ANALYSIS OF SEQUENCING IN COMPUTER PROGRAMS AND SYSTEMS

by

William Preston Alexander, III

August 1974 TR-35

ANALYSIS OF SEQUENCING IN COMPUTER PROGRAMS AND SYSTEMS*

by

William Preston Alexander, III

*This paper is suported in part by the National Science Foundation grants GJ36424 and GJ1084.

ANALYSIS OF SEQUENCING IN COMPUTER PROGRAMS AND SYSTEMS

by

WILLIAM PRESTON ALEXANDER, III, B.A., M.A.

TR-35

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 1974

ACKNOWLEDGMENTS

I would like to thank the members of my committee, each of whom helped in some unique way, and especially Dr. J. C. Browne for his encouragement and Dr. J. H. Howard for his time and effort.

I also acknowledge my debt to my parents, who are largely responsible.

This work was supported in part by NSF grants GJ36424 and GJ1084.

May 1974

ABSTRACT

Methods are developed for determining the orders in which a computer program performs its operations and for verifying from its source code whether a program adheres to any given sequencing rule. These methods are based on analysis of the program's state graph, and techniques are presented for compressing the graph to eliminate unnecessary detail and for expanding it to add necessary distinctions. A program is described which automates most of these methods and which has been used to analyze parts of a real operating system. The methods are extended to apply to systems of parallel processes. Systems are modeled as state graphs and analyzed for the order in which they perform operations as well as for blockage properties such as deadlock.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

1.1 The Problem

Many important properties of computer programs can be expressed in terms of the order in which they perform actions. The order in which a program produces its external effects can be important in many real time applications. Most methods for coordinating multiple parallel asynchronous processes require that each process observe some protocol dictating the order in which transactions between the processes may occur. Other properties involve the internal workings of the memory function, such as the usually desirable property that a program assign a value to a cell before using its contents (the initialization problem). For large and complex programs, traditional testing and debugging methods cannot assure that a program has such properties, and none of the formal proof procedures for programs are designed to prove ordering properties.

The first objective of this study is the development and automation of a method for stating and proving assertions about the order in which a program performs its operations. A second goal is to develop a method for stating and proving similar assertions about systems of processes running in parallel. The methods must verify rather than simulate or test; the source code of the object programs must be analyzed, not executed.

1.2 Summary of the Research

In this study a comprehensive method is developed for making and proving statements about the order in which programs perform actions. Individual programs are modeled empirically as directed graphs called state graphs which are generated from the source code, and paths through this graph indicate possible sequences of actions by the program. The statements to be proved are made in terms of prototype state graphs, small compact graphs specifying allowable sequences of actions. Careful attention is given to the problems of dealing with state graphs of large programs in limited space and time. Techniques are presented for expanding and compressing state graphs to reflect just the amount of detail needed and no more. Based on formal language theory, an algorithm is presented which compares the sequences represented by one graph with those of another.

The method presented for proving sequencing properties of programs is practical and can be almost completely automated. A program is described which implements most of the techniques and which has been used to analyze parts of a real operating system.

The method developed for analyzing individual processes is extended to apply to systems. Systems are modeled by their state graph, and this graph can then be analyzed to determine the order in which the system performs actions as well as for system blockage properties such as deadlock.

The program proof method presented here does not improve on or replace any existing program correctness technique; rather its importance is that it provides a whole new class of statements about programs which can be proved. The method for analyzing systems cannot be applied directly to real large-scale operating systems, but it does provide a unified model for expressing a wide class of operating system properties and should prove useful in analyzing simple systems and in evaluating protocols for coordinating parallel processes.

1.3 The Context

As more and more people learn how to program computers, the seller's market once enjoyed by programmers is turning into a buyer's market in which customers can demand correct, dependably correct, and soon even provably correct programs. For this reason, and perhaps because the challenge of applying mathematical rigor to proofs about objects with effects in the real world is irresistable, there has arisen beside the traditional interest in debugging and testing programs a growing interest in proving statements about them *a priori*. A survey of this area was recently provided by London [20]. An exhaustive bibliography by Hetzel [12] covers program testing and debugging as well as program proving.

The object of the proofs referred to is program semantics, not syntax. Throughout this study, "program" means "syntactically correct computer program." The semantics of a program is the set of all the effects of executing the program. In the broadest sense, "correctness" means that the semantics of a program match the desire of the user or the intent of the writer. Obviously then, we do not prove programs correct; in the current state of the art we prove that a program has some property or, more specifically, that its execution has certain effects or that its effects have certain properties.

Perhaps because this area in computer science has taken models as well as personnel from mathematics, programs have most often been viewed as functions whose range and domain are the memory of the (virtual) machine upon which they execute. One can take as argument the initial state, or the vector of initial values in all the memory cells, the value as the final state, or vector of final values, and conceive of a program as the function which maps one set of vectors to the other. There are many techniques for proving statements, usually expressed in first order predicate calculus, about the program-as-function. These include computational induction, structural induction, recursion induction, fixed-point induction, and inductive assertions. Manna, Ness, and Vuillemin [21] discuss all of these methods and prove some equivalences among them. Many interesting properties of programs can be expressed using these techniques, and London [19] has even been able to show that a simple but real compiler produces correct code in the sense that the compiler-produced machine code computes the same function as the source code.

However, a memory function is not a completely satisfactory model for a computer program. A program operates over a length of time, while a function is usually thought of as acting instantaneously. If a function produces as its value a vector, the order in which the values in the vector are produced does not usually matter. But because a program operates in time, the order in which it performs its actions may be important.

This is not to say that the inductive proof methods do not take any account of the sequential nature of programs. They all can allow composition of functions and provide computation rules for order of evaluation. Even more explicitly, the inductive assertion method involves explicitly marking paths through the program and making assertions which apply before, and others after, certain path segments have been executed. Perhaps the most explicit recognition of time is Good's [9] addition of a variable, t, representing the time at which a given point in the program is reached, to the domain of the assertions. The problem with using the augmented inductive assertion method to prove statements about the order in which programs perform operations is that unless one already knows the order it is very difficult to assign values to t at each point in the program or to make statements about it, particularly in any standard predicate calculus. This is because when a program has loops one must assign a different t for each time the program reaches a given point in the loop. Otherwise, if two different points on a loop were simply assigned an unsubscripted t and t', respectively, it might be the case that both $t < t'$ and $t > t'$ were true.

In fact this extension to the inductive assertion method was proposed to allow one to prove additional statements about the memory function. It can also be said of the path notation in the inductive assertion method and of the composition of functions that they all allow the prover to more accurately take into account the sequential nature of programs in order to make or prove statements about the program as memory function, but they are not well suited to stating or proving sequencing properties of programs. For this important class of properties, a new method is needed.

1.4 Summary of Chapters

The structure of the rest of this study is as follows: Chapter II introduces state graphs and develops the method for proving a wide class of sequencing assertions about individual computer programs. Chapter III describes an actual program written to implement the method of Chapter II and details the uses to which this program has been put. In Chapter IV the method of Chapter II is extended to apply to systems of parallel processes. The final chapter contains some general observations and suggestions for further work.

CHAPTER II

ANALYZING INDIVIDUAL PROCESSES

2.1 Introduction

In this chapter a method is developed for analyzing the orders in which individual computer programs perform their actions. The method consists of three steps: First the program is modeled by its state graph. State graphs are defined in the next section, and the process of building a program's initial state graph from its source code is discussed in Section 2.5. In the second step of the method the state graph is manipulated to eliminate unnecessary detail and to make additional distinctions as needed. The two main graph manipulatory techniques, folding and splitting, are explained in Sections 2.6 and 2.7. The statements to be proved about programs are embodied in small graphs called prototypes which specify allowable sequences. The third and final step in the method presented in this chapter is to compare the program's state graph with the prototype graph.

This chapter is organized so as to present as quickly as possible the broad outline of the method, leaving until later certain details. Thus the next three sections explain what a state graph is, what a prototype is, and how to compare them. The later sections explain how to get the state graph and how to assure that it compactly and accurately reflects the program's sequencing properties.

2.2 State Graphs and Traces

In the course of its execution, a program carries the machine which is interpreting it through many distinct states. By a state is meant a combination of values of all the memory cells and registers, including the program counter, used by the program.

A state graph of a program is a finite directed graph with nodes corresponding to states of the program/machine and arcs labeled with the program actions which carry the machine from one state to the next. A state graph is somewhat like a very detailed flow-chart of a program except that the roles of nodes and arcs are reversed; arcs represent actions while nodes represent states of the machine between actions.

A node may have several incident and exiting arcs, possibly with the same labels, and two nodes may have more than one arc connecting them in either direction. We use the notation $s \rightarrow s'$ to mean that there is an arc from node s to node s' and $s \xrightarrow{x} s'$ to indicate that this arc is labeled "x". Node s' is a successor of s, and s is a predecessor of s'. An arc labeled x may be called an "x-arc". An arc from a node to itself is a unit arc. $s_0 \Rightarrow s_n$ means that there is a sequence of nodes, $s_0, s_1, \ldots, s_n$, $n \geq 0$, such that $s_{i-1} \rightarrow s_i$ for $1 \leq i \leq n$. In this case we say that there is a path from s_0 to s_n and that s_n is reachable from s_0.

Notice that this model of a program leaves many aspects of its behavior uninterpreted. $s \xrightarrow{x} s'$ simply says that an action labeled "x' has occurred, but says nothing about what x is or what effects it may have. The use of directed graphs other than state graphs as models of various aspects of program structure or behavior is fairly common; a survey of some of these models is provided by Baer [2].

Typically we shall not be concerned with trying to prove an assertion about all the operations which the program may perform but about a relatively small subset of them. For example, in the case of trying to determine whether or not a program observes mutual exclusion on a given resource, the assertion to be verified will only involve the four operations "reserve the resource," "use it," "release it," and "halt," The order in which the program performs any other operations is irrelevant. Thus for the purposes of this analysis most or all other operations are completely equivalent and can be denoted by the same arc label, ϵ, which will be used to mean "uninteresting operation." Further, whole sequences of such uninteresting operations can be lumped together as a single operation represented by a single ϵ-arc to the state resulting from the execution of the last one of the sequence.

A typical feature of program execution is branching on the value of some logical expression involving memory cells or registers. If the complete state of the machine is known at all times then the results of such tests can be determined, and there will be no uncertainty about the actual paths through the state graph. However, the retention of complete information is impractical, and thus it is sometimes convenient to mark the arcs leaving a given node with the logical condition under which that transition may be made. These conditions on the arcs will be in addition to the labels on the arcs identifying actions.

Since the representation of programs by their state graphs is the basis of all analysis in this study, the terms "node" and "state" will be used interchangeably when no confusion can result.

It is convenient to assume that there is a single unique _starting state_ of the program from which all states in its graph are reachable. Real programs, however, may have several starting points, and even if they have only one it may be the case that some of their memory cells or registers are initialized externally so that there are many possible starting states. However, since we are only interested in the order in which the program performs certain interesting actions, it can do no harm to the analysis if we simply create an extra node with no incoming arcs and an ϵ-arc from it to each of the actual possible starting states and call this new node the unique starting state of the program. Throughout the rest of this study it will be assumed that this has been done to all state graphs, and this node will be called s_o. (The problem which arises when a program executes correctly with one set of initial values for some variables but may perform actions in an incorrect order for other initial values is discussed in Section 2.7.)

It is also convenient to assume that each state graph has identifiable final states representing the states of the machine after it has finished executing all of the instructions of interest. Certainly the states immediately following "halt" actions are always final states, and any states with no successors can always be automatically identified as final states. In practice it may be the case that certain programs never terminate and in fact do not even have a "halt" operation on any arc of their graph. An example might be the program which accepts input for a system. For these programs, the concept of a resting state [15] is useful. Intuitively, a resting state is the state of a process when it has "just completed" whatever it does and is about to "start over." More formally, a resting state is a state from which all other states in the graph can be reached. Notice that the assumption that all states are reachable from s_o makes it a resting state. This formal definition of resting states may include more states than the intuitive one; in fact, in a strongly connected graph all the states would be resting states. Presumably only one or a few of these would be designated as final states. In any case, it is assumed throughout the rest of this study that all state graphs have a known set of final states, whether halt states or resting states or some of each, and that these states have been explicitly so designated.

We are now ready to observe the way in which a state graph models a program's sequencing behavior. Each path through the state graph from the starting state to a final state represents a possible execution sequence of the program, and the sequence of arc labels along this path denotes a sequence of actions which the program may perform. The set of all such sequences defines all the orders in which the program may perform its actions. Such a sequence of arc labels is called a trace. The trace, α, associated with each path is the sequence of arc labels along that path. Thus the notation $s_0 \overset{\alpha}{\Rightarrow} s_n$ indicates that there is a sequence of nodes $s_0, s_1, \ldots, s_n$, $n \geq 0$, such that $s_{i-1} \xrightarrow{x_i} s_i$ for $1 \leq i \leq n$, and that $\alpha = x_1 \ldots x_n$. The length of a path is defined to be the number of arcs, not necessarily distinct, traversed along it, and the trace of a path of length 0 will be denoted by λ. In a string of symbols such as a trace, x^* denotes an arbitrary number of occurrences of the symbol x, including none, while x^+ means at least one x.

Since state graphs as defined here are finite, they may not be able to accurately model the sequencing properties of programs with loops of arbitrarily large index or with recursion to arbitrary depth. This is a limitation on all of the methods for analyzing both programs and systems presented in this study.

As a final note, observe that the complete state graph of a program actually contains all possible information about its behavior and semantics. Any assertion about the effects of a program could in theory be decided by an exhaustive examination of its state graph if such an examination were practical. Its effects as a function on memory, for example, could be discovered by examining the values of memory cells at each of the final states or at the states immediately preceding output actions. Such a proof procedure would, of course, not only be inferior to the inductive methods mentioned in the previous chapter but would be impractical for all but the most trivial programs due to the great size of most state graphs. However, state graphs do provide an adequate and practical basis for analyzing sequencing behavior in many programs.

2.3 Prototype Graphs

In any program verification application, one must decide how to form assertions about programs and how to characterize the kinds of properties which can be analyzed. State graphs, along with the idea of traces, provide a natural way of expressing sequencing properties. If the trace xyz exists in the state graph of a program, then the program may perform an x followed by a y and then a z. Thus any sequencing property of a program may be expressed in terms of the existence and/or non-existence of traces through its state graph. As a simple example, suppose we wished to verify that a certain program never performed x, y, and z consecutively in that order. It would be a relatively easy matter to devise an algorithm which searched through a state graph looking for a given trace. However, such an algorithm would be useful only if there were one or a few specific traces whose existence one wanted to deny or confirm. Typically there are an almost unlimited number of ways in which a program can perform incorrectly or correctly. A much more general way of characterizing sequencing properties is to express them as an extremely compact state graph serving as a prototype of correct action. This can be illustrated by the following example.

Suppose we have a resource, such as a table in memory, which is to be used by several asynchronous parallel processes, and that we wish to insure that no two processes ever access the table simultaneously by using Dijkstra's semaphore protocol governing entry into critical sections [6]. This protocol is defined in terms of three primitive operations:

P - Set an interlock which prevents any other process from performing the P operation until this process has performed a V operation.

V - Release the interlock.

A - Access the shared table protected by the P and V operations.

A program is defined to be correct in its usage of these operations if and only if all of the traces in its state graph from the starting state to a final state are also present from the starting state back to that state in the prototype graph of Figure 2.1.

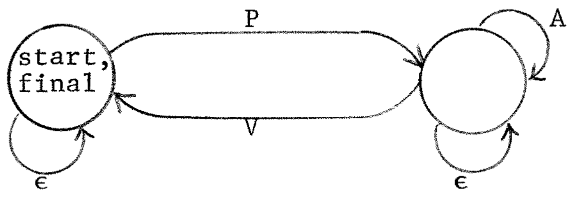

Figure 2.1. Critical Section Prototype Graph

The node on the left is both the starting and final state of the graph and represents the noncritical sections of the program. The node on the right represents the critical section of the program when it is accessing the shared table and when other programs must be prevented from doing so. This prototype graph defines all of the desired correctness criteria for using the three primitive operations: A program may not perform an A without first having done a P because there are no traces from the starting state of the prototype beginning with A; having done a P, a program may not halt without doing a V because the V-arc begins the only path back to the final state, etc.

Notice that each node of the prototype has a unit ϵ-arc indicating that other, uninteresting actions may be freely interspersed between the execution of the P, A, and V operations. All prototype graphs will have this feature by definition, since if some action were forbidden between, say, a P and a V, it would be interesting and should have a unique arc label other than ϵ to represent it.

All program verification techniques ultimately depend on a user-supplied specification of what constitutes "correctness." The prototype graph plays that role here; the methods of this chapter will aid the user in verifying that a given program adheres to the sequencing rules defined by a given prototype graph but say nothing about the correctness or appropriateness of that prototype. In Chapter IV a method will be presented which can be used to verify some properties of prototypes, but this method depends on a higher-level prototype graph which in turn must be supplied by the user as his definition of correctness.

2.4 Comparison of Graphs

Suppose we are given the state graph, G, of some program, and a prototype graph, P, which expresses the correctness properties we wish G to have. How do we verify that all of the traces in G are also in P? Is it possible to have a single decision procedure which will answer this question for any two graphs G and P?

The answer to the second question can be found in automata and formal language theory. Notice that state graphs as they have been defined here may also be thought of as finite state automata (FSA). The set of all traces through the state graph is just the regular language accepted by the FSA. The problem of determining whether or not all the traces of G are also in P is exactly equivalent to determining whether $\mathcal{L}(G) \subseteq \mathcal{L}(P)$ (where $\mathcal{L}(G)$ denotes the language accepted by the FSA G). From formal language theory we know that this question is solvable for any two FSA. (This and all following assertions about the decidability or undecidability of questions in formal language theory can be found in Hopcroft and Ullman [14], Chapter 14.)

There are many possible procedures for deciding whether or not $\mathcal{L}(G) \subseteq \mathcal{L}(P)$. One, specifically adapted to the purposes and notation of this study, is presented below followed by a proof of its correctness. It must be kept in mind that the purpose of this algorithm is not merely to decide whether or not the graph conforms to the prototype, but, in case it does not, to make it as easy as possible to find the error in the original program. This consideration precludes changing the graph G in any important way.

It is assumed that the prototype graph P is <u>deterministic</u>, by which is meant that no node has two outgoing arcs with the same label. In addition, the algorithm requires that P be <u>complete</u>, by which is meant that if anywhere in the graph there is an arc with a given label then every node must have an outgoing arc with that label. If P is not already complete it can be made so by adding to it one node called a "sink node" and adding to every node which does not have an outgoing arc with a given label an arc so labeled from it to the sink state. The nodes of P are named p_0, p, p', etc., and P has unit ϵ-arcs at every node. The nodes of G, the state graph of the program being analyzed, are named s_0, s, s', etc. G is not necessarily either deterministic or complete.

Comparison Algorithm

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

2. If there exist s, p, x, s', and p' such that (s,p) is in L, $s \xrightarrow{x} s'$ is in G, and $p \xrightarrow{x} p'$ is in P, but (s',p') is not in L, then 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.

LEMMA 1. The algorithm is finite.

Proof: All the existential searches are of finite sets. The loop in step 2 must terminate eventually, because it continues only as long as it adds new elements to L, and L is a subset of the (finite) Cartesian product of the nodes of G and the nodes of P.

LEMMA 2. At the end of step 2, (s,p) is a member of L if and only if there exists an α such that $s_0 \overset{\alpha}{\Rightarrow} s$ and $p_0 \overset{\alpha}{\Rightarrow} p$.

Proof: To show that (s,p) in L implies the existence of α, induct on the number n of iterations of step 2 before (s,p) is added to L. For $n = 0$, we have $s = s_0$ and $p = p_0$. Clearly $\alpha = \lambda$ (the empty trace) suffices, since $s_0 \xrightarrow{\lambda} s_0$ and $p_0 \xrightarrow{\lambda} p_0$. For $n > 0$, at step n-1 we had (s,p) not in L, $s' \xrightarrow{x} s$ in G, $p' \xrightarrow{x} p$ in P, and (s',p') in L. By the induction hypothesis, there exists a β such that $s_0 \overset{\beta}{\Rightarrow} s$ and $p_0 \overset{\beta}{\Rightarrow} p$ because (s',p') is in L. Now $\alpha = \beta x$.

To show that $s_0 \overset{\alpha}{\Rightarrow} s$ and $p_0 \overset{\alpha}{\Rightarrow} p$ imply that (s,p) is in L, induct on the length of α. For $\alpha = \lambda$, we have $s = s_0$ and $p = p_0$, and we know that (s_0,p_0) was put in L at step 1. If $\alpha = \beta x$, then $s_0 \overset{\beta}{\Rightarrow} s' \overset{x}{\rightarrow} s$ and $p_0 \overset{\beta}{\Rightarrow} p' \overset{x}{\rightarrow} p$. So (s',p') is in L by the induction hypothesis. By step 2, (s',p') in L, $s' \overset{x}{\rightarrow} s$ in G and $p' \overset{x}{\rightarrow} p$ in P imply that (s,p) is put in L.

<u>THEOREM</u>. The algorithm reports an error at step 3 if and only if G generates a trace not generated by P.

<u>Proof</u>: If step 3 reports an error, then there exists a (s,p) in L with s final and p not. By Lemma 2 there exists an α such that $s_0 \overset{\alpha}{\Rightarrow} s$ and $p_0 \overset{\alpha}{\Rightarrow} p$. Thus G generates α. But since P is deterministic and p is not final, P does not generate α.

If there exists an α which is generated by G but not by P, then there exists a final s such that $s_0 \overset{\alpha}{\Rightarrow} s$. Since P is complete, there exists a p such that $p_0 \overset{\alpha}{\Rightarrow} s$. Since P is complete, there exists a p such that $p_0 \overset{\alpha}{\Rightarrow} p$. By Lemma 2 (s,p) is in L. But since P does not generate α, p is not final, and step 3 will report an error. Q.E.D.

One would probably not implement the comparison algorithm as it is written. For one thing, rather than having many pairs scattered throughout L of which s is the first member, one might prefer to associate one set of nodes of P with each node of G in order to make clearer which sections of the object program correspond to which states of the prototype. The searches can be made deterministic. More importantly, it is not actually necessary to make P complete; one can modify the test at step 2 so that if (s,p) is in L and $s \overset{x}{\rightarrow} s'$ is in G but there is no outgoing arc from p labeled x, then the algorithm reports an error at once. This is because the x-arc from s to s' must be part of a trace in G which is not in P because P is deterministic. It will be assumed in the examples which follow that this second modification has been made to the algorithm.

This modification to the comparison algorithm raises a point which must be discussed in more detail. Consider the two graphs in Figure 2.2. In both graphs the node labeled 0 is the starting and only final state.

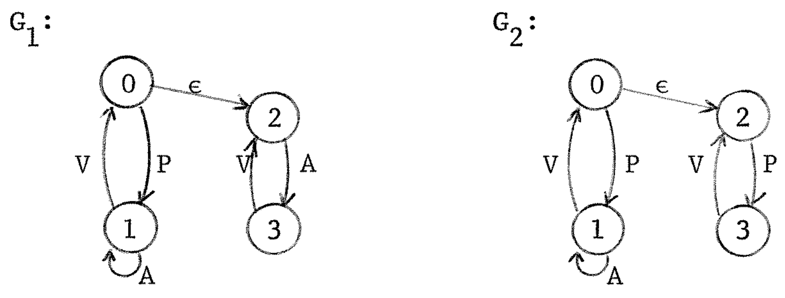

Figure 2.2. Graphs of Two Erroneous Programs

For both graphs the intended prototype is the graph P of Figure 2.1. The first graph, G_1, is clearly wrong; it contains the illegal sequences $\epsilon(AV)^*$. The original algorithm will not report these illegal sequences, however, and technically that would be correct since they do not end at a final state and hence are not elements of $\mathcal{L}(G_1)$. Obviously, however, we would like to be told about such illegal traces in programs, and modifying the algorithm so that failure is reported when arcs in G have no corresponding arcs in P accomplishes this end. The second graph, G_2, has no illegal sequences, but like G_1 it has a region from which there is no path to a final state. This presumably reflects an error in the program, but not one which the comparison algorithm will detect.

A very simple algorithm will detect them, however, First work from

s_0 forward along arcs, marking those states which can be reached from s_0; and then work from the final states backward along the arcs, marking states from which some final state is reachable. Next consider states which do not have both marks: States without the first mark are not reachable at all and may be deleted from the graph. States with the first mark but not the second have no path to a final state.

In using the comparison algorithm on the state graph G of a program, one has three choices: the above marking algorithm can be applied to G before giving it to the comparison algorithm, or one can simply assume certain properties of G without checking, or one can apply the comparison algorithm to all graphs with the understanding that technically it detects sequences rather than elements of $\mathcal{L}(G)$.

As an example of the use of the comparison algorithm, suppose we wish to verify that $\mathcal{L}(G) \subseteq \mathcal{L}(P)$ where G and P are the graphs in Figure 2.3.

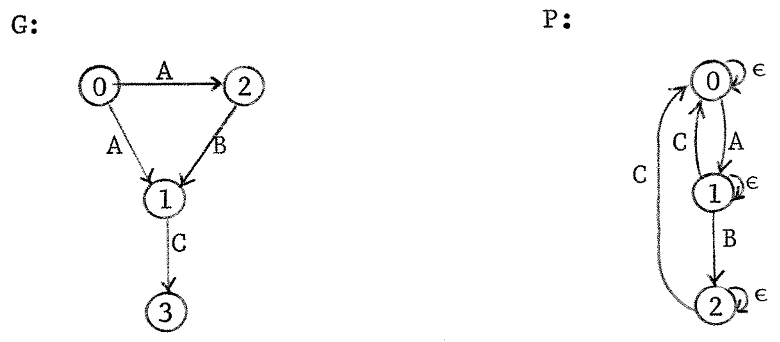

Figure 2.3. Example Graphs

Node 3 is the final state of G; 0 of P. Initially L contains only (0,0). The two A-arcs from node 0 cause (1,1) and (2,1) to be added to L. Then the C-arc from 1 to 3 adds (3,0) to L, and finally the B-arc from 2 to 1 causes (1,2) to be added. Table 2.1 summarizes the results of the algorithm so far.

TABLE 2.1

PARTIAL EXECUTION OF THE COMPARISON ALGORITHM

L	Reason for Being Added to L
(0,0)	Initialization
(1,1)	$0 \xrightarrow{A} 1$ in G, $0 \xrightarrow{A} 1$ in P
(2,1)	$0 \xrightarrow{A} 2$ in G, $0 \xrightarrow{A} 1$ in P
(3,0)	$1 \xrightarrow{C} 3$ in G, $1 \xrightarrow{C} 0$ in P
(1,2)	$2 \xrightarrow{B} 1$ in G, $1 \xrightarrow{B} 2$ in P

Nothing more can be added to L so we proceed to step 3. The only member of P paired with 3, the final state of G, is 0 which is the final state of P, so the algorithm reports success.

The comparison algorithm could be made considerably simpler by requiring that G be deterministic. In fact, there are algorithms for constructing from any nondeterministic FSA a deterministic one which accepts exactly the same language. However, these constructions may distort the original graph and destroy correspondences between states of the graph and sections of the program and thus, in the case when the algorithm reports failure, make it harder for the user to identify the faulty segment of code containing the illegal path. In all the methods of state graph manipulation and analysis presented in this study, one aim is the retention in the graph of as much information about the original object program as is practical.

Prototype state graphs allow one to state as correctness criteria any member of the whole class of regular languages. From an automata theory point of view, the method developed so far models the object program as an

FSA and then compares this FSA with another. With these observations in mind, it is natural to ask whether the method can be extended to other models and languages. Would it not be possible, for example, to model programs as pushdown automata and compare them to prototype pda, thus allowing ourselves to use as correctness criteria any context free language? The answer is no. From formal language theory we know that there can be no general algorithm for determining whether or not $L_1 \subseteq L_2$ when L_1 and L_2 are arbitrary context-free languages. The same is true when L_1 and L_2 are both any type of formal language except regular. The question is solvable when L_1 is regular and L_2 is an LR(k) language, but not when L_2 is context free or higher. This does not mean that the state graph of a program could not be used to determine if it adhered to some specific sequencing rule expressed as a higher type language by some ad hoc method but only that no general verification procedure is possible for such criteria. Thus one implication of the foregoing formal language theory results is that state graphs or some similar FSA-like model of programs are the most promising for development of general sequencing verification procedures.

2.5 State Graph Construction

Let us now turn our attention to the construction of state graphs representing programs. The general technique is to first read the source code of the object program line by line and construct an initial state graph which very much resembles the program's flow chart except that operations label arcs rather than nodes. This initial graph will then generally have to be extensively manipulated both to make it accurately reflect the program's sequencing properties and to compress it. In the discussion which follows, it will be assumed that the object program is written in an assembly language; however, the basic principles are the same for programs written in some higher-level languages.

Building the initial graph is very much like assembling the object program, with the result being a graph rather than machine code. The source code is read in one line at a time and parsed. Then, depending on the operation in the line, appropriate actions are performed on the state graph.

Some lines result in no action at all, while some classes of instructions always require actions. Instructions representing actions which appear in the prototype graph cause the creation of a new arc, appropriately labeled, from the current node to a new node representing the state of the machine resulting from that action. Any instruction which affects the flow of control of the program will result in some addition to the state graph. For example, a branch statement results in two ϵ-arcs from the current node to the two appropriate nodes. Under certain conditions these arcs may be tagged with the logical condition under which each path is taken. Any labeled instruction causes the creation of a new node corresponding to the state of the machine just before that line is executed. Program location labels may be attached to the nodes so that later the user may easily see which nodes correspond to which sections of program source code.

Techniques for building state graphs are somewhat dependent on the source language of the programs and thus may vary depending on the application, but basically are adaptations of well-known techniques in assembler and compiler writing. Chapter III contains more detail on the techniques used in one application as well as a discussion of some serious problems encountered in trying to build accurate state graphs. The present discussion will conclude with an example. Figure 2.4 shows a segment of MIX code purporting to 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.

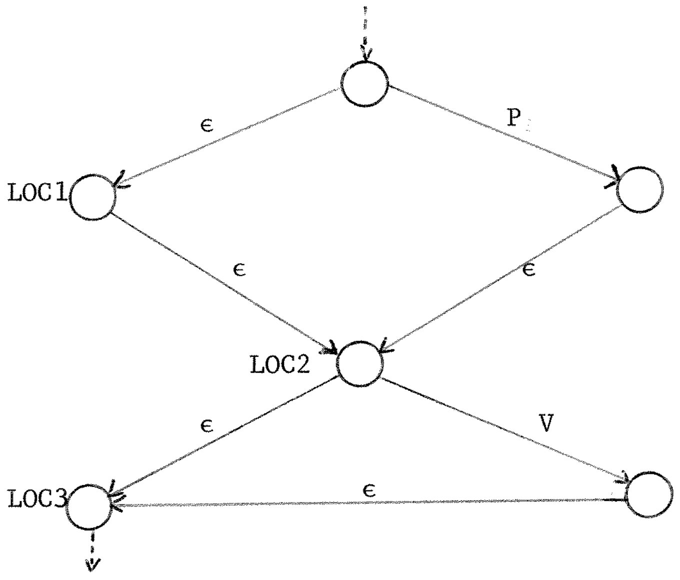

Figure 2.5. Segment of Initial State Graph (Incorrect)

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 $P\epsilon\epsilon$ and $\epsilon\epsilon V\epsilon$ 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.

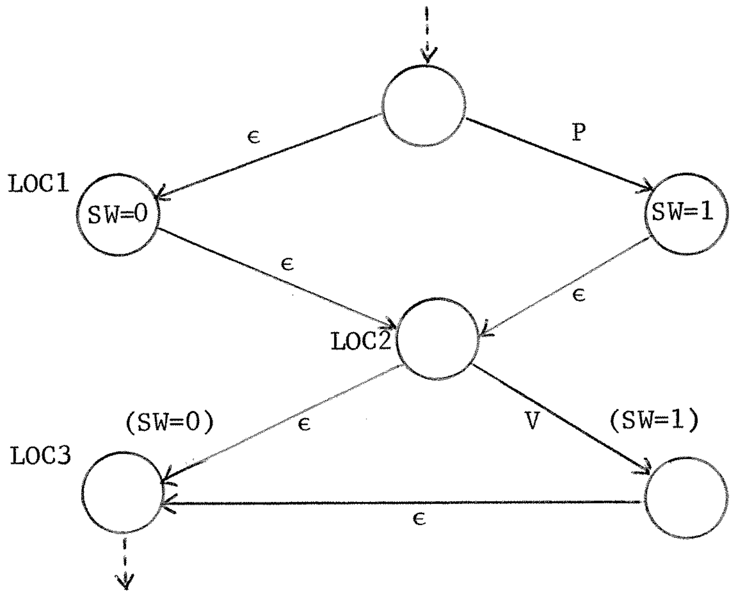

Figure 2.6. Segment of Initial State Graph (Corrected)

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 <u>splitting</u>. 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

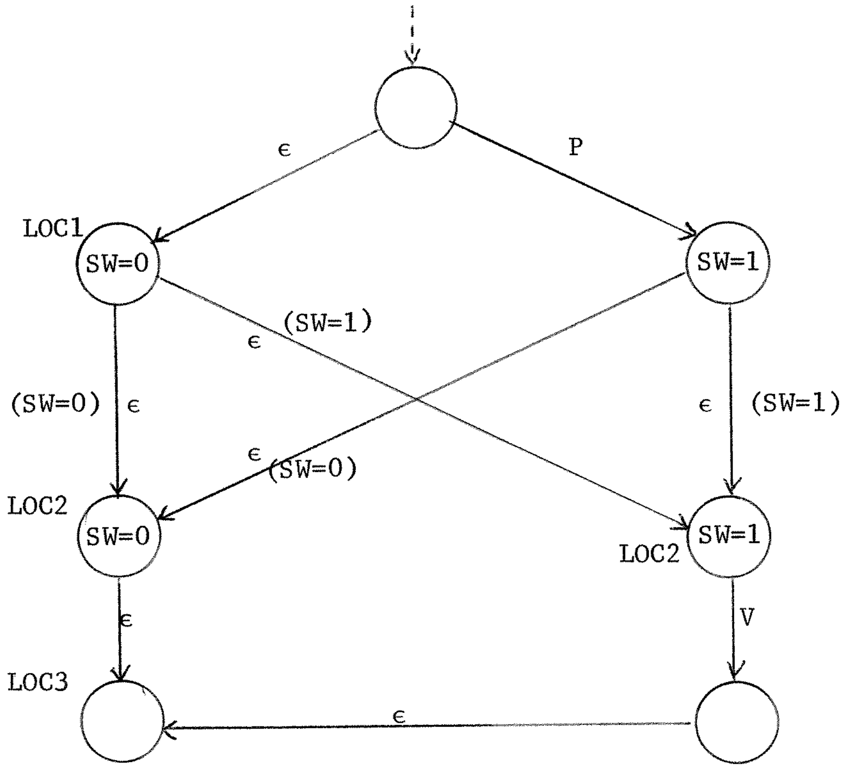

Figure 2.7. Graph Segment During Splitting

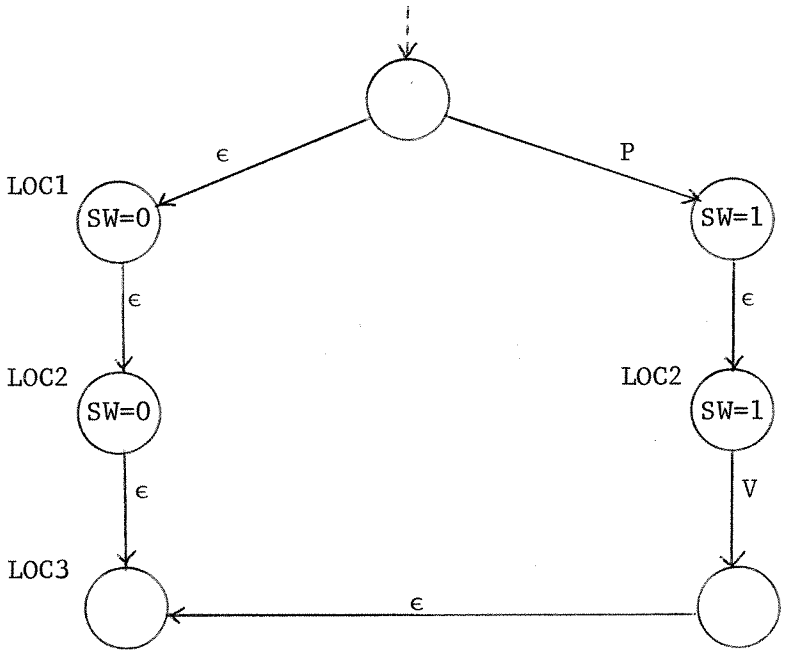

Figure 2.8. Graph Segment After Splitting

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.

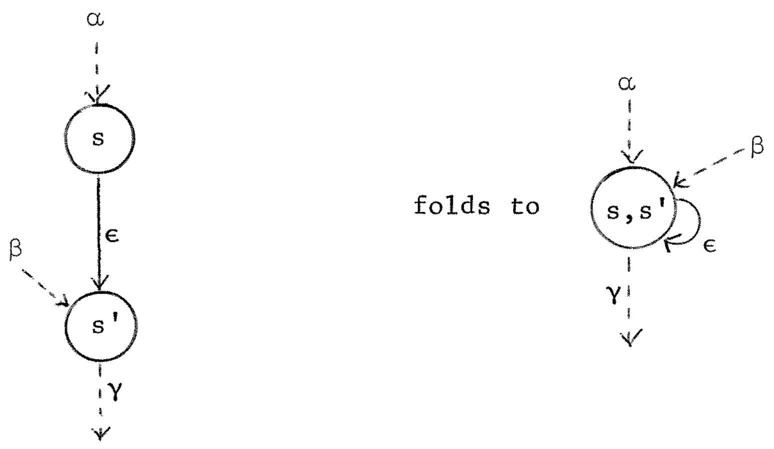

Figure 2.9. Illustration of the "Only Successor" Folding Rule

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.

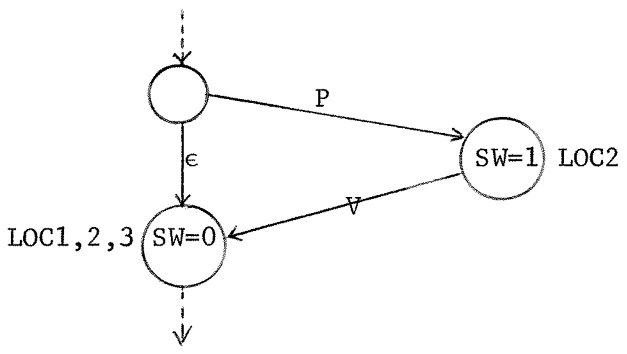

Figure 2.10. Graph Segment After Folding

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.

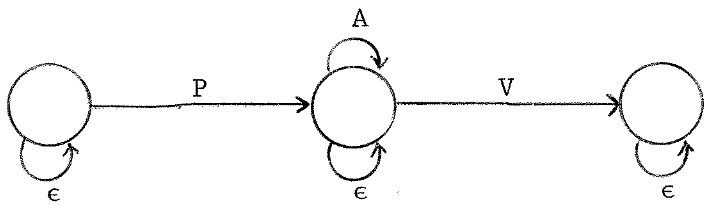

Figure 2.11. Prototype for One-use-only

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.