied until all the splitting has been done. The criterion can be applied indirectly, however, by using the one feature known to be present in all prototype graphs, namely the unit ϵ-arcs at every node. The presence of these arcs in the prototypes guarantees that two traces differing only in instances of ϵ will either both be legal or both be illegal. This observation permits the adoption of the following two folding rules:

1. If node s is directly connected to s' only by a ϵ-arc from s to s', and either (a) s has no other successors or (b) s' has no other predecessors, then s and s' may be combined.
2. If there is a ϵ-arc from s to s' and a ϵ-arc from s' to s, then s and s' may be combined.

It is easy to see that any traces added by folding in accordance with these rules will differ from some trace already present in the graph by at most some instances of ϵ. For example, consider the "only successor" rule illustrated by Figure 2.9. Represent any trace from s_0 to s by α, any trace from s_0 to s' by β, and any trace from s' to a final state by γ. Then the only traces from s_0 to a final state going through either s or s' are $\alpha\epsilon\gamma$ and $\beta\gamma$. After folding, the traces added are $\alpha\gamma$, $\beta\epsilon^{+}\gamma$, etc.

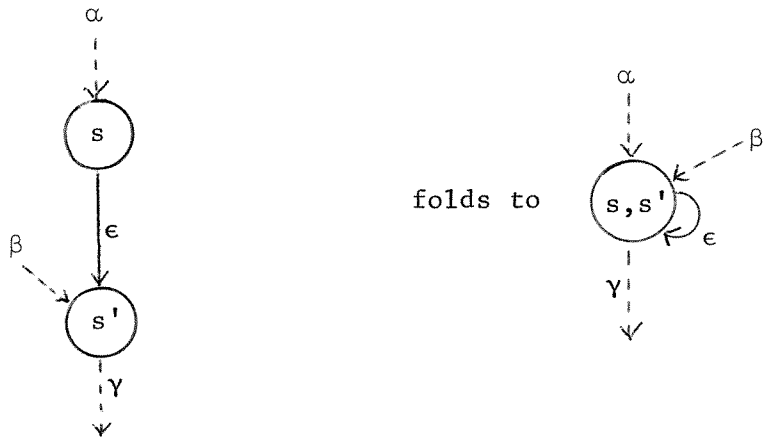

Figure 2.9. Illustration of the "Only Successor" Folding Rule

Since adding or deleting ϵ from a trace does not make it a new trace for the purposes of sequencing analysis, it can be seen that the strategy represented by the two folding rules given is quite conservative; not only are no illegal traces added, but no new traces of any kind are added. With or without allowing reference to the prototype, more sophisticated folding rules can be devised which result in an even more compact graph than that resulting from application of the two given rules; there is some discussion of folding strategy in Chapter III.

Returning once more to the table reservation example, applying the folding rules to the graph of Figure 2.8 yields the graph of Figure 2.10.

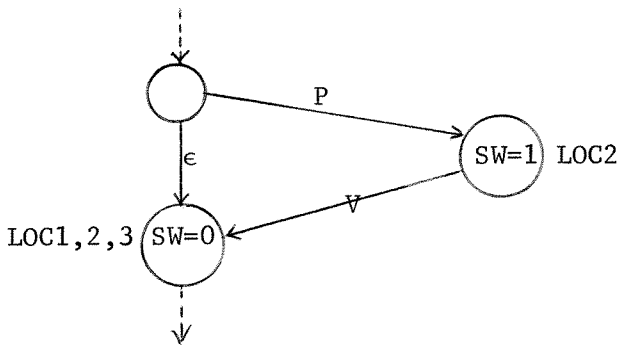

Figure 2.10. Graph Segment After Folding

The comparison algorithm can now easily verify that the graph contains only traces present in the prototype of Figure 2.1, and therefore the code segment it represents uses the P and V operations correctly.

In this particular example the two nodes connected by the only remaining ϵ-arc could have been combined safely, but only because the prototype happens to allow repeated P-V pairs. Had the prototype been instead as in Figure 2.11, this folding would have introduced spurious illegal traces. This example is sufficient to show that one cannot fold across all ϵ-arcs.

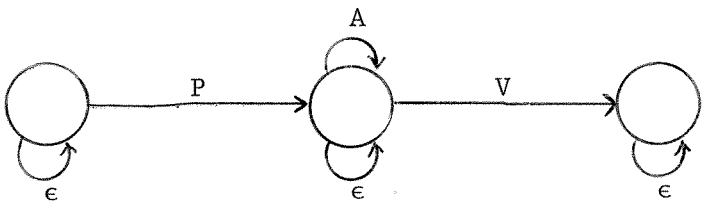

Figure 2.11. Prototype for One-use-only

This example also illustrates one reason why in practice the final graphs even of correct programs can almost never be folded to exactly match the prototype; even if there is such a folding, universally safe folding rules will usually not accomplish it. Another reason is that there may be no such folding. In Figure 2.3, G is correct with respect to P, but there is no folding of G which will result in P.

2.8 Summary

The general method for proving assertions about the order in which programs perform interesting actions will now be summarized before proceeding to more detail in Chapter III. The method assumes that three things are given:

a source listing of the program, a list of interesting actions which the program may perform, and a prototype graph. The method consists of three main steps:

First, an initial state graph is built from the source code. This is the hardest step to perform in practice, the most difficult to describe since much depends on the source language of the object program, and the step least amenable to proof of correctness. The method can probably be made provably correct but at approximately the cost of proving correct a large compiler. In practice, the user will probably have to simply assume that an initial state graph has been built which accurately reflects the sequencing properties of his program. This assumption includes two subsidiary ones: that the code which performs interesting actions can always be recognized as such, and that all critical variables have been so designated and take on only a finite range of values.

This initial state graph must then be expanded to reflect differences between states when critical variables have different values. It must also be folded to save time and space. These two graph manipulation actions can be taken with perfect confidence that the sequencing properties of the initial state graph have not been altered.

The third and final step is to verify that the state graph has the specified property. If the property has been specified by use of a prototype state graph, then the comparison algorithm can be used. It is assumed that if other formalizations are used they would be chosen with a view to algorithmic verification.

It can be seen that the model can be manipulated with more confidence than it can be built. But the assumptions involved in building the initial

state graph are not as dangerous as they might seem because the method tends to be failsafe; the most likely error in building a graph, the failure to note a critical variable, will add traces rather than delete them. If this method verifies that a program has a specified property, a user may be fairly confident in that result. If the method indicates that a program is incorrect there are three possibilities: the user can find the illegal path in the program, find an overlooked critical variable and try the method again, or remain uncertain about the correctness of his program.

CHAPTER III

PRACTICAL EXPERIENCE

3.1 Introduction

In this chapter, the techniques which are a part of the analysis method presented in the last chapter will be discussed in more detail. The emphasis will be on practical problems, and the discussion will focus on a program, TRACE, which has implemented some of the techniques to perform sequencing analysis on real object programs. TRACE is also discussed in a paper by Howard and Alexander [16].

TRACE is written in FORTRAN and accepts as input object programs written in CDC6600 peripheral processor assembly language, pp COMPASS. Using the techniques already described, it builds and manipulates state graphs and, optionally, outputs them at various stages in the processing. It may also accept a prototype graph and use the comparison algorithm to verify that the object program adheres to the prototype. TRACE is organized in overlays. The main program consists of numerous utility routines and a simple driver. The driver calls in turn three overlays which perform the three steps of the analysis method: BUILD reads the source code and the list of interesting instructions and builds the initial state graph. SPLIT processes all conditional arcs, splitting nodes to reflect different values of critical variables. CLEANUP performs final folding, reads in the prototype, and performs the comparison algorithm.

3.2 Building the State Graph

BUILD is divided conceptually into two functions, a parser and recognizer for the source code, and a set of executive routines which build the initial state graph. BUILD first reads in the user-supplied list of source code operations which result in some addition to the graph being built. In effect, by ignoring and taking no action upon those instructions whose op-code is not on the list, BUILD is folding the graph even as it builds it. This list includes, but is much larger than, the set of interesting operations which appear in the prototype graph. This larger list must include all control operations in the object language such as jumps, return jumps, and branches. While it is unlikely that such program control actions would appear explicitly on a prototype graph as an interesting action, they do determine paths through the graph from state to state and hence must be taken into account. Keeping up with the value of critical variables requires that some loads and stores also be on the list. A particular store action will result in an addition to the graph only if the operand is a critical variable. Even though the importance of some instructions depends partly on their context, once a source statement has been parsed and recognized it is relatively easy to make the appropriate addition to the graph.

The more difficult function of BUILD is the recognition of important statements. COMPASS is a relatively powerful assembly language with such features as overlays and macros with conditional assembly. Yet BUILD had to be far shorter and simpler than a full COMPASS assembler to allow more time to develop the other, theoretically more interesting, portions of TRACE. Thus no attempt was made to handle such difficult constructions as table jumps, "ZJN *+7" (branch to this address plus 7), or macro calls. In all of these

cases, the user must substitute more straightforward code into the object program before giving it to BUILD.

An even more critical problem is the recognition of important actions when these actions are not performed in one line of code. An example will illustrate the problem. In some systems there may be a one-line primitive such as Dijkstra's P operation for reserving tables, but in the UT2 operating system used at the University of Texas Computation Center this action is accomplished by a series of statements which load a certain value into a certain address (to be polled by a monitor) and then repeatedly test that address for a response signal. The names assigned to the variables involved may vary from program to program. Some system programmers use a macro to accomplish the task, in which case the name of the macro serves very nicely as the instruction to be recognized, but some programmers do not use the macro. In the latter case it would seem that the program analysis method being presented here needs to have incorporated into it both a pattern recognition facility and some technique for specifying patterns of program code. This feature has not been implemented; at present if an important action cannot be recognized by the presence of one line of code, such a line must be inserted at the appropriate point in the source by the user.

As was mentioned in the previous chapter, the user must make one other set of additions to the source code of the object program to denote critical variables. If x is critical within a certain segment of the program, then some pseudo-op such as "NOTE X" must be inserted before that segment. "ENDNOTE X" may optionally follow the segment. These insertions need not be made for the critical variables connected with subroutine calls and returns which are handled completely automatically.

3.3 Splitting

After the initial state graph has been built and some preliminary folding accomplished, SPLIT is called. During the building, branches on the value of critical variables resulted in two or more outgoing arcs from the current node being tagged with the condition under which each path would be followed. Nodes with outgoing conditional arcs are called "question nodes." Each time a question node was created as a result of such a branch, BUILD put that node on a pushdown stack which was saved for SPLIT.

SPLIT begins each cycle through its algorithm by considering the top node on this question stack. If the node has associated with it state information relevant to the conditions on the outgoing arcs, appropriate action is taken: if the condition on the arc is incompatible with the state, the arc is deleted; if the state satisfies the condition, then the condition is removed from the arc. If there remain arcs with conditions about which the node contains no information, then the node is split into as many copies as there are mutually exclusive conditions. Each node has attached to it state information compatible with one condition, and this condition is attached to all its incoming arcs. The conditions are then removed from the outgoing arcs. The node is then removed from the question stack, and if it has been split then all its predecessors are added to the stack since they now have conditional outgoing arcs. When the question stack is empty, SPLIT terminates.

Notice that since state information is attached to each newly split node before leaving it, and since an attempt is made to answer the questions on the arcs with information at hand on the node before splitting it, this process cannot get trapped on loops; it will traverse a loop in the state graph at most once for any given set of conditions.

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 e-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.

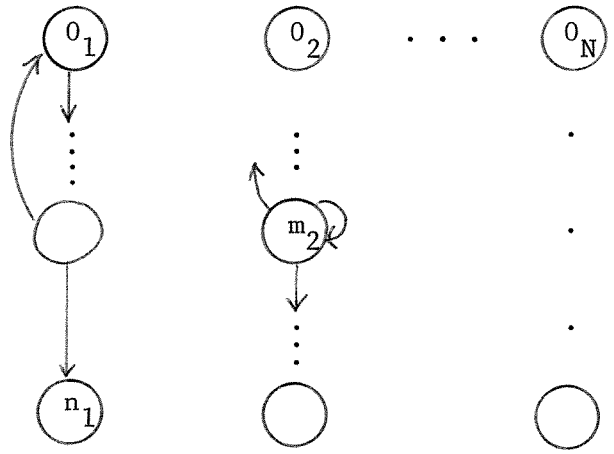

Figure 4.1. A System of Parallel Processes.

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.

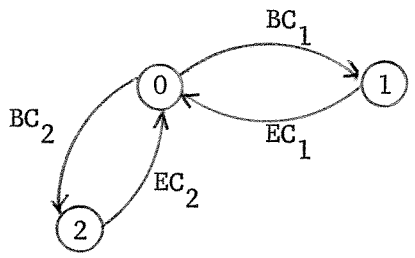

Figure 4.2. Prototype Graph for Mutual Exclusion of Two Processes with Critical Section

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, $s \xrightarrow{x} s'$ is in G and the arc traversed to carry G from s to s' was in the i^{th} process graph, and $p \xrightarrow{x_i} p'$ 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.

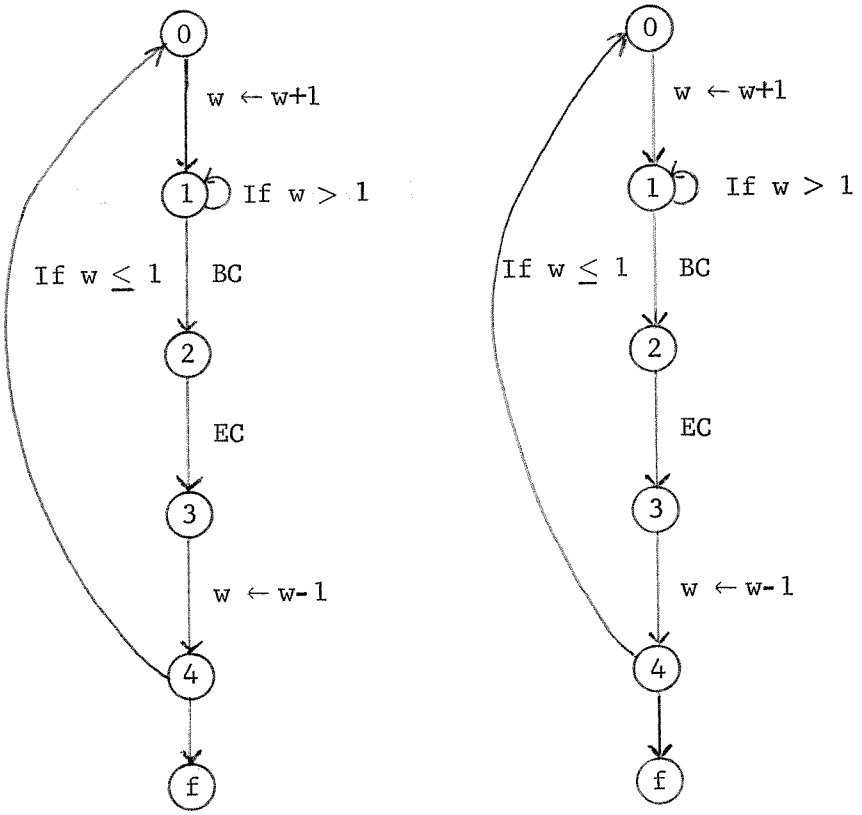

Figure 4.3. Two Parallel Processes with Interlock

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