(3) Resources cannot be forcibly removed from the tasks holding them until the resources are used to completion ("no preemption" condition).

(4) A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain ("circular wait" condition).

These four conditions are necessary for deadlock to exist and are both necessary and sufficient if all resources are unique. The circular wait condition is the special case of a knot in which each process is waiting for exactly one other process.

There is another presumably undesirable system situation which does not seem to fit any definition in the literature. It is the situation, analogous to that in Figure 2.3 for individual processes, in which every process in the system is making transitions and possibly even doing useful work, yet the system can never reach any of its resting states. Consider the system in Figure 4.4. If through programming error or other means the common variable x

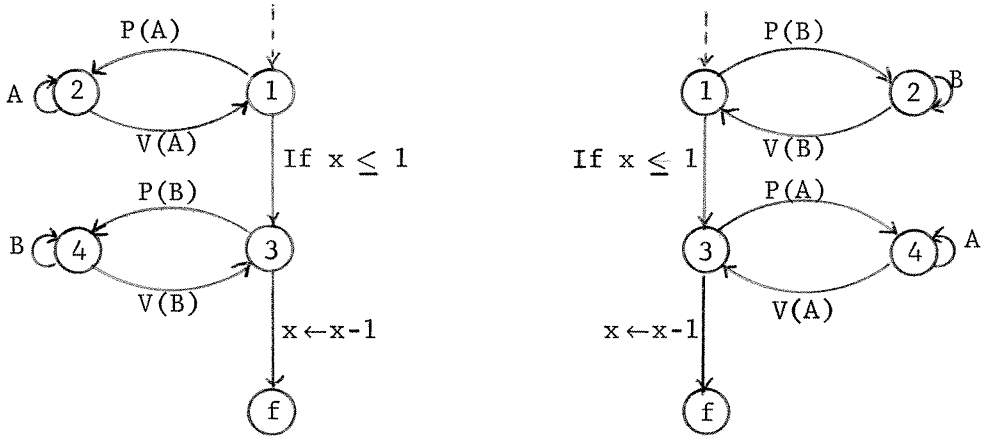

Figure 4.4 A Restless System

acquires a value of 2 before either process reaches its state 3, the system can never reach its resting state even though both processes are using a resource, making transitions, and thereby (presumably) doing useful work. It does not fit Coffman, Elphik, and Shoshani's definition of deadlock because condition 2 does not hold. It is not the allocation of resources which is the problem, so Holt's definition does not apply, and Haberman's does not because it cannot necessarily be said that no "progress" is being made. Finally, even the graph definition of deadlock does not apply because each process always has a non-ϵ transition. The four system states (1,1,2), (1,2,2), (2,1,2), and (2,2,2) form a region with no exit. We might call such states S-states for Sisyphus who, no matter how long he worked, could never rest. Notice that deadlock states are just a special case of S-states in which a set of knotted processes have no non-ϵ transitions. To put it another way, if the first condition in the definition of "wait for" was changed so that the blocked processes could make non-ϵ transitions but just could not reach a final state, then S-states would be those in which a set of processes formed a knot.

4.5 Analyzing a System for Deadlock

Analyzing a system for potential deadlock will involve generating some or all of a list, R, of the reachable system states. How much of R must be generated depends in part on the starting point of the analysis. Ideally, one would observe a reachable state D, in which two or more processes were blocked and then determine whether or not the blocked processes formed a knot. If no deadlock state D is suspected, or if we are not sure it is reachable, then the whole list R must be generated. In this case there is a general method for finding regions of states with no path to a final state, whether these states are deadlock states or just S-states. First, generate the list R of reachable states; R should be sorted into a random-access structure such as an array so efficient search methods can be used. Next, generate a second list, R', of states which can reach a final state: Begin by putting all of the system final states from R onto R', and then recursively add to R' states of R at least one of whose successors are already on R' until no new states can be added. Finally, those states, if any, on R but not on R' are the S-states and perhaps deadlock states. Obviously, this method is impractical for all but the smallest and simplest systems.

To illustrate the simplest use of these definitions and methods, let us return for the last time to the mutual exclusion example of Figure 4.3. Once again we want to generate a list of the reachable states of the system, this time examining each state to see if it is a deadlock state. Just such a list appears as the first column of Table 4.1. The fourth element of that list, state (1,1,2) has no legal transitions; both processes are blocked and each could unblock the other if it could make the transition from state 3 to 4. Thus there is a knot $P_1 \lessdot P_2 \lessdot P_1$. In fact the state (1,1,2) has no successors at all and is a deadlock state. Thus the system of Figure 4.3 has a potential deadlock and is an unsatisfactory solution to the mutual exclusion problem. (State (F,F,0) has no successors either, but it is not a deadlock state because the processes are in their final states and thus it is a system final state.)

4.6 Permanent Blocking

Even though a system of parallel processes may not have any potential deadlock nor any S-states, it may still have another possibly undesirable property which Holt [13] calls permanent blocking: "some process's request is never granted ... even though it is possible for the process to receive its request." This can occur when one process is consistently "unlucky" in that at each point in time when it could legally make a certain transition, some other process quickly makes a transition which carries the system to a state from which the unlucky process again cannot make its transition.

The notion of permanent blocking is a somewhat intuitive one; there does not seem to be a universally accepted formal definition. Two of the essential features of the notion seem to be that the blocked process is somehow "actively trying" to perform some particularly important transition, and that while it is failing, at least one other process is performing the same transition repeatedly. For the purposes of this study we shall give a semi-formal definition based on traces through the state graph.

A system of processes is said to have a potential permanent blockage when its state graph has a loop from a system state back to the same state and a process i such that

(1) the i^{th} process makes no non-ϵ transition on the loop, and

(2) some other process does make a non-ϵ transition on the loop, and

(3) from at least one state on the loop there is a reachable path on which the i^{th} process does make a non-ϵ transition.

Let us call states on such a loop permanent blockage states. Notice that there is no relation between deadlock states or S-states and permanent blockage states; from permanent blockage states the system can reach its resting states but just may happen not to.

Permanent blockage is in one sense a purely academic concept. Each time the system traverses the loop, there is a certain probability, however small, that it will leave the loop and that the i^{th} process will be allowed to proceed. Thus there is a vanishingly small probability that the i^{th} process will really be blocked permanently. Nevertheless, the concept is very useful for characterizing situations in which a process may be blocked for an unacceptably long time. In many situations, having a process accidentally blocked for a few minutes is as bad as having it blocked permanently. Thus it is worthwhile trying to characterize such conditions and to develop methods of detecting them.

Such circular traces in the state graph can be found by exhaustive search, but such a procedure is quite likely to be impractically long. The only obvious heuristic is to center the search around states in which one process is "just about" to perform the critical action. Thus as a practical matter, the presence or absence of potential permanent blockage is not as amenable to automatic verification as some other properties of systems.

4.7 An Example

As a final unified example of all of the techniques of this chapter, let us consider Dijkstra's [7] elegant solution to the mutual exclusion problem. This was the first satisfactory interlock protocol which involved only testing and setting variables in common store by the processes rather than a monitor, queues, or static priority.

For n processes the common store consists of two Boolean arrays, b, c[1:n] each initialized to TRUE, and integer k, $1 \leq k \leq n$. The algorithm for the i^{th} process is:

```
Li0: b(i) := FALSE;
Li1: if k≠i then
Li2: begin c(i) := TRUE;
Li3: if b(k) then k := i;
     go to Li1
     end
      else
Li4: begin c(i) := FALSE:
      for j := 1 step 1 until n do
       if j ≠ i and not c(j) then go to Li1
     end;

     critical section;

     c(i) := TRUE;  b(i) := TRUE:
     remainder of program in which stopping is allowed;
     go to Li0
```

A set of proofs about a system of three processes using this protocol would be a more convincing verification of the protocol than one about only two processes. As it happens, the methods of this chapter *are* still practical for the 3-process case, but the 2-process case makes a more digestible example.

Two different graphs are presented in Figures 4.5 and 4.6, each representing two processes executing Dijkstra's algorithm. It is instructive to compare these two graphs. Both graphs accurately represent the state transitions of the individual processes. In the first, Figure 4.5, the graphs of the individual processes are more compact and have fewer states; therefore, in this representation the system will have fewer states, so this would seem to be the more attractive representation. The graph in Figure 4.6 differs from the first only in putting conditions and their subsequent actions on separate arcs. Thus in the second representation processes have states such as 4 when they have tested b but not yet reset k, a state entirely missing from the first representation. Such distinctions *are* in general necessary in order to construct convincing proofs, particularly about deadlock and blockage.

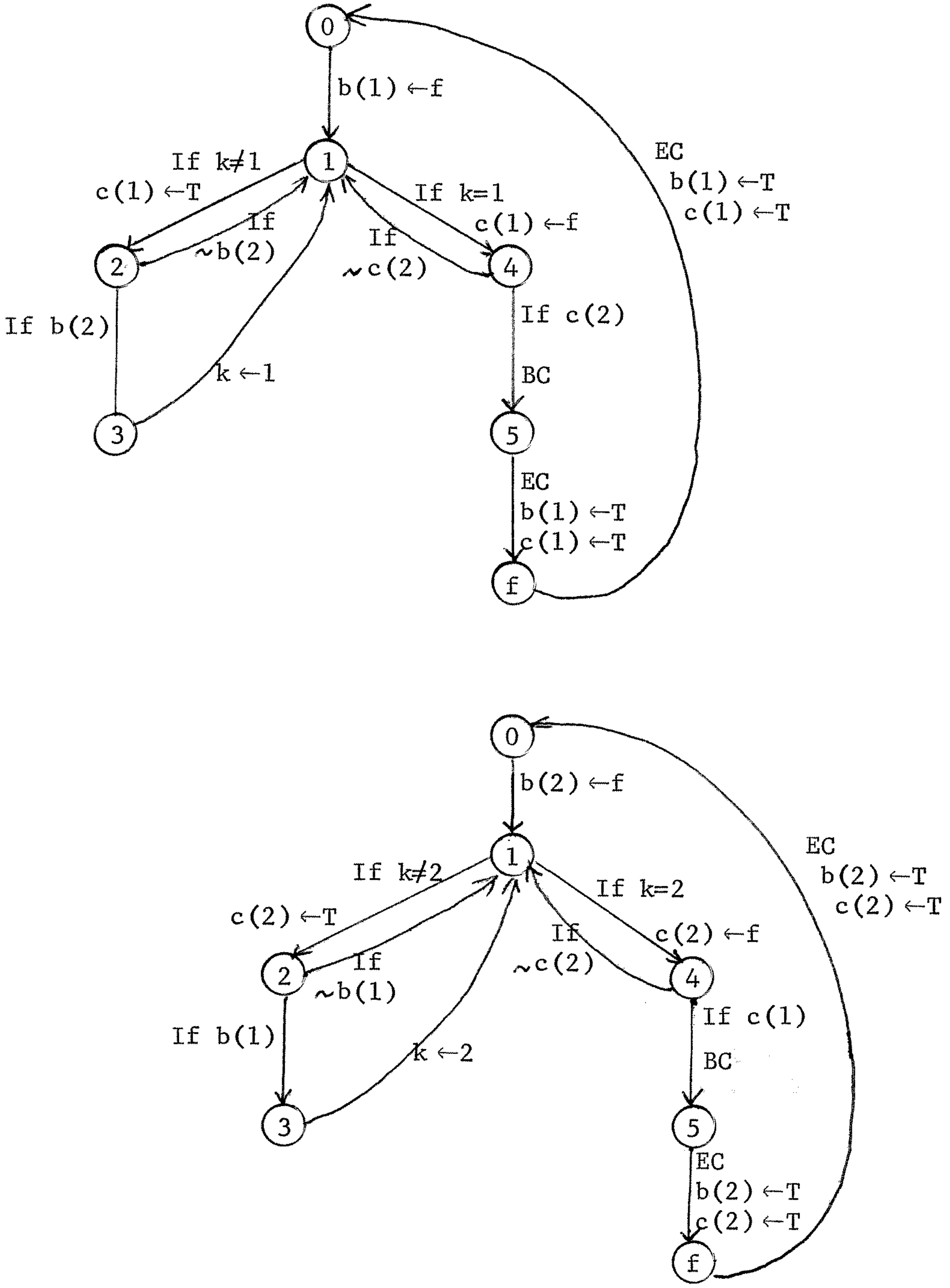

Figure 4.5 Inaccurate Representation of Two Processes

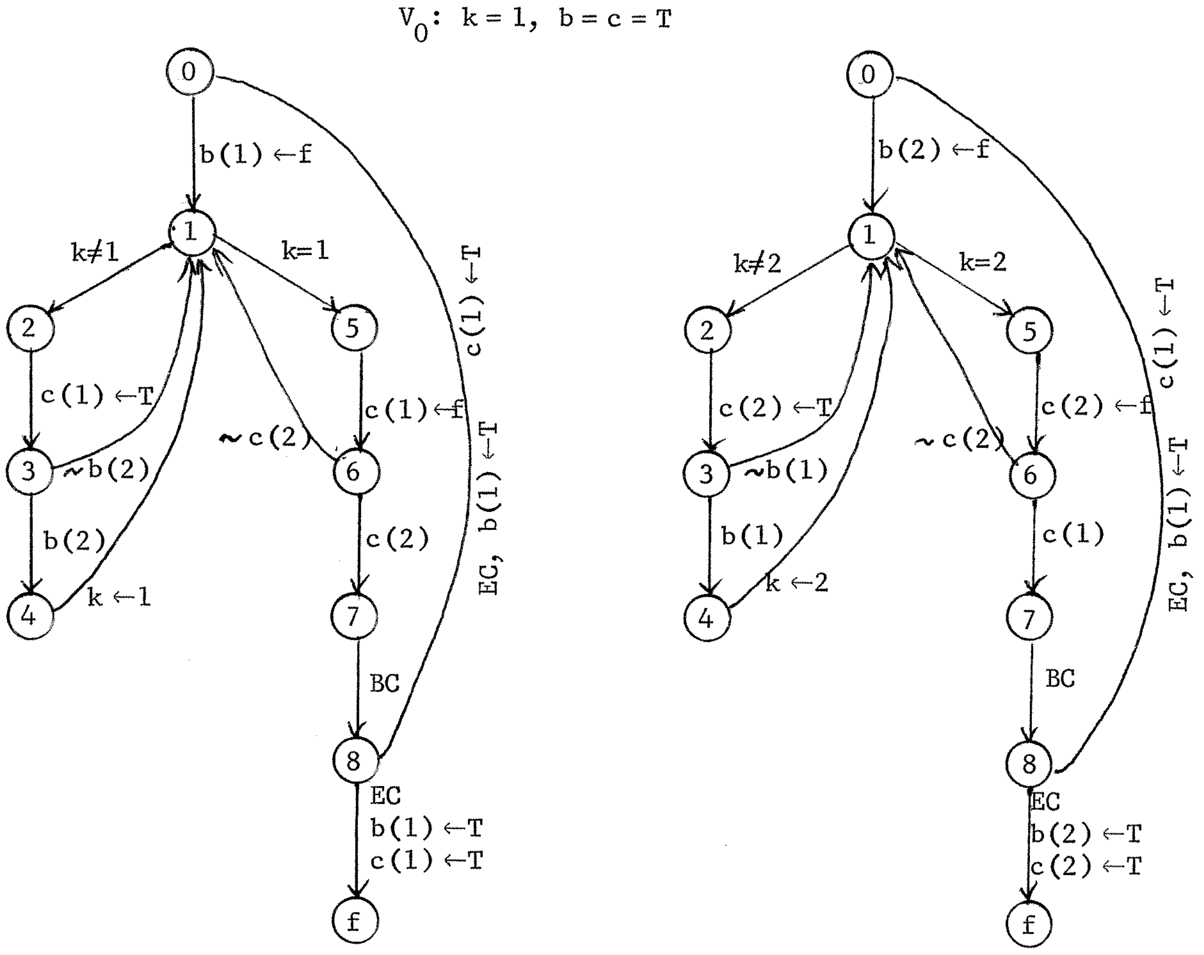

Figure 4.6 Accurate Representation of Two Processes

Typically, it is just when one process tests a condition and then another quickly changes it before the first can act that interlock schemes fail and undesirable results occur. The procedures presented in this study analyze graphs; the results of the analysis will apply to processes or systems of processes only insofar as the graphs accurately reflect the relevant properties of the processes or systems.

We shall analyze the system of Figure 4.6 in order to verify that it maps correctly to the prototype of Figure 4.2 and to determine whether or not it contains potential deadlocks or permanent blockages. In each case the first step is to generate all the reachable states of the system. The task looks formidable because, since each process has 10 states and there are 5 common variables, each binary, there are $10^2 \times 2^5 = 3200$ possible states of the system. However, it turns out that there are only about 200 reachable states!

In a fashion similar to that of Table 4.1, Table 4.2 lists reachable states, pairs (s,p), and the successors of s. However, not all reachable states are listed because two shortcuts have been taken. First, observe that whenever we reach states such as 26 in which one process has halted, its successors are not generated. Such states are marked with an asterisk in the "successors" column. It is obvious that in this example each of the processes in this system can, if running alone, reach a resting state. Thus neither deadlock nor permanent blocking can occur if only one process is running, and we need not check these traces further for these two conditions. It is also assumed that each process has previously been mapped successfully to the prototype of Figure 2.1 so that no illegal sequences of actions can occur when only one process is running, and we can safely ignore these traces. Unfortunately, this shortcut does not greatly reduce the number of states to be considered since many of the successors to states like 26 can be reached

TABLE 4.2

ANALYSIS OF THE SYSTEM OF FIGURE 4.6

Pairs in L									Successors of s
s								p	
	s_1	s_2	k	b(1)	b(2)	c(1)	c(2)		
0	0	0	1	T	T	T	T	0	1, 2
1	0	1	1	T	f	T	T	0	3, 4
2	1	0	1	f	T	T	T	0	3, 5
3	1	1	1	f	f	T	T	0	6, 7
4	0	2	1	T	f	T	T	0	6, 8
5	5	0	1	f	T	T	T	0	7, 9
6	1	2	1	f	f	T	T	0	10, 11
7	5	1	1	f	f	T	T	0	11, 12
8	0	3	1	T	f	T	T	0	10, 13
9	6	0	1	f	T	f	T	0	12, 14
10	1	3	1	f	f	T	T	0	3, 15
11	5	2	1	f	f	T	T	0	15, 16
12	6	1	1	f	f	f	T	0	16, 17
13	0	4	1	T	f	T	T	0	18, 19
14	7	0	1	f	T	f	T	0	17, 20
15	5	3	1	f	f	T	T	0	7, 21
16	6	2	1	f	f	f	T	0	21, 22
17	7	1	1	f	f	f	T	0	22, 23
18	0	2	1	T	f	T	T	0	24, 59
19	1	4	1	f	f	T	T	0	24, 25
20	8	0	1	f	T	f	T	1	0, 23, 26
21	6	3	1	f	f	f	T	0	12, 27
22	7	2	1	f	f	f	T	0	27, 28
23	8	1	1	f	f	f	T	1	1, 28, 29
24	1	1	2	f	f	T	T	0	sym 3
25	5	4	1	f	f	T	T	0	30, 31
26	f	0	1	T	T	T	T	0	* 29
27	7	3	1	f	f	f	T	0	17, 32
28	8	2	1	f	f	f	T	1	4, 32, 33
29	f	1	1	T	f	T	T	0	* 33
30	5	2	1	f	f	T	T	0	34, 35
31	6	4	1	f	f	f	T	0	35, 36
32	8	3	1	f	f	f	T	1	8, 23, 37
33	f	2	1	T	f	T	T	0	* 37
34	5	5	2	f	f	T	T	0	38, 39
35	6	1	2	f	f	f	T	0	38, 40
36	7	4	1	f	f	f	T	0	40, 41
37	f	3	1	T	f	T	T	0	* 46
38	6	5	2	f	f	f	T	0	42, 43
39	5	6	2	f	f	T	f	0	42, 43
40	7	1	2	f	f	f	T	0	43, 45
41	8	4	1	f	f	f	T	1	13, 45, 46
42	6	6	2	f	f	f	f	0	47, 48
43	7	5	2	f	f	f	T	0	49, 50

TABLE 4.2 (CONTINUED)

	Pairs in L								Successors of s
s	s_1	s_2	k	b(1)	b(2)	c(1)	c(2)	p	
44	5	7	2	f	f	T	f	0	51, 52
45	8	1	2	f	f	f	T	1	18, 50, 53
46	f	4	1	T	f	T	T	0	* 53
47	1	6	2	f	f	f	f	0	54, 55
48	6	1	2	f	f	f	f	0	55, 56
49	7	6	2	f	f	f	f	0	57, 58
50	8	5	2	f	f	f	T	1	58, 59, 60
51	6	7	2	f	f	f	f	0	61, 62
52	5	8	2	f	f	T	f	2	58, 63, 64
53	f	1	2	T	f	T	T	0	* 60
54	2	6	2	f	f	f	f	0	65, 66
55	1	1	2	f	f	f	f	0	65, 66
56	6	5	2	f	f	f	f	0	42, 67
57	7	1	2	f	f	T	f	0	68, 69
58	8	6	2	f	f	f	f	1	68, 70, 71
59	0	5	2	T	f	T	T	0	sym 5
60	f	5	2	T	f	T	T	0	* 71
61	6	8	2	f	f	f	f	2	72, 73, 74
62	1	7	2	f	f	f	f	0	72, 75
63	5	0	2	f	T	T	T	0	30, 73
64	5	f	2	f	T	T	T	0	* 74
65	2	1	2	f	f	f	f	0	76, 77
66	3	6	2	f	f	T	f	0	78, 79
67	1	5	2	f	f	f	f	0	47, 77
68	8	1	2	f	f	f	f	1	80, 81, 82
67	7	5	2	f	f	f	f	0	49, 80
70	0	6	2	T	f	T	f	0	sym 9
71	f	6	2	T	f	T	f	0	*
72	1	8	2	f	f	f	f	2	83, 84, 85
73	6	0	2	f	T	f	T	0	35, 86
74	6	f	2	f	T	f	T	0	*
75	2	7	2	f	f	f	f	0	78, 83
76	3	1	2	f	f	T	f	0	87, 88
77	2	5	2	f	f	f	f	0	54, 87
78	3	7	2	f	f	T	f	0	89, 90
79	1	6	2	f	f	T	f	0	89, 91
80	8	5	2	f	f	f	f	1	58, 92, 93
81	0	1	2	T	f	T	f	0	88, 92
82	f	1	2	T	f	T	f	0	* 93
83	2	8	2	f	f	f	f	2	90, 94, 95
84	1	0	2	f	T	f	T	0	88, 94
85	1	f	2	f	T	f	T	0	* 95
86	7	0	2	f	T	f	T	0	40, 96
87	3	5	2	f	f	T	f	0	66, 97
88	1	1	2	f	f	T	f	0	97, 98

TABLE 4.2 (CONTINUED)

Pairs in L									Successors of s
s	s_1	s_2	k	b(1)	b(2)	c(1)	c(2)	p	
89	1	7	2	f	f	T	f	0	sym 17
90	3	8	2	f	f	T	f	2	sym 32
91	2	6	2	f	f	T	f	0	sym 16
92	0	5	2	T	f	T	f	0	97, 70
93	f	5	2	T	f	T	f	0	* 71
94	2	0	2	f	T	f	T	0	99, 100
95	2	f	2	f	T	f	T	0	*
96	8	0	2	f	T	f	T	1	45, 101, 102
97	1	5	2	f	f	T	f	0	79, 103
98	2	1	2	f	f	T	f	0	76, 103
99	2	1	2	f	f	f	T	0	104, 105
100	3	0	2	f	T	T	T	0	sym 8
101	0	0	2	T	T	T	T	0	sym 0
102	f	0	2	T	T	T	T	0	* 53
103	2	5	2	f	f	T	f	0	87, 91
104	3	1	2	f	f	T	T	0	sym 10
105	2	6	2	f	f	f	T	0	54, 106
106	3	5	2	f	f	T	T	0	66, 107
107	1	5	2	f	f	T	T	0	79, 108
108	2	5	2	f	f	T	T	0	91, 106

by other paths and so appear on the list L anyway.

The second shortcut takes advantage of the fact that in this example the two processes are identical and that for every reachable system state s there is another reachable state s' "symmetric" to s in the following sense: if s = $(a_1, a_2, k, b(1), b(2), c(1), c(2))$ then s' = $(a_1', a_2', k', b(1)', b(2)', c(1)', c(2)')$ is also a reachable state where $a_1' = a_2$, $a_2' = a_1$, $k' = (k \bmod_2) + 1$, if $b(1) = b(2)$ then $b(1)' = b(2)' = b(1)$, else $b(1)' = b(2)$ and $b(2)' = b(1)$, and similarly if $c(1) = c(2)$ then leave them unchanged else switch them. For example, if s = (1,1,1,F,F,T,T) then s' = (1,1,2,F,F,T,T) and if s = (5,0,1,F,T,T,T) then s' = (0,5,2,T,F,T,T). Now notice that if s has a successor p then s' will have p' as a successor; whichever process caused the transition from s to p, the other process will have an identical legal transition which carries the system from s' to p'. Thus if s is not a deadlock state, s' cannot be and we do not need to check it.

Similarly, if $s_1,\ldots,s_n$ is not an illegal trace then $s_1',\ldots,s_n'$ is not illegal either, and if we check all traces beginning at s_1 we do not need to check any of the traces beginning at s_1'. For this reason, of the two initial states of the system, (0,0,1,T,T,T,T) and (0,0,2,T,T,T,T), we shall use only the first one, secure in the knowledge that if all traces from that state map to the prototype correctly, then so must all from the other.

Finally, consider the two traces $t_1 = s_1,\ldots,s_{i-1}, s_i,s_{i+1},\ldots,s_n$, and $t_2 = p_1,\ldots,p_{i-1}, s_1',s_{i+1}',\ldots$. It is contended that if we have followed t_1 out and know that it maps correctly, and if we have followed t_2 up to and including the transition from p_{i-1} to s_i' and the mapping has also been correct, then we need check no further along t_2. This is because we have put all of the successors of s_i, such as s_{i+1}, on L and either have checked or will check that all the transitions from s_i map it and its successors correctly, so by the symmetry of the graph all the ways of leaving state s'_i must also map it correctly. This argument applies recursively to each of the successors of s'_{i+1}, etc. on out to the end of t_2. Thus whenever we reach a state of the graph such as 59 whose symmetric state, 5, is already in L, we shall not add the successors of 59, having already added all the successors of 5.

By taking advantage of the symmetry in this particular example we can cut almost in half the number of reachable states which must be considered by the mapping algorithm and in checking for deadlock possibilities. The symmetry will not, however, necessarily help us in checking for potential permanent blockage; the non-existence of a circular trace fitting the definition in one "half" of the graph guarantees that there is not one in the other half but does not guarantee that there is not one which extends across both halves.

We are now ready to analyze the system. We can use the system comparison algorithm with the list of Table 4.2. The mapping is successful; each system state maps to exactly one state of the prototype, and so only one process at a time can be in its critical section. It is obvious that there are no deadlock states. However, in this system one process can be permanently blocked while the other repeatedly performs BC-EC pairs. There are an infinite number of circular traces fitting the definition of permanent blockage including traces in which one process performs arbitrarily many more transitions than the other while remaining blocked. For example, there is the trace 1, 3, 6, 10, 15, 7, 11, 16, 21, 12, 17, 22, 28, 32, 23, 1. Knuth [18] was the first to comment on this property of Dijkstra's protocol. Since we did find a potential permanent blockage by examining the states in Table 4.2, there is no need to generate the rest of the reachable states. Had we not found one, we would still not know whether or not one existed.

This example illustrates several points about state graph analysis. First, such analysis is only as good as the graphs it works on. Whether the graphs are being produced semi-automatically by programs such as TRACE or by hand, great care must be taken to insure that they accurately reflect the relevant properties of the processes or systems they represent.

Second, there are at least three factors which help make state graph analysis of systems of processes more nearly practical than it might first appear: one is that we need deal only with those states of the system which are actually reachable, and that the number of such states is likely to be a small fraction (in this example about 9%) of the apparently possible states of the system. Further, we do not need to explicitly represent even this smaller actual state graph; a simple list of the reachable states along with graphs of the processes retains all the necessary information. Another is that once we have constructed such a representation of the system, it can be used more than once to verify more than one sequencing property of the system, whether these properties are expressed by prototype graphs or in some other way such as the definition of deadlock states. Finally, most of the steps of these methods of analysis are algorithmic and can be implemented as computer programs. While it took five hours to work out and check this example by hand, it could have been done in a matter of seconds by a program given the prototype and either the graphs of Figure 4.6 or two different sets of source code for the two processes.

Still, even with all of these favorable considerations, given present computer speeds and memory sizes, such analysis will be totally impractical for real systems consisting of many real processes. The above example was not an analysis of a real system. But it did deal with a real problem. The verification of Dijkstra's algorithm was not trivial, and herein lies one use to which the methods of this chapter can be put immediately. Notice the relationship between the use to which we put the methods of Chapter II and the examples in this chapter; when analyzing one process, we were essentially asking, "Does the process always observe a given protocol?"; in analyzing a system of processes we asked, "Does the protocol work?" While in the first case we would probably only be interested in asking the question about real programs, it should usually be possible to answer the second by analyzing a system of only a few artificially simple processes. Thus, for example, it would be quite practical to verify Knuth's [18] improvement to Dijkstra's protocol. In fact, these methods should prove useful in verifying almost any protocol proposed to coordinate parallel processes in almost any way.

CHAPTER V

SUMMARY AND CONCLUSION

5.1 Summary of Previous Chapters

In this study, a method has been developed for analyzing the order in which a program performs its operations. State graphs are used to model a program's behavior. Prototype graphs are proposed as a general and natural way to make assertions about sequencing properties of programs. An algorithm for proving such assertions has been given which is conceptually based on formal language theory.

The problems of automating the various steps in the analysis method are addressed in some detail. Techniques called folding and splitting for manipulating the graphs in order to represent just the necessary information and no more allow the analysis of even fairly large programs in limited space and time. The most difficult problem is building from the source code graphs which accurately reflect all the sequencing properties of a program; this step has not been and probably will not be completely automated. A program implementing these techniques has been used to help verify some properties of real operating system programs.

Finally, the method is extended to apply to analysis of the sequencing properties of systems of parallel processes. It is argued that a whole system can reasonably be viewed as a single process which performs its operations sequentially and can be modeled with a single state graph. It is then observed that not only can these methods be used to prove assertions about the order in which the system performs operations, but to analyze other well-known system properties such as blockage and deadlock. While it will not be practical to use these methods on large real systems, they should be very helpful in analyzing formal systems representing proposals for the coordination of multiple parallel processes.

5.2 Suggestions for Further Research

There are several areas relating to this work which need further study. Certainly a better and more thorough treatment can be given to the problems of translating or compiling computer programs into graphs. More general graph models might also be developed. The state graph model used in this study to represent programs was developed with operating system programs written in assembly language in mind. Perhaps the model could be altered to allow other program constructions such as recursion to a fixed depth. Another possibility worth investigating is making the whole method into a man-machine interactive system so that, for example, critical variables could be designated on-line. Perhaps the most interesting development for the sequencing analysis method worked out in this study would be its integration into a larger program analysis package. This idea will now be explained in more detail.

5.3 Graphs as Models

The model for computer programs used throughout this study has been the state graph. Directed graphs seem to be a very natural representation for processes and frequently appear as the basis for program analysis. Flowcharts are the most familiar example, but the construction of a directed graph almost exactly like a flowchart is also a basic step underlying program verification by inductive assertion, even though the graph is not always presented in the final proof. Graphs are also used implicitly in other induction methods as well as in data-flow analysis. Perhaps their widespread use is because any analysis of the semantics of all but the most trivial programs must in some way deal with the existence of different execution sequences, and directed graphs are the most natural way found so far of representing sets of sequences.

It is appropriate to pause here briefly to examine the relationship between state graphs and flowcharts. The relationship is that, in the information about the program which the two models are capable of representing, they are equivalent. Consider the following simple algorithm for converting a state graph of a program to the equivalent flowchart:

Graph Conversion Algorithm

1. For every node with more than one outgoing arc: Divide each of its outgoing arcs into two consecutive arcs by inserting at the middle of the arc a new node. Put the action-label of the original arc on the outgoing arc from this new node; put the condition, if any, on the original arc on the incoming arc to this new node.
2. Move the action-labels on the arcs back to the node from which the arc leaves. Similarly, move the conditions from the arcs back to the previous node and form appropriate predicates at the node, labeling the arcs only with the values of the predicate. If desired, move state information from each node back to each of its incoming arcs.

Figure 5.1 illustrates the use of this algorithm: G_1 is a sample state graph segment, G_i is the result of applying step one of the algorithm, and G_2 is the flowchart equivalent to G_1.

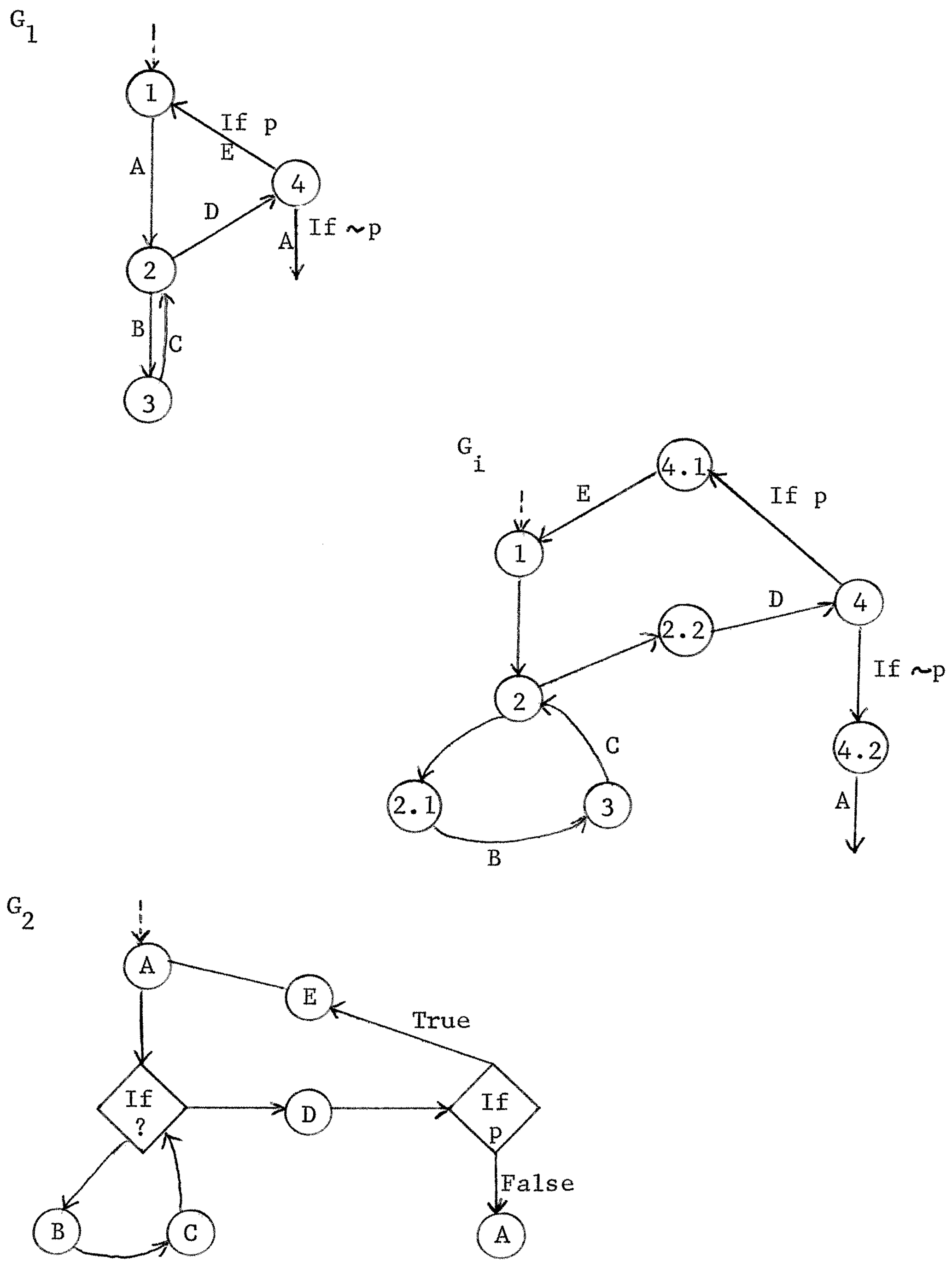

Figure 5.1 Use of the Graph Conversion Algorithm

Any property of the program which was expressed by G_1 is also expressed by G_2, and any statement about the program which could be proven by analyzing G_1 could also be proven using G_2. Thus, which of the two basic kinds of directed graph is used and whether nodes or arcs represent actions or states is a matter of convenience and will be determined by the kind of analysis to be performed. If more than one kind of analysis is to be done requiring both graph forms, one form can be built from the code and the second obtained cheaply by transforming the first.

5.4 Conclusion

The ubiquity of graphs in program analysis suggests attempting to base a large analysis package on some common graph representation of the programs. The idea would be to translate programs written in many different languages into the common graph language and then let many different analysis routines operate on this one program representation. In fact, just such a project is currently being undertaken at The University of Texas at Austin.

Whether by their inclusion in that project or alone, the sequencing analysis methods presented in this study will allow programmers and system designers to include ordering statements in their specifications of correctness and proofs of such statements in their demonstrations of correctness.

REFERENCES

1. Anderson, J. W. Primative process level modeling and simulation of a multiprocessing computer system. TR-32, Department of Computer Sciences, University of Texas, Austin, May 1974.

2. Baer, J. L. A survey of some theoretical aspects of multiprocessing. ACM Computing Surveys 5, 1 (March 1973), 31-80.

3. Bredt, T. H. Analysis of operating system interactions. Proceedings of the AICA Congress on Theoretical Informatics, Institute for the Elaboration of Information, University of Pisa, Pisa, Italy, March 1973, 253-281.

4. Coffman, E. G., Elphick, M. J., and Shoshani, A. System Deadlocks. ACM Computing Surveys 3, 2 (June 1971), 70.

5. Conway, M. A multiprocessor system design. AFIPS Conference Proceedings 33 (FJCC 1963), Spartan Books, Baltimore, 1963, 139-146.

6. Dijkstra, E. W. Cooperating sequential processes. Programming Languages (F. Genuys, ed.), Academic Press, 1968, 48-112.

7. Dijkstra, E. W. Solution of a problem in concurrent programming control. Communications of the ACM 8, 9 (September 1965), 569.

8. Gilbert, P., and Chandler, W. J. Interference between communicating parallel processes. Communications of the ACM 15, 6 (June 1972), 427-437.

9. Good, D. I. Toward a man machine system for proving program correctness. Dissertation, University of Wisconsin, 1970.

10. Habermann, A. N. Prevention of system deadlocks. Communications of the ACM 12, 7 (July 1969), 373.

11. Hedetniemi, S. T. Homomorphisms of graphs and automata. Technical Report, Communication Sciences Program, The University of Michigan, 1966.

12. Hetzel, W. C. Program Test Methods, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

13. Holt, R. C. Comments on prevention of system deadlocks. Communications of the ACM 14, 1 (January 1971), 36.

14. Hopcroft, J. E., and Ulman, J. D. Formal Languages and Their Relation to Automata. Addison-Wesley, Reading, Mass., 1969.

15. Howard, J. H. The coordination of multiple processes in computer operating systems. TSN-16, Computation Center, University of Texas, Austin, 1970.

16. Howard, J. H., and Alexander, W. P. Analyzing sequences of operations performed by programs. Program Test Methods (Hetzel, W. C., ed.), Prentice-Hall, Englewood Cliffs, New Jersey, 1973, 239-254.

17. Johnson, D. S. A process-oriented model of resource demands in large, multiprocessing computer utilities. TSN-29, Computation Center, University of Texas, Austin, August 1972.

18. Knuth, D. E. Additional comments on a problem in concurrent programming control. Communications of the ACM 5, 9 (May 1966), 321.

19. London, R. L. Correctness of a compiler for a LISP subset. Proceedings of an ACM conference on proving assertions about programs, January 1972, 121-127.

20. London, R. L. The current state of proving programs correct. Proceedings, ACM National Conference. 1972, 39-46.

21. Manna, Z., Ness, S., and Vuillemin, J. Inductive methods for proving properties of programs. Proceedings of an ACM conference on proving assertions about programs, January 1972, 27-50.

22. Schwetman, H. D. A study of resource utilization and performance evaluation of large-scale computer systems. TSN-12, Computation Center, University of Texas, Austin, 1970.

VITA

William Preston Alexander, III, was born in Sacramento, California, on September 26, 1942, the son of Mr. and Mrs. William Preston Alexander, Jr. He graduated from Waco High School, Waco, Texas, in 1960 and received the degree of Bachelor of Arts in Philosophy from Rice University in Houston, Texas, in May 1964. He then served for two years as a Peace Corps Volunteer in Ghana as a high school mathematics teacher. He entered the Graduate School of The University of Texas at Austin in September 1966 and received the degree of Master of Arts in Computer Sciences in August 1971. In 1973 he married Rosemary Schwetman of Waco, Texas.

Permanent address: 3202 Park Lake Drive
Waco, Texas 76708

This dissertation was typed by Dorothy W. Baker.