CORRECTNESS IN OPERATING SYSTEMS

Hugh Conrad Lauer

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Carnegie-Mellon University, Pittsburgh, Pennsylvania

September, 1972

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.

This work was supported by the Advanced Research Projects
Agency of the Office of the Secretary of Defense
(F44620-70-C-0107) and is monitored by the Air Force Office
of Scientific Research. This document has been approved
for public release and sale; its distribution is unlimited.

Acknowledgements

I am deeply indebted to Professor Alan Perlis, the source of a continuum of interesting ideas and problems, for his wisdom and inspiration of this work;

to Professors William Wulf and A.N.Habermann, my principal advisers, for their guidance through this research, particularly in its formative stages;

to Professors B. Randell, James J. Horning, and William Lynch, who through their stimulating discussions and subtle browbeating taught me to keep the faith and to keep my feet on the ground;

to Mrs. Dorothy Josephson, for her wizardry with the typewriter in producing this document;

and most of all, to my wife Ruth, who patiently typed countless notes and rough drafts, lest I not be able to read my own work, who listened attentively to and believed my wildest ideas, lest I lose my own confidence, who spent many a lonely evening silently watching me stare at those horrible formulas, never once losing hope that we could beat them into submission - but for her, I would have given up long ago.

ABSTRACT

In a method of program verification introduced by Floyd, assertions are attached to a flowchart description of a program, and correctness is established by showing their consistency with respect to that flowchart. In this thesis, the method has been extended to apply to concurrently executing programs such as those which occur in operating systems. This has required a careful definition of process and an effective representation of the interactions among processes. For this purpose, a flowchart-like representation was chosen to characterize all possible computations resulting from the asynchronous execution of two programs. Floyd's results were then applied to this representation to transform a problem of verifying a set of programs into a problem of proving a theorem of logic. Simplifications suggested by the structure of the programs are applied to reduce the level of difficulty. Transformations suggested by the interactions of the programs are applied to facilitate the effective characterization of properties of interest.

The verification methods are applied to several small examples of systems of cooperating processes. They illustrate that interesting properties can be proved about such systems but that there are still many unsolved problems. In particular, a precise formulation of the concept of an abstraction of a process is required. The thesis concludes with a presentation of a possible formulation of such a concept and a brief exploration of its properties.

TABLE OF CONTENTS

INTRODUCTION

This thesis considers ways of establishing the correctness of properties of cooperating processes. It extends the work of Floyd [1967], Manna [1968], King [1969], and others to include programs which execute "concurrently". It is complementary to the theoretical work of Dijkstra [1965a], Habermann [1969], Holt [1971], and others in considering the accuracy of programs which occur in operating systems. Hopefully, it will provide some insight and will lead to techniques to help solve the problem of constructing large, complex operating systems which work properly.

The difficulty of programming and debugging systems in which several coordinated activities proceed concurrently has been documented by Dijkstra [1965a], Van Horn [1966], Wirth [1969], and others, and it has become painfully apparent to systems programmers in recent years. Such systems are generally not deterministic, in the sense that from a given starting point, one could reach any of a number of different states, depending upon the relative speeds of the component activities. Errors are often difficult to trace because they are caused by events of which all record has vanished, and they are more difficult to reproduce because they depend on rare (or unknown) combinations of circumstances which cannot be readily duplicated.

There are two useful, complementary approaches to the problem. The synthetic approach concentrates on developing programming techniques which avoid the pitfalls of parallel activities and which ease the burden created by problems of scale in system design. The analytic approach

concentrates on examining systems to identify the potential trouble spots and/or to show that they do not exist. The latter category includes various techniques, one class of which involves attempting to verify programs with respect to assertions describing the properties in question. In this thesis, we will concentrate on proving the correctness of concurrently executing programs by showing the consistency of such assertions. With the exception of some work reported by Ashcroft and Manna [1971], none of the existing assertion-oriented techniques applies directly to systems of concurrent processes. But we shall see that by adopting a suitable representation, these methods can be extended to apply.

Whether or not it is practical and/or useful to prove a program correct, particularly in formal terms, is the subject of many heated debates. Furthermore, it is unlikely that anyone will, in the near future, be able to prove that a large operating system is correct - simply because it is too difficult. Even so, there are many reasons why we should develop the formal apparatus for doing so. First, proving correctness is an interesting mathematical problem in its own right. It is nice to know under what circumstances it can be done and how to do it. It is also nice to know what constraints the peculiarities of concurrent execution present to the problem.

Another, perhaps more compelling, reason for developing verification techniques is that they may lead to useful synthetic techniques for building systems which are a priori correct and which display a better organization than our present ones. This is already happening with sequential programs. Naur [1969] and Hoare [1971] have combined

a constructive approach to program design with their methods for formally proving correctness. Both have demonstrated the development of simple sequential programs using these techniques. Similarly, Henderson and Snowden [1971] have shown that program verification methods are an important conceptual tool to use with the structured programming techniques of Dijkstra [1970] - not so much for proving a program correct as for helping the programmer to maintain suitable relationships between its parts.

Formal verification methods are also useful in analyzing abstract models of various system functions. For example, Ladner [1970] has used the results of Floyd [1967] to establish the theoretical basis of his analysis techniques for communication algorithms, even though those techniques themselves bear little relation to formal logic and theorem proving. We might reasonably expect that an understanding of how to verify the programs of an operating system would be useful in constructing analytic methods to apply to such systems. Similarly, we can hope that understanding the formal techniques would lead to better understanding of the systems themselves and of the laws which govern their structure and behavior.

Finally, we will show in this thesis that our verification techniques can be applied to some very small programs. We may, in the course of future research on system structure, discover ways to reduce the dependence of system reliability on correct code. If that is done, then it will be sufficient to verify only small, critical programs to establish some confidence in a system with a suitable structure. I.e., while it would be

too difficult to consider verifying a whole system, we may obtain useful information by verifying small pieces of it.

The kinds of programs which occcur in operating systems are often quite different from those we write for many other applications. The programs, which comprise the central "monitor" and the various "system processes", usually share some of their variables with others, and the values of these variables are liable to be changed by any of the programs at almost any time. Thus they are non-deterministic - that is, their execution is not predetermined by any observable state. These programs rarely halt and are usually designed not to. Instead, they loop, continually looking for more work to do and regularly transferring important status information from one iteration to another. Thus, it is pointless to try to prove them correct with respect to the traditional criteria that they halt and that the output be a given function of the input. We must develop more appropriate criteria.

Programs in operating systems also suffer from errors which are not considered by existing verification methods. For example, they may become "blocked" and prevented from further execution. A whole system becoming blocked represents a sort of catastrophic error which we would prefer not to happen. Another example involves the problem of protecting information. Design errors which allow faulty or malicious programs to access variables which they should not must be discovered and eliminated. We need appropriate criteria of correctness to cover this kind of problem, as well.

We do not claim to have developed a complete set of criteria and formal verification methods appropriate for operating systems. But the

results of this thesis are a beginning. We will define correctness in terms of predicates which assert some relationship among the values of the variables of the programs. Because we are considering programs which execute concurrently and which exchange information, we must pay careful attention to their representation and the representation of their interactions. To this end, we will develop a flowchart-like notation which will become the basis for our verification techniques. Then, as an extension of Floyd's Induction Theorem [1967a], we will show that a problem of verifying a system of cooperating processes can be reduced to proving a theorem of the first-order logic in a particular interpretation. Methods for proving several other forms of correctness are derived from this by suitable transformations to the problems or the programs.

In the course of this research, it became apparent that apart from the tedium of proving theorems of logic, one of the greatest difficulties in verifying a system of programs is finding an appropriate statement of correctness which can be proved. In the case of several simple systems, the "obvious" statements about the programs do not contain enough information to allow a proof and the author was not able to find less obvious, but provable, statements. This demanded new techniques which allow us to infer the correctness of a system from some additional information. To do this, a second system is constructed from the first by adding redundant information or identifying certain states and changing the representation slightly so that a statement of correctness is more convenient. Then we can prove that the correctness of the second system implies that of the first.

Finally, we will see that the methods developed in the thesis are not restricted to machine level programs but can be applied to more abstract representations, as well. A verification of a system can then be factored into two parts: a verification of the abstract representation and a demonstration that the representation itself accurately reflects the system. This may be easier than considering the system as a whole, and it is a natural consequence of systems designed in "levels of abstraction", such as the THE System (Dijkstra [1968b]).

It is worth pointing out that the author's interest in this research has been directed toward the structure and correctness of computer operating systems. As such, the theorems and examples are biased in that direction. But neither the results of this thesis nor the assumptions from which they are derived are peculiar to operating systems problems and, in fact, can be extended to any system of cooperating processes at any level of abstraction. The reader is invited to generalize whenever and wherever he wishes.

Outline of the Thesis

The following material is divided into five chapters. In the first chapter we will develop a formal model of computation, inspired by the informal point of view of Dijkstra [1965a] and the semi-formal model of Horning and Randell [1969]. This model is oriented toward the problem of representing combinations of processes, and it forms the basis of our verification techniques. It is also useful in considering the more general problem of forming "levels of abstraction" or abstract representations of systems of processes, as will be seen in Chapter V.

In Chapter II we adapt Manna's graphical representation of processes to account for concurrent computation. These graphs, which we designate as transition graphs, are similar to flowcharts, but with the roles of the arcs and nodes reversed. That is, in a conventional flowchart the nodes (or boxes) represent actions and the arcs between nodes correspond to the states between actions, whereas in the transition graph, the arcs represent the actions and the nodes correspond to the states between actions. Several simple theorems will be proved to show that a transition graph formed from those of a system of concurrent processes represents the combined effect of their simultaneous operation.

In Chapter III we present a definition of correctness and extend Floyd's Induction Theorem to apply to cooperating processes. This is done by generating a formula of logic from the transition graph representing the system of processes. Then the system can be proved to be correct if those predicates satisfy the formula - i.e., if they cause it to be true, when substituted for the symbols of the formula in an orderly way. Finally in this chapter we will present some techniques for reducing the complexity of a verification based on the structure of the system.

In Chapter IV we consider three additional ways of stating and proving correctness. In the first, the transition graph representing a process or system is extended with redundant information to allow a convenient way of making assertions about the states of the system which cannot be made only in terms of its variables. These simplify proofs of correctness considerably. In the second, we consider ways of stating and proving that something does not happen in a system - i.e., that certain combinations of states never occur. This method is useful provided that

those combinations can be identified. If not, the third method may help. It allows us to prove that something does not happen (for example, a deadlock) by proving instead that something else always happens (for example, the processes always reach a home state in finite time). The last two techniques are based on the termination results of Manna [1968] for non-deterministic processes.

In the final chapter we investigate the question of what circumstances permit the correctness of a system to be established from an "abstraction" rather than from the system itself. We will consider the conditions under which it is possible to consider the cooperation of abstract processes without reference to their underlying implementation. In particular, the methods of previous chapters suggest, in principle, a method of verifying that an alleged abstraction of a process correctly represents that process.

CHAPTER I

INFORMATION, PROCESSES, AND OPERATING SYSTEMS

To prove the correctness of a set of cooperating processes, we must know precisely what is meant by the term process and by the idea of cooperation or communication among several of them. Horning and Randell [1969] have given a survey of some definitions of these terms and have extracted the essential elements into a model of the computation done by cooperating processes. In this chapter we extend that model to overcome their difficulty in characterizing "asynchronous" cooperation.

This is done by incorporating Dijkstra's important, abstract assumption that all actions in a system are timeless and no two actions occur at exactly the same instant. Thus in our model, there is never any conflict among processors about assigning a value to a variable. This abstraction bears no more relation to reality in computing than does the postulation of frictionless surfaces in physics. But it is useful for solving a large class of problems, including those of this thesis. It is, of course, possible to simulate such a world, and many hardware and software systems do to some extent. As a result we can factor a proof of correctness into parts: (1) a verification of the abstract system (using the assumption) and (2) a proof that the real system being verified correctly simulates this timeless environment. In this thesis we will concentrate on the first part, but we will comment on the second part in Chapter V.

With the current proliferation of models of computing, one may

question the value of introducing yet another. Our model will be justified if it persuades the reader that our results on proving correctness are relevant to programs which occur in operating systems. It may also be useful in solving problems beyond or orthogonal to the scope of this thesis; if that happens, it will be additional justification. On the other hand, when the results obtained with it can be understood more simply without it, then it will be time to relegate it to the role of an historical curiosity.

We will begin with the primitive, intuitive concepts of information and processor which form the basis of our programming experience. From this, we will define the union and intersection of information sets and the product of two processors. The terms computation, process, and combination of processes are based on these definitions. An essential consequence of the definition is that a process may be formed from the union of two information sets and the product of two processors, and this process represents the behavior of two processes interacting with each other. Thus all of the results about processes apply to combinations of them as well. Finally, in this chapter, we will characterize operating systems from the point of view of the model and of establishing the correctness of their components.

Information Sets

The basic stuff in computing is information. It occurs in sets of information elements, and each element is an object which has a distinct identity or name and which can assume any of a domain of values or states (these last two terms are synonymous for our purposes). Examples of

information include a bit, which can have either of two values, a 32-bit word, which can have any of 2^{32} distinct values, or an Algol integer variable, which (in the most abstract sense) can have any of the integers as value. In the discussions which follow, we will not be concerned with any internal structure of information elements. Instead, they will be regarded as primitive, indivisible objects defined only by our programming intuition.

An information set is a named set of information elements, and its value or state is the set of values of its elements. Examples of information sets include the set of variables of a program, a collection of files in an operating system, and programs or systems of programs themselves. Another, more abstract, example is the potentially infinite set of incarnations of the variables of a recursive Algol program. While the structure of an information set is of paramount importance in program design, it will not be of concern to this discussion. We are only interested in the composition of information sets - i.e., which elements are contained in it - and its value, regarded as a whole.

In practice, we deal only with finite sets of information elements, each of which can have only a finite number of values. However, it is often convenient to imagine infinite information sets for purposes of simplifying analysis. For example, we often assume that we can do arithmetic on the set of all integers, even though our computers use one or another system of finite arithmetic (see Hoare [1969]). Our model will accept axioms for either system and is, in general, not restricted by any finiteness assumptions. We will, however, take advantage of finiteness where it naturally occurs.

In order to consider communication and cooperation among processes, we will need to form combinations of information sets. The following notation will be useful:

Let the upper case letter X, Y, Z, X', Y', etc., denote information sets, and let corresponding lower case letters (possibly subscripted) denote their values. I.e.,

$$x, x_1, x_2 \ldots$$

and

$$y', y'_1, y'_2, \ldots$$

will denote values of the information sets X and Y', respectively. Because information sets are sets of information elements, it makes sense to consider the set-theoretic operations $\cup$, $\cap$, and $\setminus$ (i.e., union, intersection, and set difference) on them. Then, if X and Y are two information sets, then we will use the notation $x \cup y$, $x \cap y$, and $x \setminus y$ to denote the values of $X \cup Y$, $X \cap Y$, and $X \setminus Y$, respectively, where x is the value of X and y is the value of Y. That is, the symbols "$\cup$", "$\cap$", and "$\setminus$" do not define operations on the values of information sets. Instead, they are a convenient way of representing the values of the corresponding combinations of two information sets in terms of the values of their components. Finally, if the information set X is a subset of Y, we will use notation $y|X$ (read "y restricted to X") to denote the value of X in terms of the value of Y.

Processors

Information is useful in computing because its value can be changed

in well-defined, mechanical ways. The act of computing is the act of repeatedly transforming the value of an information set with the aim of producing some value with a specific meaning in the context of the problem being solved. The instruments of computing are processors - i.e., real or conceptual devices which implement sets of rules specifying how the values of information sets are to be changed or transformed. A processor is generally designed or defined to manipulate a particular information set (class of information sets), which we will call its operand. Each of the possible changes in value that the processor can make upon an operand is called an operation.

A processor may be a hardware automation, a program, or some conceptual entity. For example, the Central Processing Unit of a computer system is a processor which operates on an information set consisting of core memory, accumulators, and whatever other registers it may have. The rules which it implements are those described informally in its programming manual. Alternatively, a processor could be a program written in some language to specify how variable data is to be transformed in value. The operand in this case is the set of variables of the program. Interpreters of languages such as LISP fall into this category: in effect, they create imaginary processors to implement the rules of transforming information as defined by the semantics of the language. The art of programming is, then, the act of defining one or more processors and information sets, usually in terms of other processors and information sets.

This idea of processor is, of course, intuitively obvious and fundamental to competent programmers. It will form the basis for

precise definitions of computation and process and it will be convenient for describing the communication among processes. Before we proceed with these definitions, it is useful to introduce some additional notation.

Suppose Q is a processor and the information set Y is its operand. If Q defines an operation on the value of Y from y_0 to y_1, then we say

$$y_0 \xrightarrow{Q} y_1,$$

i.e., that y_1 is a successor of y_0 under Q. It may be the case that some values y_0 of Y, have no successor values. These are called blocking states of Q, and Q is said to be blocked when Y has one of these as its value.

Example:

A processor Q is defined such that one of its operations is the P-operation (Dijkstra [1965]) on a particular variable of type semaphore. The P-operation reduces the value of the semaphore by one provided the result will be non-negative; it has no effect on any other variable in the information set. The operation has no successor when the semaphore has values less than one. I.e., if s is a semaphore, the operation

$$s = 3 \xrightarrow{Q} s = 2$$

is possible, but the state

$$s = 0$$

is a blocking state of Q. I.e., Q cannot continue to operate until the value of s is changed by some other means.

In this thesis, we will ignore all issues surrounding the decidability or undecidability of whether or not a particular operation $y_0 \xrightarrow{Q} y_1$ can occur, and we will restrict our attention to processors such that this can be determined for all values y_0 and y_1.

It may be the case that some value of y_0 of an information set Y has more than one successor under processor Q - i.e., that

$$y_0 \xrightarrow{Q} y_1$$
$$y_0 \xrightarrow{Q} y_2$$
$$\vdots$$
$$y_0 \xrightarrow{Q} y_k$$

and $y_1 \neq y_2 \neq \ldots \neq y_k$. I.e., Y does not contain enough information to allow us to determine which operation Q will make next. It may be chosen by a random process or it may depend on factors external to our domain of discourse. In this case, P is called non-deterministic: otherwise it is deterministic. (Note that this use of the term "non-deterministic" is related to, but not the same as, its use by Floyd [1967b]. Ashcroft and Manna [1971] also use the term with slightly different implications.) Although we like to think of the hardware processors which we build as deterministic, we combine them in system with the result, which we will see presently, that abstract processors of operating systems are frequently non-deterministic.

Processors with Multiple States

It will be convenient to consider processors with more than one internal "state". For example, a processor may be defined by an Algol program and its operand may be the variables of that program. Clearly, the operations defined by this processor depend upon which statement is currently being executed. This is not represented by any part of the operand, but by information strictly internal to the program. I.e., for a given value of the operand, the possible successors are determined by the current internal state of the processor. In general, if a processor P in state v_0 can perform the operation $y_0 \longrightarrow y_1$ and find itself in state v_1 as a result, we say

$$(y_0,v_0) \xrightarrow{P} (y_1,v_1),$$

i.e., that (y_1,v_1) is a successor of (y_0,v_0) under the processor P. The set of internal states of P may represent values of internal registers or data structures, the position of a reel of tape, a program counter, or some other characteristic.

From a theoretical point of view, there is little to distinguish single state processors from multiple state processors, and either kind can be mapped directly to the other kind. I.e., the operand of a multiple-state processor can be extended with an information element representing the states. Then a new, single-state processor can be defined with operations similar to those of the given one and dependent in the obvious way on the new information element. Conversely, a processor with only one or a few states can be transformed into one with many states by moving information from its operand to its internal structure. For convenience,

we will usually assume that our processors can have several states. In Chapter II a method of representing a class of these will be presented.

Computation and Processes

When a processor is repeatedly applied to an information set, it defines a sequence of values of that set which is the essence of the computing task for which the two were intended. The sequence may be finite or infinite, and it is called a computation.

Definition: Let P be a processor, and let the information set Y be an operand of P, let y_0 be a value Y, and let v_0 be an internal state of P. A computation of P on Y given (y_0,v_0) is a sequence of pairs,

$$\{(y_0,v_0),(y_1,v_1),(y_2,v_2),\ldots\}$$

either finite or infinite, where for each $i = 0,1,2,\ldots,y_i$ is a value of Y, v_i is an internal state of P, and

$$(y_i,v_i) \xrightarrow{P} (y_{i+1},v_{i+1})$$

The state (y_0,v_0) is called the initial state of the computation.

Note that if P is non-deterministic, there may be more than one computation for some initial states. Finally note that Horning and Randell [1969] use the term "history" to denote what we have called "computation".

Given an information set Y, a set Σ of initial values of Y, and an effective description of a processor P which operates on Y, it is possible to construct all of the computations of P on Y given any of the initial

values in Σ. This set of computations, called the history set by Horning and Randell, represents all that P can do to Y. All of the values of Y which might conceivably occur, solely as the result of the action of P, are members of the computations in the set, and the set itself is completely determined by P, Y, and Σ. This motivates the following definition.

Definition: A process is a triple (P,Y,Σ), where

P is a processor,

Y is an operand of P, and

Σ is a (non-null) set of pairs (y,v) consisting of values of Y and internal states of P.

We will frequently represent processes by lower case letters

$$\underline{p},\ \underline{q},\ \underline{r},\ \underline{p}',\ \underline{q}',\ \underline{r}',\ \text{etc.}$$

If $\underline{p} = (P,Y,\Sigma)$ is a process and if (y_0,v_0) is in Σ, then a computation of P on Y given (y_0,v_0) is called a computation of $\underline{p}$.

The term "process" is common in operating system design and has been defined in many different, but related, ways. Saltzer's emphasis [1966] is on the activity of the processor and on which information set is involved. Lampson [1968] emphasizes that it is distinct processors which make distinct processes. Dijkstra [1965a] identifies a process as the act of a processor operating on an information set to produce a specific computation. Horning and Randell [1969] have surveyed other definitions. All of them specify, in one way or another, that the principal ingredients of a process are its own processor, an information set, and a starting

state. Brinch Hansen [1970] has emphasized that processes are not restricted to those having processors resembling the central processing units of conventional computers. The asynchronous input/output activities, the ticking of system timers, and the setting of external switches all are types of processes, and all fall within the definition given above.

Timelessness of Processor Operations

Our definitions of computation and process make no mention of time, but only of ordered sequences of states. Yet they represent activities which occur in time and which take non-infinitesimal time to happen. This important aspect of our model of computation was introduced by Dijkstra [1965a]. It simplifies considerably the analysis of cooperating processes to be considered below; but at the same time, it requires that the processes under scrutiny have a certain kind of structuring about them.

Essentially, what we do is assume that the operations of processors occur in zero time, and that processors and information sets spend all of their time between operations. In effect, time acts as a discrete counter, not as continuous measure (Habermann [1967]). We further assume that no two processors perform operations on information at precisely the same instant. I.e., whenever two processors both perform operations, we assume that one performs its operation of infinitesimal duration first, then the other acts. The order might not be predictable. (In the case of processors specifically designed to work in "lockstep", they can be combined into a single processor by the methods

of synchronous combination discussed by Horning and Randell [1969].) A consequence of this abstraction is that the value of an information set at the instant after an operation is determined only by the processor which performs it, not on any possible interference by other processors.

No piece of physical machinery actually performs operations in zero time, but most of the processors and storage media that we build today act as if this were true. That is, they are designed with interlocks and interfaces which guarantee that the sequences of observable information states are exactly those which would be observed if a timeless processor were used instead.

Example: The core memories of most modern computer systems are designed so that a CPU and an I/O channel cannot both access a given word simultaneously. One must wait until the other has completed its operation, then it operates on the new value of memory.

It is a non-trivial problem to construct processors which obey this principle and an even harder problem to prove that they do obey it. The operating system programmer can, with some restrictions, assume that his hardware obeys it; but he must write programs which are, in effect, abstract processors which also obey the principle. This has been the subject of considerable discussion in the literature, and it has given rise to the various so-called "synchronizing primitives" which are now common programming practice (see, for example, the P- and V-operations of Dijkstra [1965], the message handling functions of Brinch Hansen [1970], the functions in MULTICS as described by Saltzer [1966]). Thus, an

important verification problem is that of proving that the abstract processors simulated by a set of cooperating processes obey the principle of timelessness.

Cooperation Among Processes

Let $\underline{p} = (P,Y,\Sigma)$ and $\underline{q} = (Q,Z,T)$ be processes, as defined previously. Let us consider what happens if $Y \cap Z$ is not null. That is, we will allow the processor P to access some information elements in Z and Q to access some elements in Y. Clearly, the sequences of states of Y and Z realizable by this case are not necessarily computations of $\underline{p}$ and $\underline{q}$, but may be the result of interactions between the two processors. For example, P may change the value of an information element in $Y \cap Z$ upon which the next operation of Q depends; such a variable is called a significant variable of Q by Horning and Randell [1969].

In order to see what sequences of states of $Y \cup Z$ are possible, we will construct a processor R from the definitions of P and Q. For simplicity, let us first assume that P and Q have only one internal state each. Then we define the single state processor R as follows:

whenever $y_a \cup z_a$ and $y_b \cup z_b$ are values of $Y \cup Z$,

then

$$y_a \cup z_a \xrightarrow{R} y_b \cup z_b$$

if and only if either

$$y_a \xrightarrow{P} y_b \text{ and } z_a \setminus y_a = z_b \setminus y_b$$

or

$$z_a \xrightarrow{Q} z_b \text{ and } y_a \setminus z_a = y_b \setminus z_b$$

I.e., an operation of R on $Y \cup Z$ is either an operation of P on Y with that part of Z which does not intersect with Y ($Z \setminus Y$) left unchanged, or it is an operation of Q on Z with $Y \setminus Z$ left unchanged.

In general, R is not deterministic because there can be values of $Y \cup Z$ which are not blocking states for either P or Q. For these states, the choice of which operation R performs next depends upon the relative speeds of P and Q, the physical implementation of the processors and information sets, and/or other factors which may be random or imperceivable.

In order to define a process based on R and $Y \cup Z$, we must also define a set of initial states of this information set. In particular, let υ be the set of values

$$\{y \cup z \mid y \in \Sigma \text{ and } z \in T\}.$$

I.e., it is the set of values of $Y \cup Z$ such that the Y-part is an initial state of $\underline{p}$ and the Z-part is an initial state of $\underline{q}$. In symbols, we say $\upsilon = \Sigma * T$. Then, by construction, the computations of the process

$$\underline{r} = (R, Y \cup Z, \upsilon)$$

are exactly the sequences of values of $Y \cup Z$ which result from an arbitrary interleaving of the operations of P and Q, subject to the assumption of timelessness. We called the processor R the <u>product</u> of P and Q, and similarly, we call $\underline{r}$ the <u>product</u> of $\underline{p}$ and $\underline{q}$; in symbols, we write

$$R = P \times Q$$

and

$$\underline{r} = \underline{p} \times \underline{q}.$$

Example: The OS/360 operating system for the IBM System/360 [1970b] defines one or more abstract processes (called tasks in the terminology of that system) which are, in effect, product processes. Each task includes one "computing" process, zero or more "input-output" processes, and zero or more "timer" processes. The computing process is the only one that is programmable in the conventional sense, the others being table-driven. All of the processors are simulated by a combination of the hardware and system software. They communicate using operations on information contained in shared tables - for example, the computing processor performs "GET" and "PUT" operations which manipulate and test specific variables, while the input-output processors keep buffers full (or empty) and record their status in those same variables. The system programmer generally regards a whole task as a programmable process, but he must be aware of the non-determinism of the synchronization among the component processes. I.e., a task is a product of simpler processes.

In the case of processors with more than one internal state we can formulate the notion of product precisely as follows:

Definition: If P is a processor of the information set Y and Q is a processor of the information set Z, then the product, R, of P and Q is a processor of the information set $Y \cup Z$ such that

(a) the set of internal states of R is the set of pairs $\{(v,w)\}$ where v is an internal state of

P and w is an internal state of Q, and

(b) for values $y_a \cup z_a$ and $y_b \cup z_b$ of $Y \cup Z$ and for states (v_a,w_a) and (v_b,w_b) of R,

$$(y_a \cup z_a,(v_a,v_w)) \xrightarrow{R} (y_b \cup z_b,(v_b,w_b))$$

if and only if <u>either</u>

$$(y_a,v_a) \xrightarrow{P} (y_b,v_b) \text{ and } z_a \setminus y_a = z_b \setminus y_b \text{ and } w_a = w_b$$

<u>or</u>

$$(z_a,w_a) \xrightarrow{Q} (z_b,w_b) \text{ and } y_a \setminus z_a = y_b \setminus z_b \text{ and } v_a = v_b.$$

The <u>product</u>, <u>r</u>, of two processes $\underline{p} = (P,Y,\Sigma)$ and $\underline{q} = (Q,Z,T)$ is the process $(R,Y \cup Z,\upsilon)$, where $\upsilon = \Sigma * T$. In symbols, we write

$$R = P \times Q$$

and

$$\underline{r} = \underline{p} \times \underline{q}.$$

Informally, we say that the processes <u>p</u> and <u>q</u> <u>cooperate</u> and <u>communicate</u> through their common information set $Y \cap Z$.

Clearly it follows from that definition and the commutativity of set union that the processors $P \times Q$ and $Q \times P$ define exactly the same set of operations on $Y \cup Z$, and so can be considered the same processor. It is also apparent from the construction that the processors

$$P \times (Q \times R)$$

and

$$(P \times Q) \times R$$

are identical.

Note that according to the definition, a processor cannot force a change in internal state upon another. There is no practical loss of generality here because the representation can be changed so that the internal states are described by information elements common to the operands of both processors.

Finally observe that since by our definition the combination of a set of processes is also a process, we could use the term "process" to mean a single, deterministic process or the combination of a set of processes. However, it will sometimes be advantageous to talk about a "system of processes" in order to make it clear that we are interested in properties relating to the combination of the processes and not specific to any individual one.

A Characterization of Operating Systems

Randell [1971], paraphrasing Barron [1969], has observed that while it is difficult to define precisely what we mean by the term "operating system", any experienced programmer would recognize one when he sees it. Nevertheless, we can characterize operating systems with respect to certain properties relevant to specific problems. For example, a person interested in performance might characterize all operating systems as resource allocation mechanisms, while someone else might characterize them as interpreters of command languages. From the point of view of an abstract study of correctness, we can regard an operating system as a collection of processes which cooperate to simulate one or more abstract processors and information sets. I.e., using a common but imprecise term, we regard an operating system as a collection of processes

which create one or more "virtual machines". The programs of an operating system collectively act as interpreters of the abstract processes so created. The question is then, "Do the processes correctly simulate what they are designed to simulate?".

There are several important characteristics of these programs which affect the problem of proving them correct. One is that many of them are not meant to terminate, so that verification techniques based on termination do not apply. I.e., these programs, particularly those that manage the processors and principal memory resources, loop and repeatedly look for work to do. They also transmit information from one iteration to another, so that there is no natural terminating point.

Another characteristic of programs in operating systems is that they define, communicate with and synchronize with other processes. I.e., they transmit information to others via shared data, and they await answers. They become blocked by menas of synchronizing operations such as the P- and V-operations or their equivalent. Thus there are the natural questions of whether or not they communicate and synchronize correctly, and whether they will be restarted correctly when they are blocked pending some event.

A third characteristic is that, despite the best intentions of operating systems programmers, the processes are at times non-deterministic. Processes are combined with others to form product processes which are non-deterministic, at least to the extent of the relative speeds of the components. This complicates the verification problem because the values of shared variables are not necessarily safe - i.e.,

one process may set a value but another may change it at any time.

These considerations will motivate the verification methods introduced in later chapters. It will be apparent that they also apply to other systems of cooperating processes which are not operating systems. However, we shall restrict our attention and examples to problems drawn from the operating system area.

CHAPTER II

TRANSITION GRAPHS

In later chapters, a principal method of proving the correctness of a process or product of processes will be to prove that a statement is true for each of its computations. This requires a representation of processes and processors which facilitates systematic proofs about all computations, possibly infinitely many of them. A suitable representation for this purpose is the directed graph used by Manna [1968] which, for lack of a better term, we will call the transition graph. It is analogous to a flowchart, but the roles of nodes and arcs are interchanged, the arcs representing actions and nodes representing the intervals between actions.

In this chapter, the idea of the transition graph will be illustrated by a brief example and then defined precisely. We will show that it represents all of the computations of the corresponding process. We will also present a natural way of combining transition graphs of several processes which results in a transition graph representing the product of those processes. This is convenient for considering the computations which result from the cooperation of several processors operating on a common information set. Finally, we will comment on some useful properties of transition graphs.

Transition Graph of a Process

Let us consider a process designed to multiplex and transmit messages of N "logical" communication channels over one physical communication channel. This process will be part of a system for reliable and

efficient communication, proposed by Lynch [1968 and 1971] and to be discussed in Chapter V. The processor and its information set can be described by the pseudo-Algol program of Figure II.1. The process continually loops, transmitting the messages contained in its buffer named "Message". If "OK" indicates that a message has been correctly received, a new message is fetched from the corresponding logical channel and the value of "Alternate" is reversed. The physical message actually transmitted consists of the identification "k", the k^{th} message, and the values of the k^{th} Verify and Alternate bits.

In Figure II.2, a flowchart of the program is shown. It is reasonable to expect that each solid box in the flowchart could correspond to one, indivisible, timeless operation by a suitable processor. The dotted boxes, however, represent operations which would probably be implemented as several operations spanning a non-trivial amount of time. But for purposes of illustration, we will imagine them to be timeless operations, as well.

```
begin string array Message (0:N-1);
     Boolean array OK (0:N-1) = true;
     Boolean array Verify, Alternate (0:N-1) = false;
     semaphore sync=N; integer k;
     comment: The string array "Message" contains the
          current messages being transmitted on the
          logical channels. "OK" is set by another pro-
          cess to indicate that messages have been re-
          ceived correctly. "Verify" and "Alternate"
          will be explained later. The semaphore "sync"
          is used for synchronizing this process with
          another one;
     for k := 0, (if k ≥ N-1 then 0 else k+1) while true do
          begin P(sync);
          if OK (k) then begin
               Message (k) := <next message on channel k>;
               Alternate (k) := ¬ Alternate (k) end;
          transmit (k, Message (k), Verify (k),
                    Alternate (k)); end;
end;
```

Figure II.1

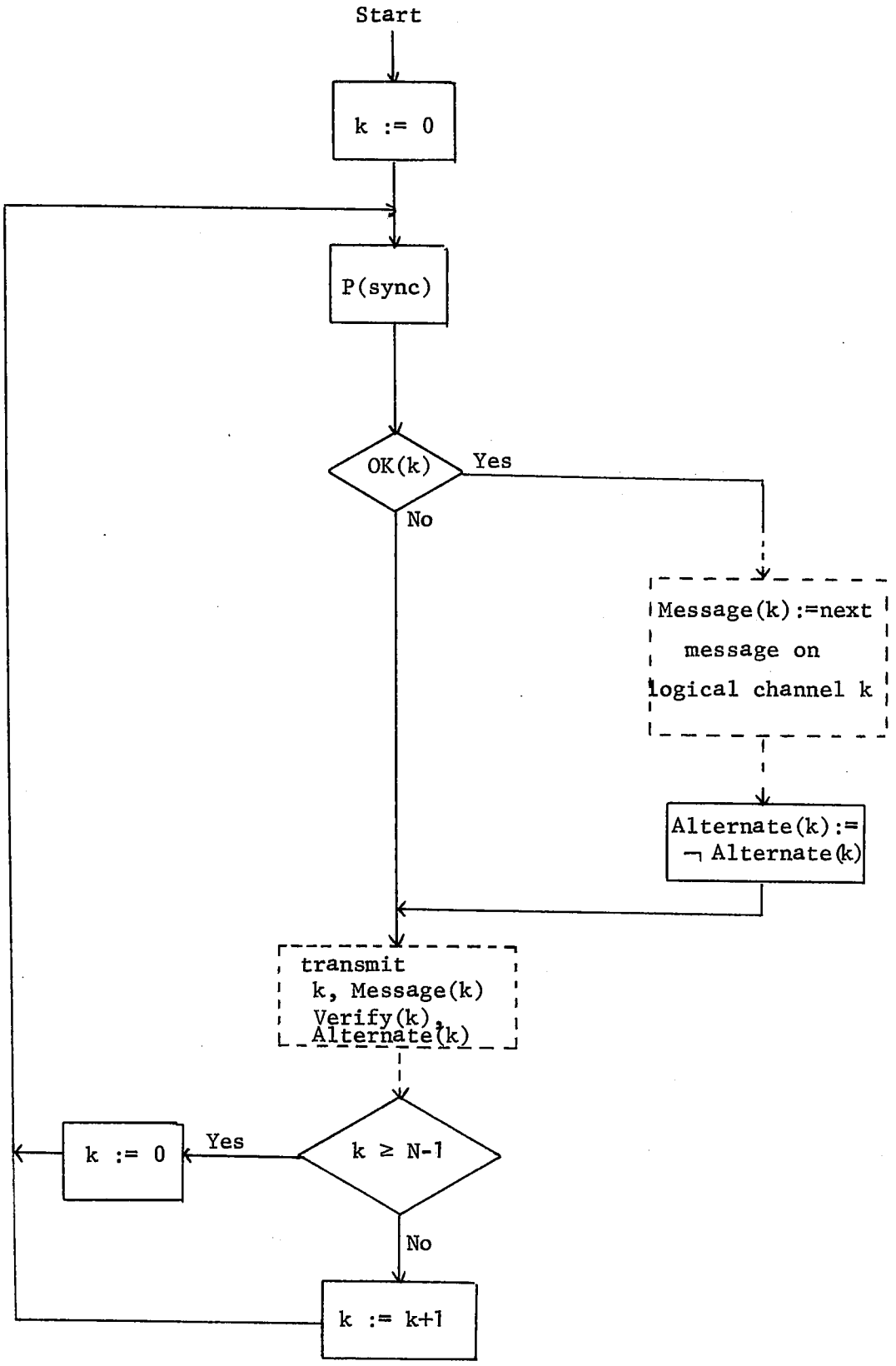

Start
k := 0
P(sync)
OK(k)
Yes
No
Message(k):=next message on logical channel k
Alternate(k):= ¬ Alternate(k)
transmit k, Message(k) Verify(k), Alternate(k)
k ≥ N-1
Yes
k := 0
No
k := k+1

Figure II.2

In Figure II.3, we show a transition graph representation of the same processor. The boxes of the flowchart have been replaced by labeled arcs and the lines of the flowchart have been replaced by nodes. Some of the labels on the arcs have two components separated by a colon : a condition and an action. The condition is a predicate on the values of variables of the process which, if true, allows control to "follow" that arc. The remaining labels have only one component, an action. The action is either null or an assigmment to one or more variables of the process; it represents an operation of the processor. For example, the box of Figure II.2 which performs the operation "P(sync)" corresponds

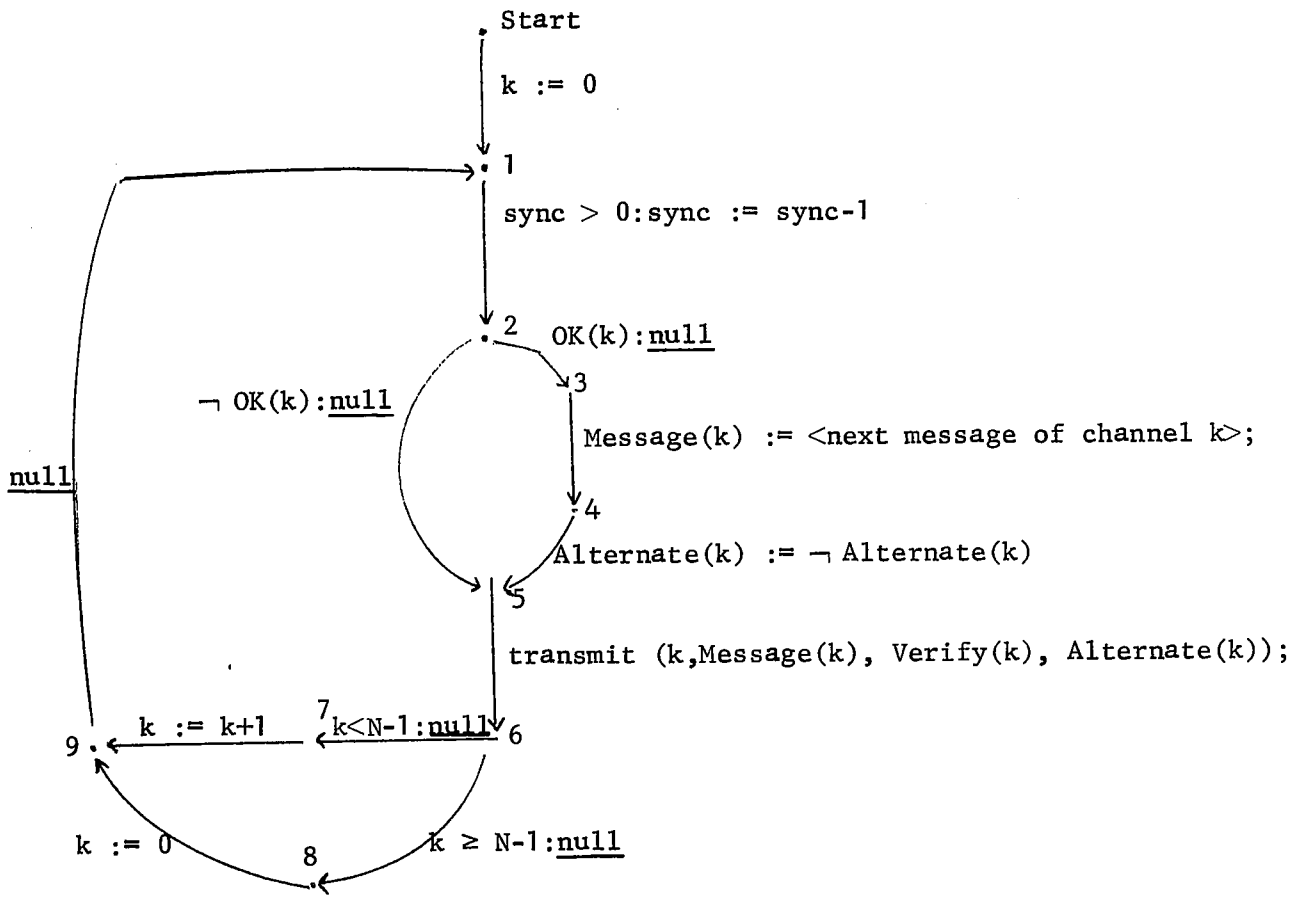


Figure II.3

to the arc between nodes 1 and 2 in Figure II.3 and is labeled with a predicate representing the condition

$$\text{sync} > 0$$

and action

$$\text{sync} := \text{sync}-1.$$

The test and the action are assumed to take place simultaneously and in infinitesimal time.

Each node of the transition graph represents an internal state of the processor described by the program, and each arc leading from a node represents an operation or class of operations which can occur when the processor is in the corresponding state. Thus it is possible to construct computations of the process by following paths through the graph and performing the operations indicated by the arcs. A given arc can be included in such a path only if its condition label is true at the time it is encountered. If all of the operations of the processor are represented by the appropriate arcs, then all such paths through the graph correspond to computations. Conversely, every computation of the process corresponds to some path. This can be made more precise by the following construction.

Let $\underline{p}$ be the process (P,Y,Σ), and suppose that V, the set of internal states of P, is finite. Let G be a directed graph with labeled arcs such that:

(a) The nodes of G are the elements of V.

(b) The arcs are represented by ordered pairs of elements of V and the set of arcs is denoted by Γ.

(c) Associated with each arc $\langle v,w \rangle$ in Γ is a label consisting of two parts: a predicate $\varphi_{\langle v,w \rangle}(y)$ on the values of Y and a function $t_{\langle v,w \rangle}(y)$ which maps the set of values of Y into itself. Denote the set of labels by L.

Thus the graph G is a triple (V,Γ,L).

Definition: The graph $G = (V,\Gamma,L)$ is a transition graph representation of the process p if and only if the following is true:

For any values y_1 and y_2 of Y and any elements v_1 and v_2 in V,

$$(y_1,v_1) \xrightarrow{P} (y_2,v_2)$$

if and only if

(a) there is an arc $\langle v_1,v_2 \rangle$ in Γ,

(b) $\varphi_{\langle v_1,v_2 \rangle}(y_1)$ = true, and

(c) $y_2 = t_{\langle v_1,v_2 \rangle}(y_1)$.

Thus a directed graph of this form is a transition graph representation of a process if and only if its arcs exactly represent the operations of the processor.

In Figure II.3, the action labels were shown as assignments, whereas in the definition they are presented as functions. We will use this "assignment notation" as a useful way of representing functions on the states of information sets, i.e., the assignments describe how the functions operate on the individual elements of an information set. For example, if the set Y consists of the set of elements {a,b,c}, then

the assignment

$$\underline{a} := \underline{b}+\underline{c}$$

is shorthand for the function

$$t(\underline{a},\underline{b},\underline{c}) = \{\underline{b}+\underline{c},\underline{b},\underline{c}\}$$

while the two simultaneous assignments

$$\underline{a} := 0$$

$$\underline{b} := \underline{a}+\underline{c}$$

are shorthand for the function

$$t'(\underline{a},\underline{b},\underline{c}) = \{0,\underline{a}+\underline{c},\underline{c}\}.$$

Both functions take as arguments states of Y and have values which are also states of Y. Throughout the thesis, we will use the assignment notation when it is convenient, but we will always interpret it as a function.

Computational Paths

It is now possible to show that paths through a transition graph representing a process correspond to the computations of that process and conversely. Let us define a simple path through a transition graph $G = (V,\Gamma,L)$ as a sequence of nodes of V,

$$\{v_0,v_1,v_2,\ldots\}$$

either finite or infinite, such that for each $i = 0,1,2,\ldots$, there is an arc $\langle v_i,v_{i+1}\rangle$ in Γ. That is, a simple path is just a way of tracing

through the transition graph by following arcs. A computational path is a sequence of pairs

$$\{(y_0,v_0),(y_1,v_1),(y_2,v_2),\ldots\}$$

of values of Y and elements of V, such that

$$\{v_0,v_1,v_2,\ldots\} \text{ is a simple path}$$
$$\varphi_{<v_i,v_{i+1}>}(y_i) = \underline{\text{true}} \text{ for all } i = 0,1,2,\ldots$$
$$y_{i+1} = t_{<v_i,v_{i+1}>}(y_i) \text{ for all } i = 0,1,2,\ldots$$

That is, a computational path is a way of tracing through a transition graph while performing the tests and operations indicated by the labels on the arcs.

Lemma II.1: Let $\underline{p} = (P,Y,\Sigma)$ be a process, let $G = (V,\Gamma,L)$ be a transition graph representation of $\underline{p}$, and let $(y_0,v_0) \in \Sigma$. Then the sequence of pairs

$$\{(y_0,v_0),(y_1,v_1),(y_2,v_2),\ldots\}$$

is a computation of $\underline{p}$ if and only if it is a computational path in G.

Proof: From the definition of transition graph, the operation

$$(y_i,v_i) \xrightarrow{P} (y_{i+1},v_{i+1})$$

is part of the processor p if and only if

$$\varphi_{<v_i,v_{i+1}>}(y_i) = \underline{\text{true}}$$

$$y_{i+1} = t_{<v_i,v_{i+1}>}(y_i)$$

for any i = 0,1,2,...,

That is, the operation is part of P if and only if the arc $<v_i,v_{i+1}>$ can be included in a computational path. The lemma follows by induction on the length of the sequence.

Transition Graphs of Combinations of Processes

We are now ready to present the main result of this chapter. Let $\underline{p} = (P,Y,\Sigma)$ and $\underline{p}' = (P',Y',\Sigma')$ be processes represented by transition graphs $G = (V,\Gamma,L)$ and $G' = (V',\Gamma',L')$, respectively. Let the process $\underline{q} = \underline{p} \times \underline{p}'$ be the product of $\underline{p}$ and $\underline{p}'$ according to the definition of the previous chapter. We will construct a graph $H = (W,\Delta,M)$ and show that it is a transition graph representation of $\underline{q}$. Then as a consequence of the previous lemma, the set of computations of $\underline{q}$ will correspond exactly to the set of computational paths in this graph.

To construct H, let the set of nodes W be the set $V \times V'$ - i.e., the nodes of H are pairs of nodes of G and G'. Let the set of arcs Δ and the set of labels M be defined as follows:

(a) for each arc $<v_1,v_2>$ of Γ with label

$$\varphi_{<v_1,v_2>}(y):t_{<v_1,v_2>}(y)$$

and for each node v' of V', let an arc

$$<w_1,w_2> = <(v_1,v'),(v_2,v')>$$

be included in Δ and labeled by

$$\theta_{<w_1,w_2>}(y \cup y'):u_{<w_1,w_2>}(y \cup y')$$

where

$$\theta_{\langle w_1,w_2\rangle}(y \cup y') = \varphi_{\langle v_1,v_2\rangle}(y)$$

$$u_{\langle w_1,w_2\rangle}(y \cup y') = t_{\langle v_1,v_2\rangle}(y) \cup (y' \setminus y)$$

(b) symmetrically, for each arc $\langle v_1',v_2'\rangle$ of Γ' with label

$$\varphi'_{\langle v_1',v_2'\rangle}(y'):t'_{\langle v_1',v_2'\rangle}(y')$$

and for each node v of V, let an arc

$$\langle \bar{w}_1,\bar{w}_2\rangle = \langle (v,v_1'),(v,v_2')\rangle$$

be included in Δ and labeled by

$$\bar{\theta}_{\langle \bar{w}_1,\bar{w}_2\rangle}(y \cup y'):u_{\langle \bar{w}_1,\bar{w}_2\rangle}(y \cup y')$$

where

$$\theta_{\langle \bar{w}_1,\bar{w}_2\rangle}(y \cup y') = \varphi'_{\langle v_1',v_2'\rangle}(y')$$

$$\bar{u}_{\langle \bar{w}_1,\bar{w}_2\rangle}(y \cup y') = t'_{\langle v_1',v_2'\rangle}(y') \cup (y \setminus y')$$

(c) no arcs are included in Δ except by virtue of (a) or (b).

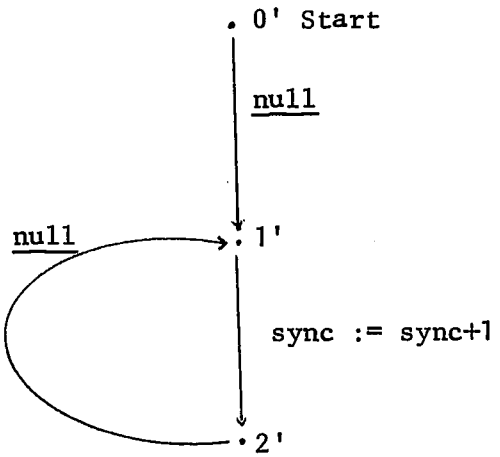
That is, each arc of G has a corresponding arc in H for every node of G' and each arc of G' has a corresponding arc in H for every node of G. The condition and action labels on the arcs of H are exactly the same as those on the corresponding arcs of G and G', but defined in terms of the information set union $Y \cup Y'$. Thus the condition $\theta_{\langle w_1,w_2\rangle}$ on arc $\langle w_1,w_2\rangle = \langle (v_1,v'),(v_2,v')\rangle$ above is a predicate on values of $Y \cup Y'$ which is equal to $\varphi_{\langle v_1,v_2\rangle}$ defined on the values of the Y-component only. Similarly, the function $u_{\langle w_1,w_2\rangle}$ denotes an operation on the value $Y \cup Y'$ equal to the operation defined by

$t_{\langle v_1, v_2 \rangle}$ on values Y, leaving that part of Y' not in the intersection unchanged. Note that the assigmment notation discussed above simplifies the labels on arcs of H considerably.

We can see intuitively how the graph H is formed from an example. Figure II.4 shows transition graph G' representing a small, useless process which cooperates with that of graph G of Figure II.3. Figure II.5 shows the "product" H of these two transition graphs. Along the side of the figure is a copy of G, stretched out in a "linear" form; and similarly G' is depicted at the top of the figure. Some of the labels are omitted or abbreviated where they can be deduced from context. H is essentially formed from a copy of G for each node of G' "crossed" with a copy of G' for each node of G. Note that from most nodes, there are two arcs which can be followed at any given time. This reflects the natural non-determinacy of two processes in which an operation of either processor might be executed next. The paths through the graph intuitively reflect the computations which can result from the two processors working together. This conclusion is based, in part, on the next lemma.

Lemma II.2: The directed graph H constructed above is a transition graph representation of the process $\underline{q} = \underline{p} \times \underline{p}'$.

Proof: Note that W, the set of nodes of the graph H, is exactly the set of pairs of internal states of P and P', which is also the set of internal states of Q by the definition of product processor in Chapter I. Thus, H is a transition graph representation of $\underline{q}$ if and only if there is an exact correspondence



The transition graph G' representing a simple process

Figure II.4

between the arcs in Δ and the operations or classes of operations in Q. Suppose that

(1) $(y_1 \cup y_1', (v_1, v_1')) \xrightarrow{Q} (y_2 \cup y_2', (v_2, v_2'))$

and suppose that this is true because

(2) $(y_1, v_1) \xrightarrow{P} (y_2, v_2)$,

with $v_1' = v_2'$, and $y_1' \setminus y_1 = y_2'$. I.e., suppose the operation (1) has been included in the product processor Q because it is part of the component processor P (a symmetric argument would apply for P'). Then since G is a transition graph representation of p, there is an arc $\langle v_1, v_2 \rangle \in \Gamma$ such that

$$\varphi_{\langle v_1, v_2 \rangle}(y_1) = \underline{\text{true}}$$

and

$$y_2 = t_{\langle v_1, v_2 \rangle}(y_1)$$

Thus, by construction of H, there is an arc

$$\langle (v_1, v_1'), (v_2, v_2') \rangle = \langle w_1, w_2 \rangle$$

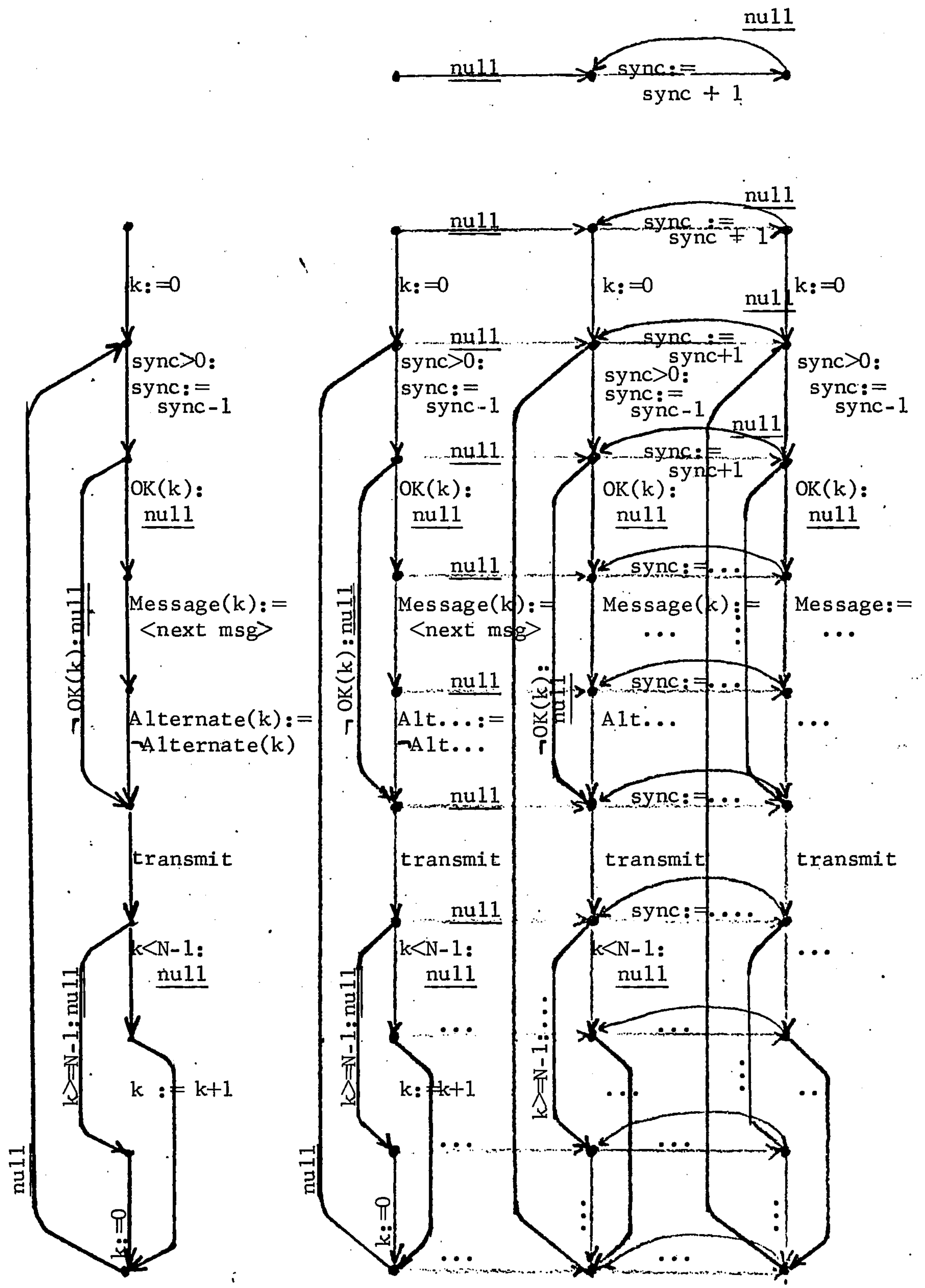

null
null
sync:=
sync + 1
null
null
sync :=
sync + 1
k:=0
k:=0
k:=0
k:=0
null
null
null
sync :=
sync+1
sync>0:
sync:=
sync-1
OK(k):
null
Message(k):=
<next msg>
Alternate(k):=
¬Alternate(k)
¬OK(k):null
transmit
k<N-1:
null
k>=N-1:null
k := k+1
k:=0
null
Alt...:=
¬Alt...
Alt...
Message:=
...
sync:=...
¬OK(k):null
k>=N-1:...

Figure II.5

(since $v_1' = v_2'$) in Δ with

$$\theta_{\langle w_1, w_2\rangle}(y_1 \cup y_1') = \underline{\text{true}}$$

and

$$y_2 \cup y_2' = u_{\langle w_1, w_2\rangle}(y_1 \cup y_1').$$

Conversely, suppose $\langle w_1, w_2\rangle$ is an arc of Δ with label

$$\theta_{\langle w_1, w_2\rangle}(y \cup y') : u_{\langle w_1, w_2\rangle}(y \cup y')$$

and suppose $y_1 \cup y_1'$ is any value of $Y \cup Y'$, with $\theta_{\langle w_1, w_2\rangle}(y_1 \cup y_1') = \underline{\text{true}}$. Then there is a corresponding arc in either Γ or Γ' - say Γ since the argument is symmetrical for Γ' - such that

$$\langle w_1, w_2\rangle = \langle (v_1, v'), (v_2, v')\rangle,$$
$$\langle v_1, v_2\rangle \in \Gamma,$$

and

$$\varphi_{\langle v_1, v_2\rangle} : t_{\langle v_1, v_2\rangle}$$

is the label on this arc. Thus, by construction,

$$\varphi_{\langle v_1, v_2\rangle}(y_1) = \underline{\text{true}};$$

and furthermore the operation

$$(y_1, v_1) \xrightarrow{P} (t_{\langle v_1, v_2\rangle}(y_1), v_2)$$

is part of P, since G is a transition graph representation of p. Thus, the operation

$$(y_1 \cup y_1', (v_1, v')) \xrightarrow{Q} (t_{\langle v_1, v_2\rangle}(y_1) \cup (y_1' \setminus y_1),\ (v_2, v'))$$

is part of the product processor. Since by construction

$$u_{\langle w_1,w_2\rangle}(y_1 \cup y_1') = t_{\langle v_1,v_2\rangle}(y_1) \cup (y_1' \setminus y_1).$$

this is exactly the operation

$$(y_1 \cup y_1', w_1) \xrightarrow{Q} (u_{\langle w_1,w_2\rangle}(y_1 \cup y_1'), w_2).$$

Thus the definition of transition graph is satisfied and H is a transition graph representation of the process $\underline{q}$.

We have shown that we can construct a transition graph representing two cooperating processes from the transition graphs representing each one individually. In keeping with previous notation, we can say

$$H = G \times G'.$$

Clearly, this product operation is both commutative and associative, just as the product operation on processes is. That is, graphs $G \times G'$ and $G' \times G$ are indistinguishable for our purposes, as are $G \times (G' \times G'')$ and $(G \times G') \times G''$.

Comments

The transition graph representation of a process will be used to facilitate proving statements about all of the computations of a process. Note that such a representation is not necessarily unique, but that Lemma II.1 guarantees that all transition graphs representing the same process have exactly the same set of computational paths. Furthermore, because of the correspondence between operations of a processor

and arcs of a transition graph, the latter can be used as a definition for the former.

Without loss of generality, we can assume that transition graphs (and their corresponding processors) have at most one node designated as a start node and at most one halt node. That is, a processor and information set can easily be redefined so that this is true without effectively changing the computations it can make or the way it cooperates with other processors. Similarly, we can assume that we can delete arcs with identically false conditions and subgraphs which are never on any simple path beginning at the start node. This is because no action of any processor can force the given processor to traverse such arcs or enter states represented by such subgraphs.

There is one important situation in operating system programming which is not conveniently represented by our model of computation and by transition graphs. This is the interruptible processor, i.e., the processor which can have a forced transfer of control imposed upon it by another. However, this is not a serious restriction if the interrupt handler and the "interruptible" process are two separate, cooperating processes, each with its own abstract processor. The interrupt handler would effectively be a loop with a blocking state which is unblocked whenever the interrupt condition occurs. The non-determinacy of the interrupt system is then represented by the cooperation of two processes. This approach is in keeping with efforts to rationalize interrupt systems by imposing some process structure on them (see, for example, Wirth [1969]).

CHAPTER III

PROVING CORRECTNESS OF COOPERATING PROCESSES

The statement "Program A is correct" can have many meanings. Intuitively, the most common of these attempt to convey the idea that the program does what the designer intends it to do and/or that the thing he intends is the "right" thing in the context of the problem to be solved. Deciding what algorithm is the right one for a problem is, of course, one of the fundamental aspects of the art and science of programming. Determining whether a particular program in a particular language is an effective representation of an algorithm is a non-trivial problem outside the scope of this thesis.

We can, however, consider a more restrictive concept: "Program A is correct with respect to the statement S". In this case, S is a precise, effective, and meaningful statement characterizing the function of the program. Such statements can take many forms. For example, in Knuth [1969], p. 318, we find the statement "...Algorithm T traverses a tree of n nodes in postorder", where the terms "traverse" and "postorder" had been defined previously. In Dijkstra [1965a], we find the statements

(a) that at any moment, at most one of the processes is engaged in its critical section;

(b) that the decision which of the processes is the first to enter its critical section cannot be postponed to eternity;

(c) that stopping a process in its "remainder of cycle" has no effect upon the others.

Van Horn [1966] considers one criterion of correctness in non-deterministic processes to be "output-functionality" - i.e., that the output of a process is a function only of its inputs and is independent of the behavior of other processes. Each of these statements is meaningful in the sense that it is possible to conceive arguments which could confirm or deny them and that it is possible to draw useful conclusions from them. On the other hand, the statement "This compiler compiles programs correctly" is hardly meaningful without a supporting theory about what correct compilation is and how to do it (see, for example, Good and London [1970]).

In this chapter we will consider the type of correctness criteria introduced by Floyd [1967] and subsequently used by Cooper [1968], Manna [1968], King [1969], London [1970], and others. Statements of correctness assert some relationship among the variables of the process during computation, and the process is considered correct if the statements are true for each computation of that process. These statements, called <u>assertions</u>, are typically associated with lines of the flowchart, nodes of the transition graph, or statements of the program representing the process (although Manna and Pnuelli [1970] have generalized upon this). The assertions take the form of predicates on the values of the process variables which say something about their "meaning". For example,

$$\underline{a} = \underline{b}+3$$

asserts a clear relationship between the values of the variables <u>a</u> and <u>b</u> which is alleged to be true for every computation when the process is executing the corresponding statement of the flowchart, graph, or program.

Assertions specified in this manner can be considered a form of redundancy in programming. That is, the designer specifies an algorithm in a programming language and then a second time (usually incompletely) as a collection of predicates describing the states of the information set. Proving that the process is "correct" with respect to the assertions, then, amounts to showing the mutual consistency of these two specifications. More often than not, both will contain bugs in their initial form. However, we will see that the nature and difficulty of certain kinds of bugs is different in the two characterizations.

In this chapter we will consider assertions about cooperating processes. The Induction Theorem of Floyd [1967] will be applied to verify such processes with respect to these assertions, using the product transition graph representation from the previous chapter. It will be apparent that any attempt to verify a product process this way will be too cumbersome to be practical. However, certain simplifications can be derived from the structure of the components. This chapter concludes with a presentation of some of these techniques and an example of their application.

Assertions and the Induction Theorem

In order to present these results we will use the following notation and terminology.

Definition: Let $p = (P,Y,\Sigma)$ be a process represented by transition graph $G = (V,\Gamma,L)$. An assertion associated with a node $v \in V$ is a predicate on the values of Y. The process p is correct with respect to the assertion α_v if and only if for each state (y,v) which occurs in a computation of p,

$$\alpha_v(y) = \underline{\text{true}}.$$

The process p is correct with respect to a set of assertions if and only if it is correct with respect to each assertion in the set.

Example: Figure III.1(a) shows a "producer" and a "consumer" process designed to communicate through a ring buffer of N elements. (The parallel block structure notation of Dijkstra [1965a] is used here. Each statement between the parbegin-parend brackets is considered to define a separate processor, and the operand of that process consists of the variables declared on the normal scope of the corresponding statement or its parts.) Synchronization is achieved with two semaphores, E indicating the number of empty buffers and F indicating the number of full buffers. Figure III.1(b) shows a transition graph representation of the producer with assertions attached to its nodes.

At node 1 we have asserted that the sum of semaphores E and F is either N-1 or N and that each is non-negative. At node 2 we have asserted that the sum of E and F is either N-2 or N-1. Similar assertions are applied to each node of the transition graph. If these assertions are true for every computation of the producer (i.e., for every computational path of Figure III.1(b)), then that process is considered correct with respect to them.

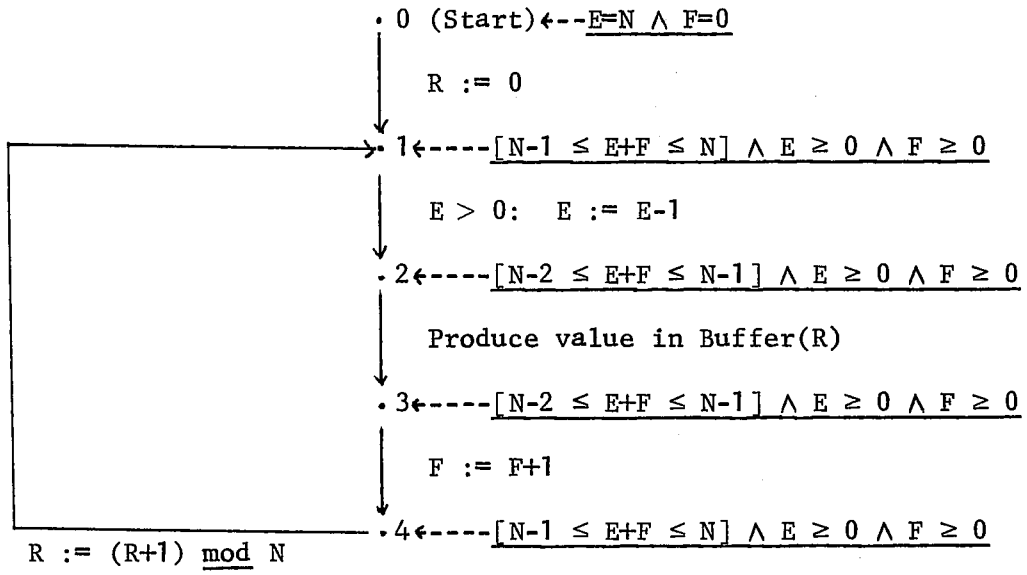
```
parbegin semaphore E=N, F=0;
         array buffer(0:N-1);

     Producer Process:  begin integer R;
         for R := 0, (R+1)mod N while true do
              begin P(E);
                   <produce value in buffer(R)>;
                   V(F) end;
         end;

     Consumer Process:  begin integer S;
         for S := 0,(S+1)mod N while true do
              begin P(F);
                   <consume value from buffer(S)>;
                   V(E) end;
         end;
parend;
```

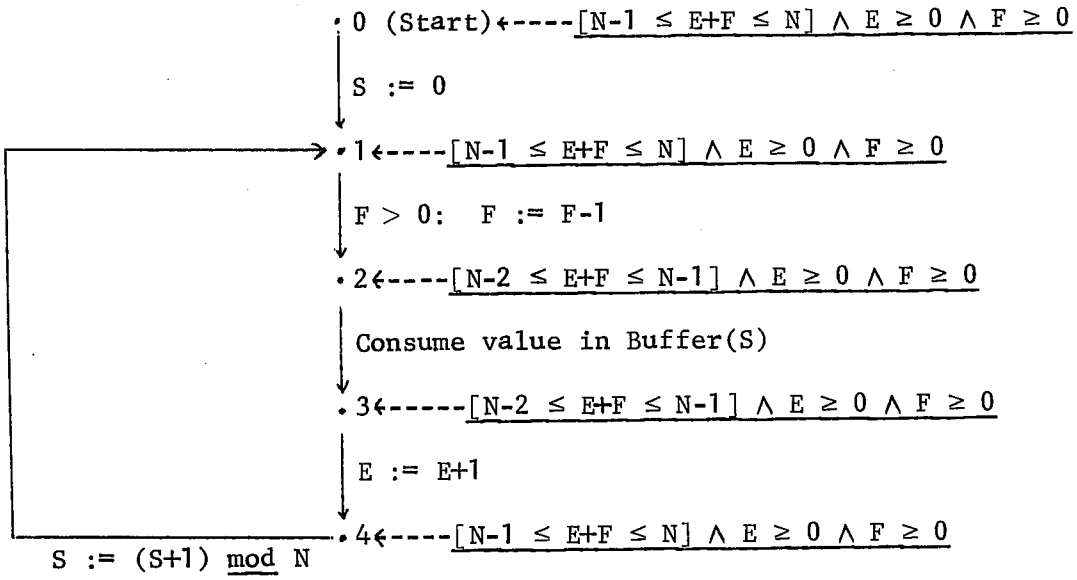
Algol-like description of Producer and Consumer

Figure III.1(a)



Transition graph of the Producer

Figure III.1(b)



Transition graph of the Consumer

Figure III.1(c)

Now consider a set A consisting of assertions for each of the nodes of a transition graph G - i.e.,

$$A = \{\alpha_v(y) \mid v \in V\}.$$

Let y_1 be an arbitrary value of the information set Y, let $\langle v_1, v_2\rangle$ be an arbitrary arc in the set Γ, and let $\varphi_{\langle v_1,v_2\rangle}$:$t_{\langle v_1,v_2\rangle}$ be its label. Suppose that the truth of the assertion $\alpha_v(y_1)$ and of the condition label $\varphi_{\langle v_1,v_2\rangle}(y_1)$ implies the truth of the assertion α_{v_2} on the new value of Y, that is, the truth of $\alpha_{v_2}(t_{\langle v_1,v_2\rangle}(y_1))$. If this assumption were true for all values of Y and all arcs in Γ and if the assertion for every initial state is true, then we could infer by induction the truth of all assertions on all states which occur in computations.

This can be expressed as a well-formed formula of logic as follows:

$$(1)\quad [\alpha_{v_1}(y_1) \wedge \varphi_{\langle v_1,v_2\rangle}(y_1)] \supset \alpha_{v_2}(t_{\langle v_1,v_2\rangle}(y_1))$$

For example, if arc $\langle v_1,v_2\rangle$ is the arc $\langle 1,2\rangle$ of Figure III.1(b), then (1) becomes

α_1: $[[N-1 \leq E+F \leq N] \wedge [E \geq 0] \wedge [F \geq 0] \wedge$

$\varphi_{\langle 1,2\rangle}$: $[E > 0]] \supset$

$\alpha_2 \circ t_{\langle 1,2\rangle}$*: $[[N-2 \leq (E-1) + F \leq N-1] \wedge [(E-1 \geq 0] \wedge [F \geq 0]]$.

Note that in the right side of the implication the new value of each variable is inserted wherever that variable appears in Figure III.1(b); in this case, (E-1) appears for each instance of E and F is unchanged.

The Induction Theorem states that in order to verify that a process is correct, it is sufficient to show that implications of the form of (1) are true for all values of the information set. Establishing this can be done informally or formally. For example, King [1969] and Good [1970] discuss mechanical theorem-proving techniques which require an explicitly-stated set of axioms in a formal system of logic, such as the Peano axioms for integers or one of the "computer" arithmetic systems discussed by Hoare [1969]. In such cases the implications are established by detailed formal proofs. However, Hoare pointed out that for many purposes, semi-formal arguments can be used instead. Such arguments can be regarded as outlines of sketches of formal proofs. They are presented with the understanding that they can be transformed directly into correct, formal proofs if necessary. For other purposes,

*The notation $\alpha \circ t$ means the composition of functions α and t with t applied first to the argument, then α applied to the result.

completely informal, though rigorous, arguments are sufficient. For example, the reader can easily convince himself that the example implication above is true for all integers E,F, and N simply by noting that every positive integer is greater or equal to one. The conclusions then follow from the hypotheses by subtracting one from both sides of each inequality.

In this thesis we will use semi-formal arguments and the notation of first-order logic whenever it is convenient. Well-formed formulas, such as (1), are constructed in the standard way from the ordinary logical punctuation or connective symbols, symbols for individual, function, and predicate constants, and symbols for individual variables. In particular, the symbols

$$x,\ y,\ z,\ x_1,\ y_1,\ z_1,\ \text{etc.}$$

(i.e., the same as those representing values of information sets) stand for individual variables, and all other non-punctuation symbols stand for constants. All logical connectives associate to the left except where indicated by square brackets, "[" and "]". To reduce the number of brackets in a formula, the "dot" notation of Church ([1956], p. 75) is used. Each dot represents a left bracket placed in the position of the dot and a corresponding right bracket placed immediately before the next unbalanced right bracket, or at the end, if there is none. For example, the formula

$$\forall y \cdot [P(y)] \supset \cdot\ Q(y) \wedge R(y)$$

is an abbreviation for

$$\forall y\ [[P(y)] \supset [Q(y) \wedge R(y)]]$$

Finally we will use the symbols $\bigwedge_{i\in I} w_i$ and $\bigvee_{i\in I} w_i$ to represent finite conjunctions and disjunction, respectively, of the well-formed formulae

$$w_{i_1}, w_{i_2}, \ldots, w_{i_k},$$

where I is a finite index set consisting of the members

$$i_1, i_2, \ldots, i_k.$$

Well-formed formulae will have only one "interpretation" (in the formal sense) for a given verification problem. That is, there will be only one domain of objects with which individual variables and constants will be associated, namely the set of values of the information set of the process under consideration. All function and predicate constants will be interpreted as functions and predicates on this domain, and all relevant axioms and theorems applying to those functions will be assumed. These are exactly the axioms and theorems to which the programmer appeals, either explicitly or implicitly, when programming the process. In general, the symbols chosen to represent specific objects, functions and predicates will be their conventional names. This interpretation is called the intended interpretation to distinguish it from other possible interpretations of the same formulae (see Manna[1968]).

An assignment to a well-formed formula is an association of the free individual variables of that formula with elements from the domain of the interpretation. An assignment satisfies a formula if that formula is true when the values associated with all constants and variables are evaluated in the normal way. A well-formed formula ω, is valid if and

only if every assignment satisfies it, or equivalently, if and only if no assignment satisfies the well-formed formula $\neg \omega$. We will use the notation

$$\vdash \omega$$

to mean that ω is valid in the intended interpretation.

We can now state and prove the Induction Theorem in our notation.

Induction Theorem (Floyd [1967]): Let $\underline{p} = (P,Y,\Sigma)$ be a process represented by the transition graph $G = (V,\Gamma,L)$ and let $A = \{\alpha_v \mid v \in V\}$ be a set of assertions on the nodes of G. To prove that $\underline{p}$ is correct with respect to A, it is sufficient to prove that

$$(2) \quad \vdash \forall y \cdot [(y,0) \in \Sigma \supset \alpha_0(y)] \wedge \bigwedge_{\langle v,w\rangle \in \Gamma} \cdot \alpha_v(y) \wedge \varphi_{\langle v,w\rangle}(y) \supset \alpha_w(t_{\langle v,w\rangle}(y))$$

where 0 is the sole starting node of G.

Note that formula (2) has two parts, both quantified over all values of Y. The first part states that α_0 (on the start node) is true for all initial states of the process. The second part is a conjunction of instances of (1) for all arcs of the graph.

Proof: Suppose (2) has been proved. Let

$$C = \{(y,0), (y_1,v_1), (y_2,v_2),\ldots\}$$

be any computation of $\underline{p}$. Then by the first part of (2), $\alpha_0(y_0)$ is true. Let v_0 denote the node 0, and suppose for any

integer $i \geq 0$, $\alpha_{v_i}(y_i)$ is *true*. From the second part of (2) we can infer that

$$\alpha_{v_i}(y_i) \wedge \varphi_{\langle v_i, v_{i+1}\rangle}(y_i) \supset \alpha_{v_{i+1}}(t_{\langle v_i, v_{i+1}\rangle}(y_i))$$

is *true*. Since C is a computation, $\varphi_{\langle v_i, v_{i+1}\rangle}(y_i)$ is *true* and $y_{i+1} = t_{\langle v_i, v_{i+1}\rangle}(y_i)$ by Lemma II.1. Thus $\alpha_{v_{i+1}}(t_{\langle v_i, v_{i+1}\rangle}(y_i))$ -- i.e., $\alpha_{v_{i+1}}(y_{i+1})$ -- is *true*. By induction, the assertions of A are true for all states of C, and since C was arbitrary, the process p is correct with respect to C.

There are several important observations about this theorem which should be noted. First, Floyd showed that it is not essential to specify a complete set of assertions. One assertion for each loop and one for the halt node (if any) are sufficient because the rest can be generated as follows. Suppose that for some node v in V, no assertion has been specified but that for each node v_j such that $\langle v, v_j\rangle$ is an arc of Γ, an assertion α_{v_j} has been specified or already generated. Let α_v be the assertion

$$\bigwedge_{\langle v, v_j\rangle \in \Gamma} [\varphi_{\langle v, v_j\rangle}(y) \supset \alpha_{v_j}(t_{\langle v, v_j\rangle}(y))].$$

Then by a simple theorem of the propositional calculus

$$\vdash \alpha_v(y) \wedge \varphi_{\langle v, v_j\rangle}(y) \supset \alpha_{v_j}(t_{\langle v, v_j\rangle}(y))$$

for each arc $\langle v, v_j\rangle$, so that the corresponding part of the conjunction in (2) is always *true*. Since the transition graph is finite and every

cycle (loop) contains at least one assertion, this method of generating assertions terminates and includes all nodes of a transition graph. It does not depend in any way on the determinism or non-determinism of the process.

A second observation is that the converse of the Induction Theorem is not true; that is, a process may be correct with respect to a set of assertions but the well-formed formula (2) may not be valid in the intended interpretation. This is the case if the assertions are too weak, as in the example in the first part of Chapter IV. However, there is always a non-trivial set of assertions which satisfies (2). This is the set of minimal assertions, defined by

$$A^* = \{\alpha_v^*(y) \mid \alpha_v^*(y) \equiv \text{"}(y,v) \text{ is in some computation of } \underline{p}\text{"}\}.$$

(In this case, the conjunction

$$\alpha_v^*(y) \wedge \varphi_{\langle v,w\rangle}(y)$$

is true if and only if the state (y,v) and the arc $\langle v,w\rangle$ are both part of a computational path. Whenever it is,

$$\alpha_w^*(t_{\langle v,w\rangle}(y))$$

is also true and (2) follows immediately.)

The minimal assertions are, in general, not recursively constructable. That is, no algorithm can be designed to generate the minimal predicates from an arbitrary transition graph. This is because whenever the set Σ is recursively enumerable, so is the set of values of y for which $\alpha_v^*(y) = \underline{\text{true}}$. (Simply follow all computational paths.) If the

set of values for which $\alpha_v^*(y)$ = false could be generated, the halting problem for the process p could be decided, contrary to well-known results.

Finally, we should observe that if p is correct with respect to an assertion α_v, then it is obvious that

$$\vdash \forall_y[\alpha_v^*(y) \supset \alpha_v(y)].$$

I.e., the minimal assertion on that node implies all others with respect to which the process is correct. Similarly, if p is correct with respect to α_v and

$$\vdash \forall_y[\alpha_v(y) \supset \alpha_v'(y)],$$

then p is also correct with respect to α_v'. I.e., to prove a process correct with respect to the assertion α_v', it is sufficient to prove it correct with respect to another one which implies the given one. Finally, note that p is correct with respect to two assertions α_v and α_v' if and only if it is correct with respect to their conjunction.

Correctness of Cooperating Processes

Our model makes it relatively easy to extend this method of verifying processes to collections of cooperating processes such as occur in operating systems. Let $p = (P,Y,\Sigma)$ and $p' = (P',Y',\Sigma')$ be two cooperating processes represented by transition graphs $G = (V,\Gamma,L)$ and $G' = (V',\Gamma',L')$ respectively. Lemma II.2 allows us to form a transition graph representing their product, namely $G \times G'$ (which for convenience we will denote by the triple $(V \times V',\Delta,M)$). Since this is a transition graph, the Induction Theorem can be applied to it and a set of assertions

$$\{\beta_{\langle v,v'\rangle} \mid (v,v') \in V \times V'\},$$

such as the sets we will consider below. To verify $\underline{p} \times \underline{p}'$ with respect to this, it is sufficient to show that a well-formed formula similar to (2) is valid in the intended interpretation. In this case, we have

$$(3) \quad \vdash \forall(y \cup y')\,[(y \cup y'),\ (0,0')) \in \Sigma * \Sigma' \supset \beta_{(0,0')}(y \cup y')] \wedge$$

$$\bigwedge_{\langle (v,v'),(w,w')\rangle} \cdot\ \beta_{(v,v')}(y \cup y') \wedge \theta_{\langle (v,v'),(w,w')\rangle}(y \cup y')$$

$$\supset \beta_{(w,w')}(u_{\langle (v,v'),(w,w')\rangle}(y \cup y')),$$

where each subscripted θ:u is the label on the corresponding arc, and where (0,0') is the start node of the product. The formula (3) is apparently not well-formed because $y \cup y'$ is not an individual variable but the value of a function. However, a formula of the form of (3), that is

$$\forall(y \cup y')\ F(y,y'),$$

can be regarded as an abbreviation for

$$\forall z \forall y \forall y'[z = y \cup y' \supset F(y,y')],$$

which is well-formed.

The set of assertions $\{\beta_{\langle v,v'\rangle}\}$ may, of course, consist of any which might describe either process or their interactions. There are three particularly interesting cases which are constructed from assertions on the nodes of the component graphs. Let

$$A = \{\alpha_v \mid v \in V\}$$

and

$$A' = \{\alpha'_v \mid v' \in V'\}$$

be sets of assertions on the nodes of G and G', respectively. Then

$$B = \{\beta_{(v,v')} \mid \beta_{(v,v')}(y \cup y') = \alpha_v(y), (v,v') \in V \times V'\}$$

and

$$B' = \{\beta'_{(v,v')} \mid \beta'_{(v,v')}(y \cup y') = \alpha'_v(y'), (v,v') \in V \times V'\}$$

can be considered extensions of these assertions to the product process. If $\underline{p} \times \underline{p}'$ were correct with respect to, say, B, we could pervert our terminology slightly and say that $\underline{p}$ itself is correct with respect to A *in the context of the product* $\underline{p} \times \underline{p}'$. A third interesting set of assertions is the conjunction of the first two, that is

$$\bar{B} = \{\bar{\beta}_{(v,v')} \mid \bar{\beta}_{(v,v')}(y \cup y') = \alpha_v(y) \wedge \alpha'_{v'}(y'), (v,v') \in V \times V'\}$$

These assertions simply state that the product process is correct with respect to the assertions on both components simultaneously.

These ideas can be easily generalized to products of $\underline{n}$ processes. We observed earlier that the product operator on both transition graphs and processes is commutative and associative. Thus, it makes sense to consider assertions of the form

$$\beta^i_{(v^1,v^2,\ldots,v^n)}(y^1 \cup y^2 \cup \ldots \cup y^n) = \alpha^i_v i(y^i)$$

and

$$\bar{\beta}_{(v^1,v^2,\ldots,v^n)}(y^1 \cup y^2 \cup \ldots \cup y^n) = \alpha^1_v 1(y^1) \wedge \alpha^2_v 2(y^2) \wedge \ldots \wedge \alpha^n_v n(y^n).$$

There is no conceptual difficulty with generalization but the notation

can get extremely cumbersome very quickly. Thus we will present the results of this chapter in terms of products of two processes and simply note where and how to generalize upon them.

Let us show how the Induction Theorem can be used to verify a set of cooperating processes. This problem is based on the program and transition graph of Figure III.2 and derived from famous algorithms of Dijkstra [1965b] and Knuth [1966]. Each of N processes executes the same algorithm with its own identity substituted for i. For purposes of illustration, we have simplified the algorithm to the point where the underlying processor is probably unrealistic. For example, the tests implied by the condition labels, would hardly be implemented by single, indivisible operations in any but the most unusual processors.

The algorithm controls the access of the processes to "critical sections" or program using the integer array C(0:N-1) and the integer k, all initialized to zero. We have not shown the details of the critical section (dotted arc<4,5>) or of the remainder of the program (dotted arc >7,8>). This will be justified in due course.

The reader can imagine an N-dimensional product transition graph,

$$G^0 \times G^1 \times \ldots \times G^N$$

representing the product of these N processes. Such a graph would be impractical to draw for N=2 and virtually impossible for $N \geq 3$. This is apparent because the product has

$$9^N \text{ nodes}$$

and

$$11(9^{N-1})N \text{ arcs.}$$

```
parbegin integer array C(0::N-1) = 0;
     integer k = 0;
          :
          :
     process i: begin integer j;
        L: C(i) := 1;
                    N-1
           while ¬⋀[(k-j) mod N < (k-i) mod N ⊃ C(j) = 0] do;
                    j=0
           C(i) := 2;
           if ⋁[C(j) = 2] then go to L;
             j≠i
           <critical section of process i>;

           k := (i-1) mod N;
           C(i) := 0;
           <remainder of process i in which stopping is allowed>;

           go to L end;
           :
           :
parend;
```

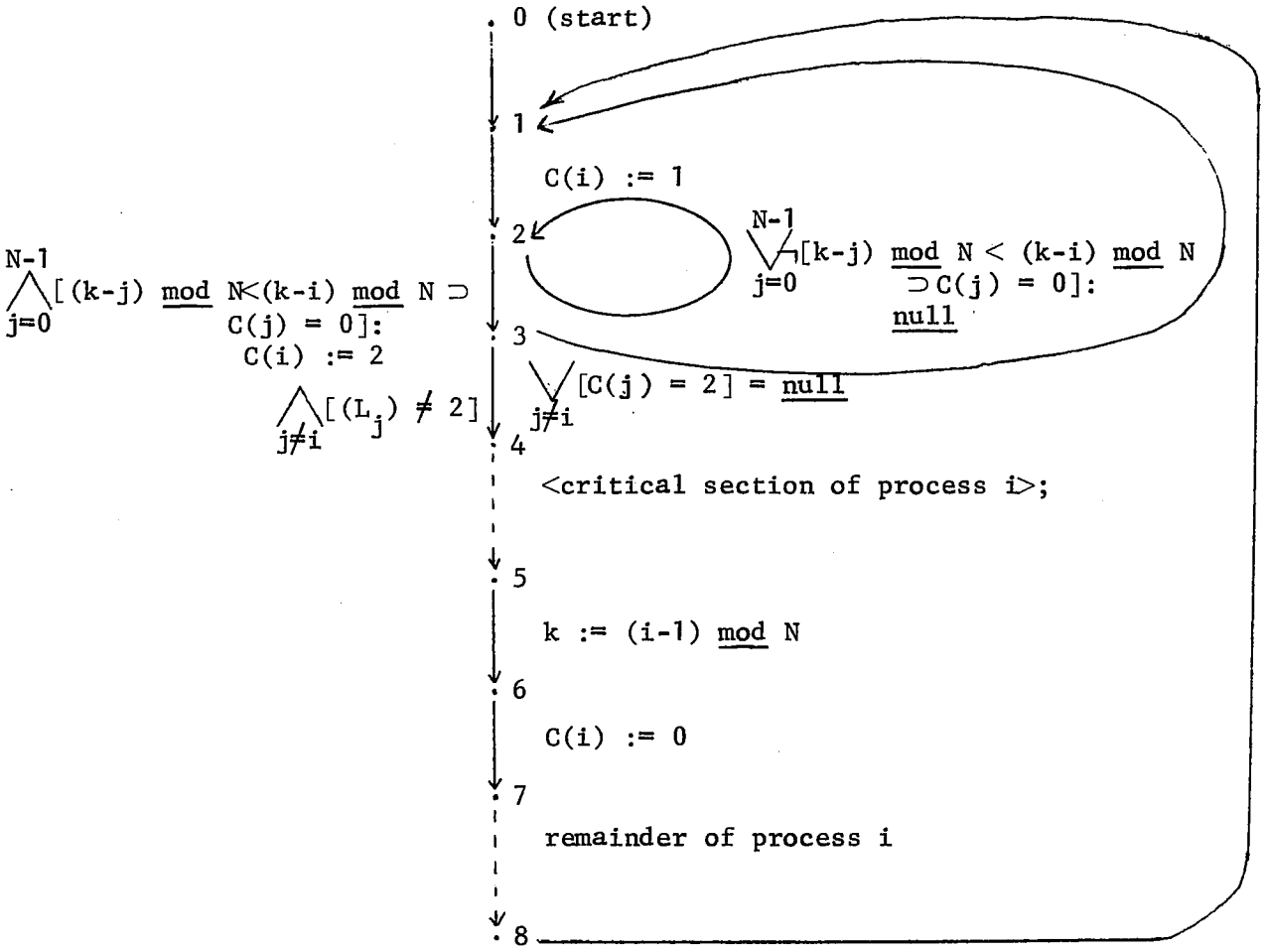


Figure III.2

However, we will be able to proceed without a picture, and in fact we will rarely, if ever, need one.

Table III.1 gives a set of assertions which we claim characterizes the behavior of the $\underline{i}^{th}$ process in the context of the product. These can be associated with the 9^N nodes of the product transition graph in the manner suggested by the description of the set B^i above. Clearly, if we prove that the $\underline{i}^{th}$ process is correct with respect to these assertions in the context of the product, we have also shown that each of the other processes is correct with respect to similar assertions. From these we will be able to conclude that the algorithm

	Assertion	Comment
α_0^i	$C(i) = 0$	The initial condition
α_1^i	$C(i) = 0 \vee C(i) = 2$	Either no claim is made on the critical section or a previous claim has been rejected.
α_2^i	$C(i) = 1$	A tentative claim is made.
α_3^i	$C(i) = 2$	The claim is strengthened.
α_4^i	$C(i) = 2 \wedge \bigwedge_{j \neq i} [C(j) = 2 \supset (k-j) \bmod N < (k-i) \bmod N]$	The only processes for which $C(j) = 2$ are those between process $\underline{i}$ and process $\underline{k}$ in cyclic order.
α_5^i	same as α_4^i	
α_6^i	$C(i) = 2 \wedge k = (i-1) \bmod N$	This is the only time we can know the value of $\underline{k}$.
α_7^i	$C(i) = 0$	
α_8^i	$C(i) = 0$	The claim on the critical section is released.

Table III.1

correctly controls access to the critical sections. For if two processes, i and i' were both in their critical sections (i.e., at node 4 or 5 in both component graphs) during some computation, we would have

$$C(i) = 2 \wedge \bigwedge_{j \neq i} [C(j) = 2 \supset (k-j) \bmod N < (k-i) \bmod N]$$

and

$$C(i') = 2 \wedge \bigwedge_{j \neq i'} [C(j) = 2 \supset (k-j) \bmod N < (k-i') \bmod N]$$

both true, which is impossible for $i \neq i'$ when $0 \leq i < N$, $0 \leq i' < N$.

Using the Induction Theorem to establish the correctness of the process with respect to Table III.1 is an unreasonable task if attempted directly. It would mean showing that a formula of the form of (3) is valid. But this formula has as many terms in its conjunction as there are arcs in the product graph, namely

$$11(9^{N-1})N.$$

Any feasible method of verification must reduce these to a reasonable number. The next section shows how we can use our knowledge about the structure of the processes and the assertions to do this.

Simplifications

A quick inspection shows that the arcs of the product graph fall into 11 categories corresponding to the 11 arcs of a component transition graph. The labels on the arcs within one category are all similar or identical and the proof of the implications corresponding to the arcs within a group follow a similar pattern. That is, verifying the product process is very much like verifying one of its components. Thus, we can take advantage of this to simplify the proof.

Observe that, in principle, it would be possible to prove that a process is correct with respect to a set of assertions in the context of a system of processes by:

1. proving that the process is correct when considered alone, and
2. proving that none of the other processes alters the truth of those assertions.

This decomposition is recursive because a component process of a product might, itself, be a product of other processes; thus part (1) could be decomposed the same way. However, at some level a process can be verified by direct application of the Induction Theorem and part (1) would be proved. The difficulty of part (2) depends upon the structure of the system and in some cases it can be treated by inspection, as we will see below.

The decomposition of a verification can be expressed in formal terms as follows: Let $G = (V,\Gamma,L)$ and $G' = (V',\Gamma',L')$ be the two transition graphs representing processes $\underline{p} = (P,Y,\Sigma)$ and $p' = (P',Y',\Sigma')$, respectively. Let $A = \{\alpha_v \mid v \in V\}$ be a set of assertions on the nodes of G. Then, to prove that $\underline{p}$ is correct with respect to A in the context of the product $\underline{p} \times \underline{p}'$, it is sufficient to prove:

1. $$\vdash \forall y \cdot [(y,0) \in \Sigma \supset \alpha_0(y)] \wedge \bigwedge_{\langle v,w\rangle \in \Gamma} \cdot \alpha_v(y) \wedge \varphi_{\langle v,w\rangle}(y) \supset \alpha_w(t_{\langle v,w\rangle}(y))$$

2. for every node $v \in V$ and every arc $\langle v',w'\rangle \in \Gamma'$,

   $$\vdash \forall(y \cup y')[\alpha_v(y) \wedge \varphi'_{\langle v',w'\rangle}(y') \supset \alpha_v \text{ (new value of } y)]$$

(The "new value of y" in this formula is the result of applying the operation $t'_{<v',w'>}$ to y' - something which was hard to describe in the functional notation (see page 36) but easy to imagine in the assignment notation.) If both parts can be proved, then formula (2) of the Induction Theorem is satisfied for the product process and the correctness follows immediately.

Proving part (1) is straightforward. Part (2) is trivial for all of those cases in which the assertion α_v and the operation $t'_{<v',w'>}$ are "independent" of each other. This could be the case if α_v depends only on variables which are never changed by the process p', or if $t'_{<v',w'>}$ operates only on variables not known to p.

Example: In Table III.1 the assertions α_v for v = 0,1,2,3,7 and 8 (of Figure III.2) depend only on values of C(i). But by inspection, we see that this element of the array C is changed only by the i^{th} process, never by any of the others. Thus these assertions remain invariant under all operations of the other process.

Example: In the producer-consumer system of Figure III.1, the variable R is local to the producer and not known to the consumer. Thus all assertions on the consumer are independent of arc <4,1> of Figure III.1(b), since it describes an operation which changes only R.

The only cases in which part (2) requires any serious attention are those in which the assertion and operation are not obviously independent.

In the example below this includes only five cases out of 99.

Another observation we can make is that, in principle, the details and fine structure of the "local computations" of a process do not affect its correctness in the context of a system. Thus we could replace a sequence of arcs or a subgraph describing such operations by a single arc representing the equivalent effect.

Example: In Figure III.1(b) arc <4,1> describes the assignment

$$R := (R+1) \underline{\text{mod}} N.$$

In practice, this assignment would probably be implemented as a sequence of several operations. One possible way is represented by the fragment of a transition graph in Figure III.3. A test is made to determine whether to increment R or reset it to zero. If it is incremented, this must be done using a temporary (local) accumulator ACC. This detail is completely irrelevant to the question of cooperation between the producer and consumer, so that including it in the transition graph would only serve to complicate matters. Thus, we have simplified the transition graph by using a single arc <4,1> as originally given.

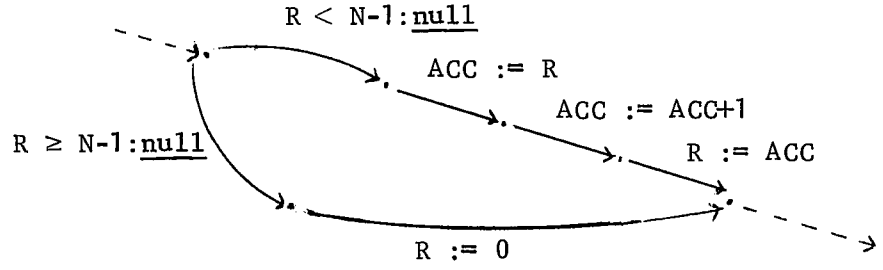


Figure III.3

In this kind of simplification we are replacing a subgraph of one of the component transition graphs by a single arc representing an equivalent operation, subject to the condition that no operation in that subgraph affects any variable mentioned by an assertion on the other process. We wish to infer that the unsimplified system is correct with respect to a particular set of assertions, by showing that the simplified system is correct with respect to that same set. If, as above, we are verifying $\underline{p}$ with respect to the set of assertions A in the context of the product $\underline{p} \times \underline{p}'$, there are two cases to consider:

a. the simplification is done to the transition graph G' which represents $\underline{p}'$, or

b. it is done to graph G representing $\underline{p}$.

For case (a) observe that a proof of part (1) - that $\underline{p}$ is correct when considered alone - is unaffected by the simplification. In part 2 each of the arcs $\langle v',w' \rangle$ involved in the simplification is independent of each of the assertions α_v, since we stipulated these operations must not affect any variable mentioned by any α_v. Thus, a proof of part (2) is also unaffected by the simplification, and so we can infer the correctness of the unsimplified system by proving parts (1) and (2) for the simplified system.

In case (b) we are simplifying the transition graph G, so we must be able to infer the correctness of the unsimplified version of $\underline{p}$ from that of the simplified version. That is, we must be able to conclude that part (1) is true for a suitable extension of the set of assertions

A to the unsimplified case. If the single arc is, indeed, equivalent to the subgraph it replaces, this is no problem. Part (2) follows immediately because the assertions on nodes of the subgraph differ from those at the ends of the equivalent arc only in their statements about variables not mentioned by process p'. Thus, a proof of part (2) on the simplified system also applies to the unsimplified one, and the correctness follows.

Therefore, we can conclude that from the point of view of proving correctness, we can ignore the details of computations on "local" variables and variables not mentioned by the other processes of the system. An obvious extension of this result is that we can also make the same simplification even when the variables are not necessarily local or private to a process; provided that whenever the process is executing in the subgraph in question, we can show that those variables are "temporarily" not mentioned by the other component process. Other, more general types of simplifications fall into the scope of Chapter V.

Example

Let us illustrate the verification of a collection of cooperating processes by completing the example begun in Figure III.2. We can see by inspection that the initial condition - i.e., the first implication of (3) - is valid because all of the elements of the vector C are initially set to zero. Table III.2 is useful in showing that i^{th} component process is correct when considered alone. The implications in this table are exactly the implications in the conjunction of the formula of part (1) above. Note that the right side of each implication shows the new values of the variables resulting from the operation

arc	Formula of part (1)
<0,1>	$C(i) = 0 \supset \cdot\ C(i) = 0 \vee C(i) = 2$
<1,2>	$C(i) = 0 \vee C(i) = 2 \supset 1 = 1$
<2,2>	$C(i) = 1 \wedge [\bigvee_{j=0}^{N-1} \neg \cdot (k-j) \bmod N < (k-i) \bmod N \supset C(j) = 0] \supset C(i) = 1$
<2,3>	$C(i) = 1 \wedge \bigwedge_{j=0}^{N-1} [(k-j) \bmod N < (k-i) \bmod N \supset C(j) = 0] \supset 2 = 2$
<3,1>	$C(i) = 2 \wedge \bigvee_{j \neq i} [C(j) = 2] \supset \cdot\ C(i) = 0 \vee C(i) = 2$
<3,4>	$C(i) = 2 \wedge \bigwedge_{j \neq i} [C(j) \neq 2] \supset \cdot\ C(i) = 2 \wedge \bigwedge_{j \neq i} [C(j) = 2 \supset (k-j) \bmod N < (k-i) \bmod N]$
<4,5>	$C(i) = 2 \bigwedge_{j \neq i} [C(j) = 2 \supset (k-j) \bmod N < (k-i) \bmod N] \supset .C(i) = 2 \wedge \bigwedge_{j \neq i} [C(j) = 2 \supset (k-j) \bmod N < (k-i) \bmod N]$
<5,6>	$C(i) = 2 \wedge \bigwedge_{j \neq i} [C(j) = 2 \supset (k-j) \bmod N < (k-i) \bmod N] \supset .C(i) = 2 \wedge (i-1) \bmod N = (i-1) \bmod N$
<6,7>	$C(i) = 2 \wedge k = (i-1) \bmod N \supset 0 = 0$
<7,8>	$C(i) = 0 \supset C(i) = 0$
<8,1>	$C(i) = 0 \supset \cdot\ C(i) = 0 \vee C(i) = 2$

Table III.2

represented by the corresponding arc. The validity of each of these implications follows directly from the propositional calculus and the following theorems of the intended interpretation:

$$\vdash 0 = 0$$

$$\vdash 1 = 1$$

$$\vdash 2 = 2$$

$$\vdash (i-1) \underline{\text{mod}}\ N = (i-1) \underline{\text{mod}}\ N.$$

Thus we can infer that part (1) is valid in the intended interpretation - that is, that the i^{th} process is correct with respect to the assertions of Table III.1 when considered alone.

Table III.3 shows which of the implications of part (2) are trivial and which are not. Rows marked with an asterisk correspond to operations of the process i' (i' $\neq$ i) which do not affect the variables of the i^{th} process (in this example all of these operations are null). Columns marked with an asterisk correspond to assertions on the i^{th} process which are independent of all variables which can be changed by any other process. In this case, these assertions depend only on the value of C(i). Column 5 is also marked because the assertion α_5^i is the same as α_4^i, so that proving part (2) for one will serve the other, as well. By inspection, we also see that the α_6^i depends only on the variables C(i) and k, neither of which is manipulated by operations on arcs <1,2>, <2,3> and <6,7>. Thus, part (2) holds for these, as well.

There are only five blank entries left in Table III.3. These correspond to cases of (2) which demand some kind of proof. The following arguments are an outline of suitable proofs:

a. arc <1,2>, node 4: The body of (2) becomes

arcs of graph i'	nodes of i^{th} transition graph								
	0	1	2	3	4	5	6	7	8
	*	*	*	*				*	*
<0,1> *	*	*	*	*	*	*	*	*	*
<1,2>	*	*	*	*		*	*	*	*
<2,2> *	*	*	*	*	*	*	*	*	*
<2,3>	*	*	*	*		*	*	*	*
<3,1> *	*	*	*	*	*	*	*	*	*
<3,4> *	*	*	*	*	*	*	*	*	*
<4,5> *	*	*	*	*	*	*	*	*	*
<5,6>	*	*	*	*		*		*	*
<6,7>	*	*	*	*		*	*	*	*
<7,8> *	*	*	*	*	*	*	*	*	*
<8,1> *	*	*	*	*	*	*	*	*	*

Table III.3

$$C(i) = 2 \wedge \bigwedge_{j \neq i} [C(j) = 2 \supset (k-j) \underline{\text{mod}}\ N < (k-i) \underline{\text{mod}}\ N \,] \supset$$

$$. C(i) = 2 \wedge \bigwedge_{\substack{j \neq i \\ j \neq i'}} [C(j) = 2 \supset (k-j) \underline{\text{mod}}\ N < (k-i) \underline{\text{mod}}\ N]$$

$$\wedge [1 = 2 \supset (k-i') \underline{\text{mod}}\ N < (k-i) \underline{\text{mod}}\ N]$$

This follows directly from the propositional calculus and the theorem $\vdash [1 \neq 2]$.

b. arc <2,3>, node 4: The body of (2) becomes

$$C(i) = 2 \wedge \bigwedge_{j \neq i} [C(j) = 2 \supset (k-j) \underline{\text{mod}}\ N < (k-i) \underline{\text{mod}}\ N]$$

$$\wedge \bigwedge_{j' \neq i'} [(k-j') \underline{\text{mod}}\ N < (k-i') \underline{\text{mod}}\ N \supset C(j') = 0] \supset$$

$$.C(i) = 2 \wedge \bigwedge_{\substack{j \neq i \\ j \neq i'}} [C(j) = 2 \supset (k-j) \underline{\text{mod}}\ N < (k-i) \underline{\text{mod}}\ N]$$

$$\wedge [2 = 2 \supset (k-j') \underline{\text{mod}}\ N < (k-i) \underline{\text{mod}}\ N].$$

The second line implies that

$$(k-i) \underline{\text{mod}}\ N < (k-i') \underline{\text{mod}}\ N \supset C(i) = 0.$$

But $C(i) = 2$ from the first line and $i \neq i'$ by hypothesis, so that

$$(k-i') \underline{\text{mod}}\ N < (k-i) \underline{\text{mod}}\ N,$$

and the implication follows by the propositional calculus.

c. arc <6,7>, node 4: The body of (2) becomes

$$C(i) = 2 \wedge \bigwedge_{j \neq i} [C(j) = 2 \supset (k-j) \underline{\text{mod}}\ N < (k-i) \underline{\text{mod}}\ N]$$

$$\supset C(i) = 2 \wedge \bigwedge_{\substack{j \neq i \\ j \neq i'}} [C(j) = 2 \supset (k-j) \underline{\text{mod}}\ N < (k-i) \underline{\text{mod}}\ N]$$

$$\wedge 0 = 2 \supset (k-i') \underline{\text{mod}}\ N < (k-i) \underline{\text{mod}}\ N.$$

This follows from the fact that

$$\vdash \neg [0 = 2]$$

and the propositional calculus.

The last two cases - i.e., arcs <5,6> for nodes 4 and 6 - do not submit to our simplifications. For example, in the case of arc = <5,6>,

node = 4, the formula of part (2) becomes

$$C(i) = 2 \wedge \bigwedge_{j \neq i} [C(j) = 2 \supset (k-j) \bmod N < (k-i) \bmod N]$$

$$\supset C(i) = 2 \wedge \bigwedge_{j \neq i} [C(j) = 2 \supset ((i'-1)-j) \bmod N < ((i'-1)-i) \bmod N],$$

which is not necessarily true if $(i'-1) \neq k$. Thus, our simplification techniques have treated all but two terms in the conjunction in formula (3) corresponding to these two arcs. However, if we consider the conjunctions of assertions on two processes simultaneously, we can complete the verification. By considering the appropriate terms of formula (3) with $\beta_{(v,v')} = \alpha_v \wedge \alpha'_{v'}$, we obtain an implication with an identically false left side, rendering the implication itself true for all values of the variables. This can be seen as follows:

d. arc <5,6>, node 4: The conjunction of the assertions is

$$C(i) = 2 \wedge \bigwedge_{j \neq i} [C(j) = 2 \supset (k-j) \bmod N < (k-i) \bmod N]$$

$$\wedge C(i') = 2 \wedge \bigwedge_{j' \neq i'} [C(j') = 2 \supset (k-j') \bmod N < (k-i') \bmod N].$$

In particular, we can infer from this both

$$(k-i') \bmod N < (k-i) \bmod N$$

and

$$(k-i) \bmod N < (k-i') \bmod N,$$

a contradiction.

e. arc <5,6>, node 6: The conjunction of the assertions is

$$C(i) = 2 \wedge \bigwedge_{j \neq i} [C(j) = 2 \supset (k-j) \bmod N < (k-i) \bmod N]$$

$$\wedge\ C(i') = 2 \wedge k = (i'-1) \bmod N.$$

By substituting $(i'-1) \bmod N$ for k in the first line, we can infer that

$$((i'-1)-i') \bmod N < (i'-1-i) \bmod N.$$

But a property of the integers modulo N is that

$$\begin{aligned} ((i'-1)-i') \bmod N &= (-1) \bmod N \\ &= N-1 \\ &\geq x \bmod N \end{aligned}$$

for any integer x. Since $i \neq i'$, we know that

$$((i'-1)-i') \bmod N > ((i'-1-i) \bmod N,$$

a contradiction. Thus, for these two arcs, the assertions satisfy the appropriate term of formula (3) directly.

This essentially completes the proof that the process of Figure III.2 is correct in the context of a product of identical processes. What we have shown is that the product is correct with respect to a set of assertions of the form

$$\{\alpha_v^0 0 \wedge \alpha_v^1 1 \wedge \ldots \wedge \alpha_v^{N-1}(N-1) \mid (v^0, v^1, \ldots, v^{N-1}) \in V^0 \times V^1 \times \ldots \times V^{N-1}\}.$$

We showed this by using the Induction Theorem on the ith process in isolation, then by showing that the interactions between the ith and (i')th process do not affect the validity of the assertions α_v^i. In

all but five cases, this could be done by inspection. In two of those five cases, we had to consider assertions on at least two component processes at a time.

Comment

The actual proof techniques which might be used to prove well-formed formulas such as (2) and (3) are beyond the scope of this thesis. We are only interested in converting the problem of verifying a system of cooperating processes into a reasonable theorem to be proved. If it is necessary or desirable, any of various forms of mechanical theorem provers can be applied. Or we can be content, as in this thesis, to restrict our attention to problems which can be proved manually.

CHAPTER IV

SOME OTHER FORMS OF CORRECTNESS - ADDING INFORMATION

In the previous chapter, "proving correctness" meant showing that assertions about the values of, and relationships among, the variables of a process are consistent with a transition graph representation of that process. For an interesting class of problems, this is sufficient. That is, some interesting properties of programs can be stated in the form of assertions about the program variables and proved correct. For example, London [1970] proved, albeit laboriously, that a certain sorting algorithm does correctly sort an array in place. Hoare [1971] has used similar methods to help synthesize an algorithm which is a priori correct. Others, including Good [1970], Floyd [1971], and Snowden [1971], have considered man-machine systems based partly on these methods to support the development and/or verification of programs. All of these efforts have been concentrated on processes which are deterministic and which do not communicate with other processes.

When we consider cooperating processes, such as are common in operating systems, the results of the previous chapter are not as effective as they might be. One reason is that, although a property of the system might be expressable in the form of assertions and although the system might be correct with respect to those assertions, the assertions do not contain enough information to allow a proof. That is, the set of assertions may not satisfy formula (2) of the Induction Theorem, even though they correctly characterize all computations of

the product process. In the author's experience, this is the case all too often.

A second reason is that there are interesting questions regarding the correctness of the cooperation which cannot be directly reflected in assertions about process variables. By verifying a collection of processes with respect to a set of assertions, we are effectively saying "for every computation, the value of the variables have a specific property or relationship." We would also like to ask "Are there any computations with a certain property?" or "Does every computation reach a certain state?"

In this chapter we will consider some ways of modifying the representations of processes and adding information in order to overcome these difficulties. In particular, we will consider the addition of "pseudo variables" to a representation of a process to make possible stronger assertions about it. We will consider the application of variations of Manna's results involving termination of non-deterministic algorithms [1968] to prove certain properties of cooperating processes.

The difficult part, it seems to this author at least, of verifying any collection of processes is not the proof but the construction of a suitable set of criteria. Once the behavior of the system has been suitably characterized by a set of assertions or otherwise - a proof is relatively straightforward. It may be lengthy, especially if a complete, formal proof is desired. But it is not conceptually difficult. Frequently it depends only upon the theorems and axioms which the programmer had in mind when he designed the algorithms (see Hoare [1968] and Dijkstra [1970]). For example, the verification of

the algorithm of Chapter III involved only a few equalities and inequalities of the theory of integers modulo N. The deductive reasoning was simple, although tedious. Most of the examples which follow have the same property, so it is natural that we concentrate on discovering what to prove rather than on the details of a particular verification.

Adding Variables to Processes

Consider the simple producer-consumer example from the beginning of the previous chapter. Figures III.1(b) and III.1(c) show some assertions about the values of the two semaphores governing the cooperation between these processes. The reader can convince himself intuitively that each of these processes is correct in the context of their product with respect to those assertions. However, the Induction Theorem provides no help in proving it because the assertions on node 1 of each component process are too weak.* Of both processes, it is asserted on node 1 that

$$(1) \quad [N-1 \leq E+F \leq N] \wedge E \geq 0 \wedge F \geq 0.$$

In order for the techniques of the previous chapter to apply, we must be able to prove that the operation described by arc $\langle 1,2 \rangle$ of, say, the producer preserves the invariance of (1) on node 2 of the consumer. That is, we must be able to prove that

*It might be suggested that they are too weak to be interesting, but it is not our purpose to evaluate the merits of a desire to prove a particular property about a process. We are primarily interested in the effect that cooperation among processes has on proving them to be correct.

$$(2) \quad \vdash [N-1 \leq E+F \leq N] \wedge E \geq 0 \wedge F \geq 0 \wedge E > 0 \supset \cdot\ [N-1 \leq (E-1) + F \leq N] \wedge (E-1) \geq 0 \wedge F \geq 0,$$

the appropriate instance of the last formula of page 63. But (2) is not true, as the case

$$E+F = N-1$$

illustrates. Thus, the Induction Theorem does not apply.

One of the difficulties in this example is that although the assertions of Figure III.1 do correctly characterize this system of two processes, they do not assert under what condition the sum E+F has the value N-1 and under what conditions it has the value N. The author could discover no suitable stronger assertions without introducing extra information. Habermann [1972], in his discussion of this problem, introduced three counters which effectively describe the "progress" of the processes. Then interesting properties of the processes were described in terms of these counters and proved. However, they are not in the information set of either of the processes and could not be identified in an implementation of them.

This suggests that it is both helpful and necessary to add information to a representation of a system of cooperating processes in order to state and prove interesting properties about it. Such information would effectively reflect the conventions that the programmers have agreed upon in order to make communication work. For example, the programmer of the producer knows that the semaphore E reflects the number of empty buffer positions except possibly when the consumer is

operating upon one. Furthermore, this is true independently of the internal structure of the consumer, its method for naming buffer positions, or the particular transition graph which represents it.

In effect, the set of variables and the program counter (i.e., location in the transition graph) do not always form an adequate description of the "state" of a cooperating process for characterizing its properties within a system. By adding extra information elements, which we call pseudo variables, the state can be expanded for descriptive purposes. The counters introduced by Habermann are pseudo variables in this sense, as are the various "coordinates" suggested by Dijkstra [1968a] for measuring the progress of the execution of a program. In Theorem IV.1 we will present a justification for drawing conclusions about a process by proving assertions on its pseudo variables. This can be done provided the pseudo variables are added in such a way as to preserve the nature of the computations of the original process. I.e., they must be completely redundant from an operational point of view.

In order to prove the theorem we will define "augmented process" and prove a lemma about the states in the computations of an augmented process.

Definition: Let $\underline{p} = (P,Y,\Sigma)$ and $\underline{p}' = (P',Y',\Sigma')$ be two processes represented respectively by transition graphs $G = (V,\Gamma,L)$ and $G' = (V,\Gamma,L')$ which differ only by the labels on the arcs. Furthermore, suppose that:

1. Y is a subset of Y' (i.e., Y' is constructed from Y by adding variables),

2. there is a one-one correspondence of initial states of $\underline{p}$ and $\underline{p}'$ such that if $(y'_0,0) \in \Sigma'$ corresponds to $(y_0,0) \in \Sigma$, then $y'_0 \mid Y = y_0$.
3. the difference between the labels of corresponding arcs of the two graphs consists solely of extensions of the operations of P to include assignments to those elements of Y' not in Y (i.e., for each $\langle v,w\rangle \in \Gamma$

   a. $\varphi'_{\langle v,w\rangle}(y') = \varphi_{\langle v,w\rangle}(y' \mid Y)$, and

   b. $(t'_{\langle v,w\rangle}(y') \mid Y) = (t_{\langle v,w\rangle}(y' \mid Y))$

Then the process $\underline{p}'$ is called an augmentation of $\underline{p}$, or more briefly, an augmented process.

Lemma IV.1: There exists a one-one correspondence between the set of computations of a process and the computations of any augmentation of that process. Furthermore, the states of corresponding computations are also in one-one correspondence.

Proof: Suppose that process $\underline{p}' = (P',Y',\Sigma')$ is an augmentation of process $\underline{p} = (P,Y,\Sigma)$, and that

$$C' = \{(y'_0,0), (y'_1,v_1), (y'_2,v_2),\ldots\}$$

is a computation of $\underline{p}'$. By 3a and 3b of the definition above,

$$C = \{(y'_0 \mid Y,0), (y'_1 \mid Y,V_1), (y'_2 \mid Y,v_2),\ldots\}$$

is a computation of C. Conversely, let

$$C = \{(y_0,0),\ (y_1,v_1),\ (y_2,v_2),\ldots\}$$

be any computation of $\underline{p}$. Then by 2, there is a $(y'_0,0) \in \Sigma'$ such that $y_0 = y'_0 \mid Y$. For $i = 1,2,3,\ldots$, define inductively,

$$y'_{i+1} = t'_{\langle v_i, v_{i+1}\rangle}(y'_i).$$

Then by 3a and 3b,

$$C' = \{(y'_0,0),\ (y'_1,v_1),\ (y'_2,v_2),\ldots\}$$

is a computation of $\underline{p}'$ and for each $i = 0,1,2,\ldots$,

$$y_i = y'_i \mid Y.$$

Thus the correspondence is established.

Now consider an assertion A_v associated with node v of the transition graph representing $\underline{p}$. The same assertion can be associated with the corresponding node of a representation of an augmentation of $\underline{p}$ by considering the truth value of $A_v(y' \mid Y)$.

Theorem IV.1: If $\underline{p}'$ is an augmentation of $\underline{p}$, then $\underline{p}$ is correct with respect to A_v if and only if $\underline{p}'$ is.

Proof: By Lemma IV.1, the state (y,v) occurs in a computation of $\underline{p}$ if and only if state (y',v) occurs in a computation of $\underline{p}'$, where $y = y' \mid Y$. Thus $a_v(y)$ is true if and only if $A_v(y' \mid Y)$ is true; and by definition of correctness, the theorem follows.

Note that the assigmment notation is most convenient for applying this theorem. The hypotheses are automatically satisfied if G' is

constructed by adding pseudo variables and assignments to those pseudo variables without changing any other parts of the labels on the transition graph. In particular, none of the condition labels nor any of the assignments to the original variables may change, because these determine the character of the computations of a process or system of processes.

Example

Let us consider once again the producer-consumer example of Figure III.1. We can quickly see how to verify this system with respect to the assertions given. Then we will turn our attention to proving a more interesting property: namely, that the consumer receives values from the buffer in exactly the same order as produced by the producer. The second verification will depend, in part, upon the first one.

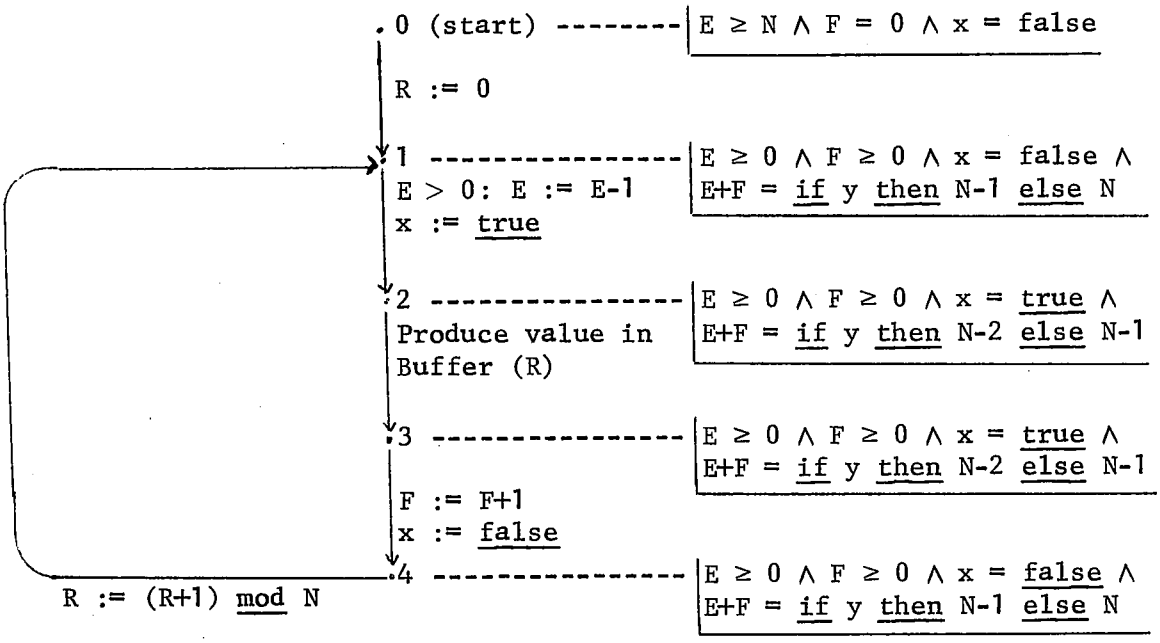
First, let us augment the information set of the producer-consumer system with two Boolean pseudo variables, x and y, both with initial value false. To arc <1,2> of Figure III.1(b) we add the assignment x := true; and to arc <3,4> we add the assignment x := false. Similarly, we add assignments y := true and y := false to arcs <1,2> and <3,4>, respectively, of Figure III.1(c). The modified transition graphs are presented in Figure IV.1 along with some new assertions. (Note the abbreviation

$$E+F = \underline{\text{if}}\ x\ \underline{\text{then}}\ N-1\ \underline{\text{else}}\ n$$

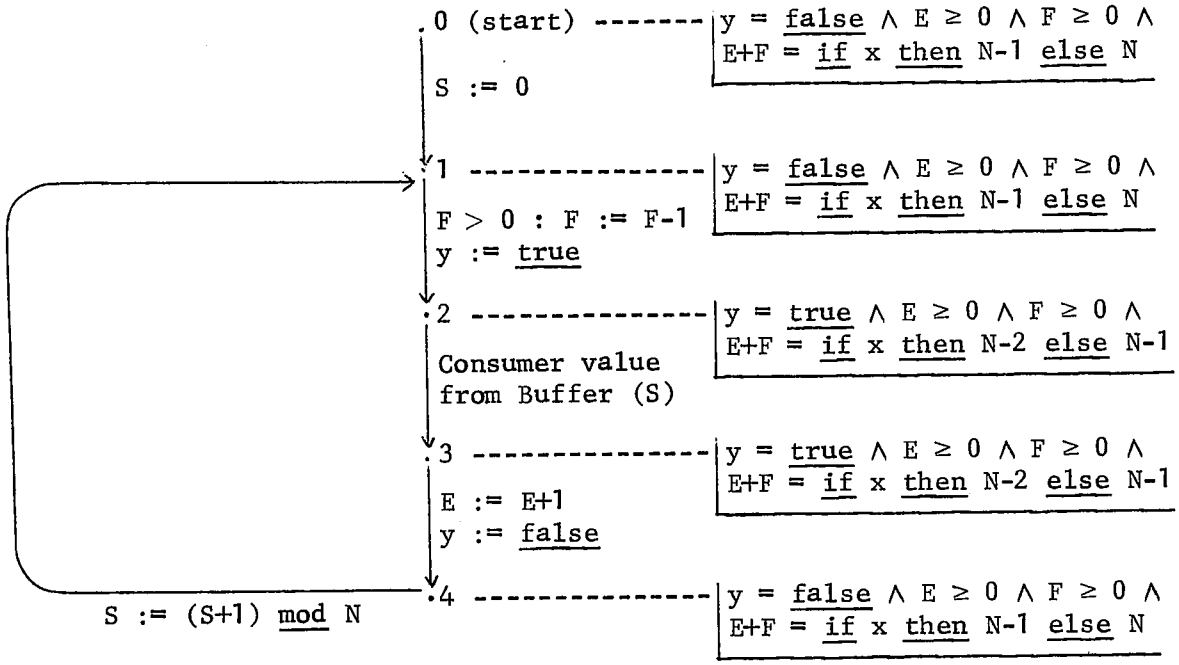
which means

$$[x \supset E+F = N-1] \wedge [\neg x \supset E+F = N].)$$

It is now trivial to prove that each of these two processes is



Augmented Producer Process



Augmented Consumer Process

Figure IV.1

correct with resepct to these new assertions. The details are left to the reader. Furthermore, each of the assertions in Figure IV.1 implies the corresponding assertion of Figure III.1 in the intended interpretation. Thus, we can conclude from the remarks of Chapter III (page 56) that the augmented producer-consumer system is also correct with respect to assertions of Figure III.1. But then, by Theorem IV.1, the original producer-consumer system is also correct with respect to those assertions.

In order to establish the more interesting property that the system maintains a First-In, First-Out discipline, we will need some extra notation as well as some pseudo variables. Let us denote by M_k the $\underline{k}^{th}$ value produced by the producer and by N_k the $\underline{k}^{th}$ value consumed by the consumer. Then the action

"produce value in Buffer(R)"

can be represented by the assignment

$$\text{Buffer(R)} := M_k$$

for an appropriate value of k. Similarly, the action

"consume value in Buffer(S)"

can be represented by the assignment

$$N_k := \text{Buffer(S)}.$$

Let us also introduce a counter $\underline{j}$, analogous to one of Habermann's counters, which counts the number of times the consumer loop is

traversed and, hence, the number of values consumed from the buffer. This pseudo variable, which initially is zero, will be incremented by an assignment $j := j+1$ on arc $\langle 1,2 \rangle$ of the consumer.

The essential assertion (on node 1 of the consumer) with respect to which we will verify the system is

$$\forall k \quad [1 \leq k \leq j \supset M_k = N_k].$$

That is, each of the first j values consumed is equal to the corresponding value produced. In order to facilitate our proof, we will also add the following two assertions to node 1:

$$\forall k \; [-E \leq k < F \wedge j+k \geq 0 \supset \text{Buffer}((S+k) \underline{\text{mod}} \; N) = M_{j+k+1}]$$
$$\wedge \; S = j \underline{\text{mod}} \; N$$

This formula describes the values of the elements of buffer, except for the one element which is being processed by the producer. Assertions on the other nodes of the consumer transition graph can be derived by working backwards from those on node 1. Figure IV.2 illustrates the augmented consumer transition graph with these assertions.

To prove that the augmented consumer is correct with respect to these assertions in the context of the product with a producer process, we must show first, that it is correct when considered alone and, second, that the actions of the producer preserve the invariance of the assertions on the consumer. The first part involves little more than simple substitution and a direct application of the Induction Theorem. The second part is harder. In order to show that the operations of the producer preserve the invariance of the assertions on the consumer, we will

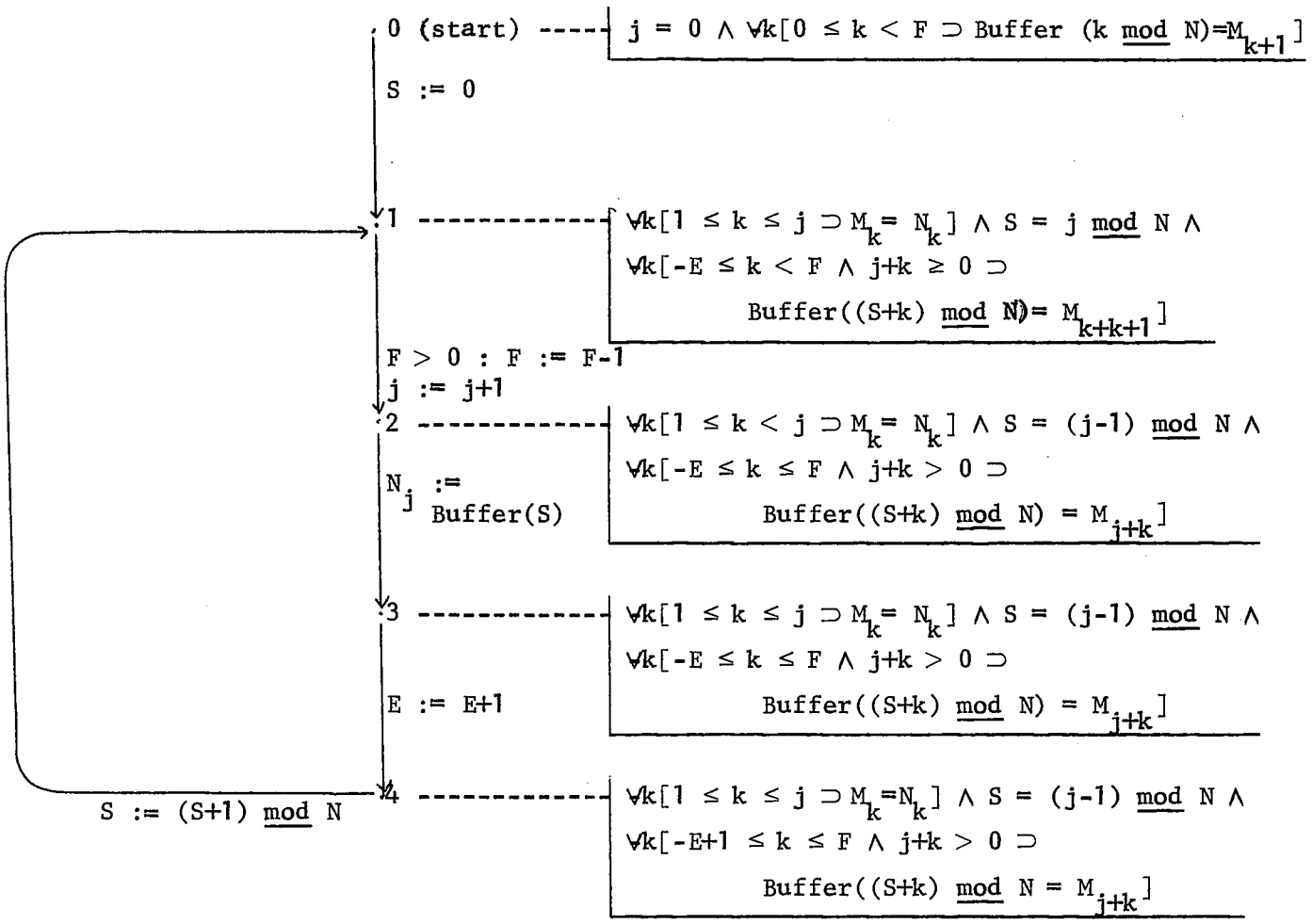


Figure IV.2

identify three properties that the producer should possess. Then we will show that our producer does, indeed, possess these properties.

Observe that the variables S and j are only changed by the consumer, so that no operation of the producer can alter their values. Thus, we need only consider the variables E, F, and Buffer and ensure that producer operations do not change the truth of any assertions on the consumer. The three properties, then, are restrictions on the producer. The first is that the producer operates upon the semaphore E only by decrementing it. We observe that decrementing E can only reduce the

range of an inequality on the left side of an implication, and therefore does not affect the truth of the implication. This holds for each of the assertions on nodes 1, 2, 3, and 4.

The second property is that when the producer operates upon the semaphore F, it does so only by incrementing it by one, and then only when the assertion

$$\text{Buffer}\ ((j+F)\ \underline{\text{mod}}\ N) = M_{j+F+1}$$

is true. Suppose this property holds. Then, in order to preserve the assertion on node 1 of the consumer, we must have

$$\begin{aligned} \vdash\ & \forall k[-E \le k < F \wedge j+k \ge 0 \supset \text{Buffer}((S+k)\ \underline{\text{mod}}\ N) = M_{j+k+1}] \\ & \wedge\ S = j\ \underline{\text{mod}}\ N \wedge \text{Buffer}((j+F)\ \underline{\text{mod}}\ N) = M_{j+F+1} \\ & \supset \forall k[-E \le k < F+1 \wedge j+k \ge 0 \supset \text{Buffer}((S+k)\ \underline{\text{mod}}\ N) = M_{j+k+1}]. \end{aligned}$$

(The right side of this implication is derived from the appropriate part of the left by substituting F+1 for F.) This is obviously true as substitution for the case K = F will show. Similarly, the assertions on nodes 2, 3, and 4 are invariant if

$$\begin{aligned} \vdash\ & \forall k[-E \le k \le F \wedge j+k > 0 \supset \text{Buffer}((S+k)\ \underline{\text{mod}}\ N) = M_{j+k}] \\ & \wedge\ S = (j-1)\ \underline{\text{mod}}\ N \wedge \text{Buffer}((j+F)\ \underline{\text{mod}}\ N) = M_{j+F+1} \\ & \supset \forall k[-E \le k \le F \wedge j+k > 0 \supset \text{Buffer}((S+k)\ \underline{\text{mod}}\ N) = M_{j+k}] \end{aligned}$$

(For node 4, read "-E+1" in place of "-E".) This is true for the same reason as the previous one. The invariance of the assertion on node zero is trivial. Thus, an assignment to the semaphore F, made under the restrictions of the second property, preserves the invariance of the assertions on the augmented consumer.

The third propety of the producer process is that it should not make an assignment to a buffer element mentioned in any assertion on the augmented consumer. One way to guarantee this is if the producer operates only upon the element

$$\text{Buffer}((j+F) \bmod N)$$

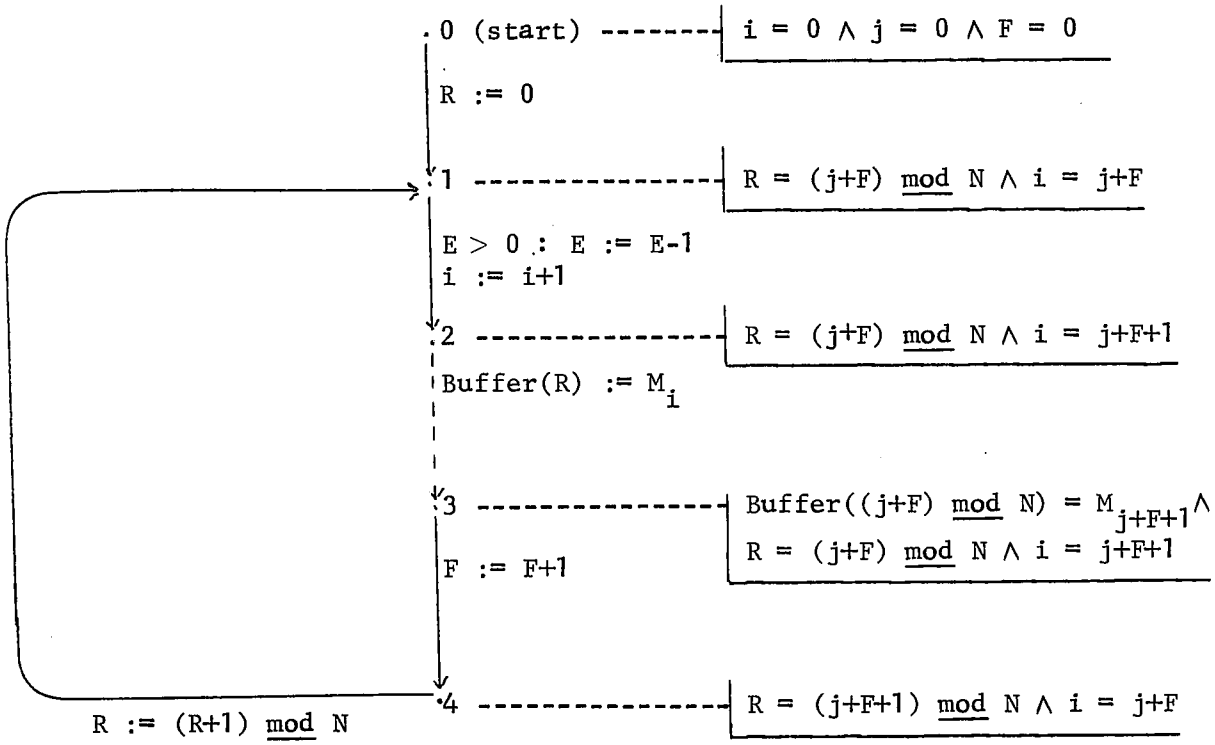
and then only when the relation $N-2 \leq E+F \leq N-1$ is true. This is because under this relation, each assertion specifies values for only (a subset of) the buffer elements of the ring buffer, namely

$$\text{Buffer}((j+F+1) \bmod N),\ldots,\text{Buffer}((j+F-1) \bmod N).$$

Thus an assignment to Buffer((j+F) mod N), which is not amongst these, does not affect the truth of the assertion.

Provided that the producer's operations on the buffer and the semaphores have these three properties, the invariance of the assertions on the consumer is preserved. This is because the three properties account for all of the variables upon which the producer can operate. We have used informal but vigorous arguments to show this. At the expense of clarity, they could be translated into formal, but tedious, arguments which show the same thing. (This seems to be a characteristic of most program verification efforts - that the verification becomes extremely long and tedious, relative to program size, if it carried out at an appropriate level of detail. Hopefully, the results of current and future research will provide insight to help us reduce this complexity and tedium.)

In order to show that our producer process has the three properties we have just presented, we will augment it with a counter i which counts the number of values produced. Figure IV.3 illustrates the augmented process, along with some useful assertions. To prove that this process is correct with respect to these assertions is trivial. It is clearly correct when considered alone. When considered in the context of the augmented producer-consumer product, the only variables which are changed by the augmented consumer and which also appear in these assertions are j and F. But they always appear as a sum, j+F. The only operation performed by the consumer on these is to decrement F by one and simultaneously increment j by one. Thus the sum remains invariant under the action of the augmented consumer, and hence, so do the assertions on the augmented producer. Thus, the producer process is correct with respect to these assertions in the context of the product.



Augmented Producer Process

Figure IV.3

That the first of the three properties holds is evident from Figure IV.3, since E only appears in an assignment representing a P-operation, i.e., which reduces its value. The second property holds because the assertion

$$\text{Buffer}((j+F) \underline{\text{mod}}\ N) = M_{j+F+1}$$

is true whenever the assignment F := F+1 is executed. The third property holds because when the producer executes the assignment

$$\text{Buffer}(R) := M_i,$$

R = (j+F) mod N and i = j+F+1. Furthermore, the relation $N-2 \leq E+F \leq N-1$ holds, by our verification of Figure III.1 and Theorem IV.1.

Thus the augmented producer process has all three properties and therefore preserves the invariance of the assertions on the augmented consumer. So we conclude that the latter is correct in the context of the augmented producer-consumer product with respect to those assertions. This implies that each value consumed by the augmented consumer was produced by the augmented producer in the same order. Thus, by Theorem IV.1, we can conclude the same about the original producer-consumer system.

Note that as a corollary of our verification, we have a justification for representing the operation

"produce a value in Buffer(R)"

as a single arc in the transition graph, even though this might be a simplification of a more complex subgraph. This is because we have

shown that both before and after the buffer element is changed, that element is, in effect, a "temporary local variable" of the producer (since it is not mentioned by any test, operation, or assertion of the consumer). By the results of the previous chapter, we can replace a subgraph representing this part of the computation with a single arc. Thus, this verification also applies to producers which are **not instantaneous** in their production. A similar argument applies to the consumer operation

"consume value in Buffer(S)."

One lesson to be drawn from this example is that of the difficulty of proving even the most obvious properties about cooperating processes. The very simple assertions of Figure III.1 could not be proved without introducing additional machinery. The somewhat more interesting assertions of Figure IV.1 required an argument many times as long as the program as well as extra machinery. In effect, the program describing the two processes is a very compact representation of an otherwise very cumbersome theorem.

Weak Termination, Deadlocks, and Illegal States

In the previous section we showed how some properties of a set of cooperating processes could be studied by adding information to the representation of those processes. In this section and the next section, we will see how some other properties can be studied by considering the circumstances under which the processes do or do not terminate. We showed in Lemma IV.1 that there is a one-one correspondence between

the computations of a process and the computations of an augmented version of it. Thus we can study the termination characteristics of either version and infer the same conclusions about the other.

We have noted previously that operating systems programs are often designed not to terminate. That is, after completing some work they loop back and look for more to do. Termination is a special - and sometimes very unusual - occurrence. On the other hand, there are occasions when some processes may stop unexpectedly. For example, the designers of a system may have defined a state to be "illegal" in the sense that if the system ever reaches that state, some catastrophe occurs. Thus, an important property which one would like to prove about such a system is that no computation ever reaches that state. The discussions of the prevention of system deadlocks (Habermann [1969], Havender [1968], Holt [1971a and 1971b], and others) fall into this category. The condition of deadlock is loosely defined as the condition in which each processor of a system is blocked, awaiting action by some other processor. Certain states of the system are identified as potential causes of deadlocks, and algorithms are designed to prevent the system from reaching those states.

Observe that to prove that no computation of a process ever reaches a certain state, it is sufficient to prove the process correct with respect to an assertion which is false for that state. This follows directly from our definition of correctness. It is equivalent to transforming the state into a halting state of the process and then applying the Weak Termination Theorem of Manna [1968] to determine whether or not there is some computation which halts.

We can illustrate this sort of proof of correctness by the following example:

Example

Bruno, Coffman, and Hosken [1972] have considered a generalization of the producer-consumer problem. Instead of a system of two processes, they have considered a system of n processes. The i^{th} process acts as a consumer to the $(i-1)^{th}$ process and as a producer to the $(i+1)^{st}$ process, and the n^{th} process acts as the producer to the first process. The processes are synchronized by a set of integer semaphores $s_1, s_2, \ldots, s_n$, and a set of generalized P-and V-operations defined by:

$P(a,s) \equiv$ <decrement semaphore s by amount a only if the result would be non-negative>

$V(b,s) \equiv$ <increment semaphore s by b>

Figure IV.4 illustrates a program and a transition graph representation of a typical process of this system (for i = n, replace i+1 by 1). The integers a_i and b_i are both positive.

Observe that there is a danger that this system of n processes could reach a deadlock where each process is blocked at its P-operation waiting for another to do a V-operation. Bruno et al have proved that, given suitable initial states, a necessary and sufficient condition to avoid deadlocks is that

$$(3) \quad a_1 * a_2 * \ldots * a_n \leq b_1 * b_2 * \ldots b_n .$$

Their proof is complicated, based on sequences of states and extensions to such sequences. We shall present a different proof: in this section

```
process i:  begin constant integer a_i,b_i;
            L:  P(a_i,s_i);
                <task i>;
            V (b_i,s_i+1);
              <remainder of process i>;
            go to L end
```

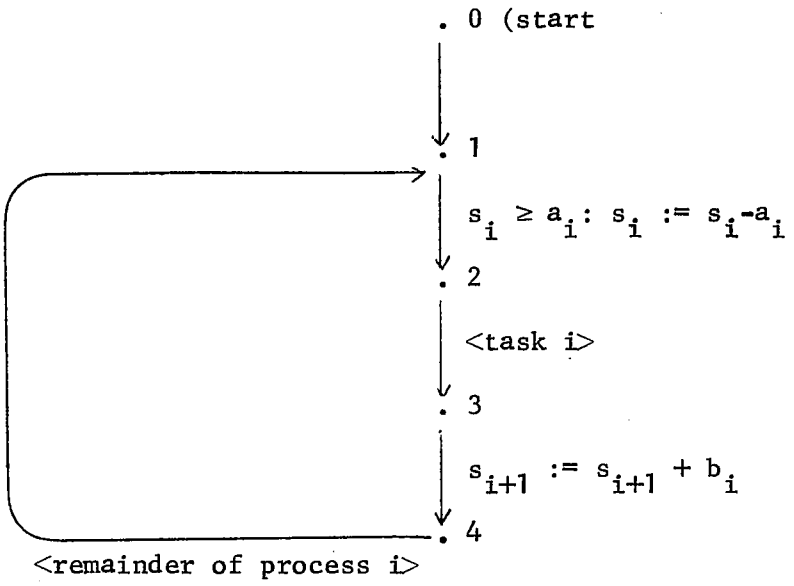


Figure IV.4

we consider the sufficiency of (3); and in a later section, we consider its necessity.

By inspection we see that the only possible way in which the system can deadlock is if each of the component processes is blocked at its P-operation - i.e., at node 1. In the product transition graph, this corresponds to the node

$$(v_1,v_2,\ldots,v_n) = (1,1,\ldots,1).$$

Thus, we need only show that the system is correct with respect to an assertion on this node which is <u>false</u> if all of the s_i are less than the corresponding a_i. In order to state such an assertion, let us add the integer pseudo variables $\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_n$ to the system and two assignments to arc <1,2> of each transition graph, as illustrated in Figure IV.5a. For each i, the initial value of $\bar{s}_i$ is the same as that of s_i. Then it is obvious that the $\underline{i}^{th}$ augmented process is correct with respect to the assertions in the figure. These state that the $(i+1)^{st}$ pseudo variable reflects either the current value of the $(i+1)^{st}$ semaphore or what its value will be as soon as the V-operation is performed. The purpose of making this augmentation is to transform the process into something resembling the idealized process of Figure IV.5b. For the latter process we need only find an assertion for a single node (which we will then use to construct the appropriate assertions for the former).

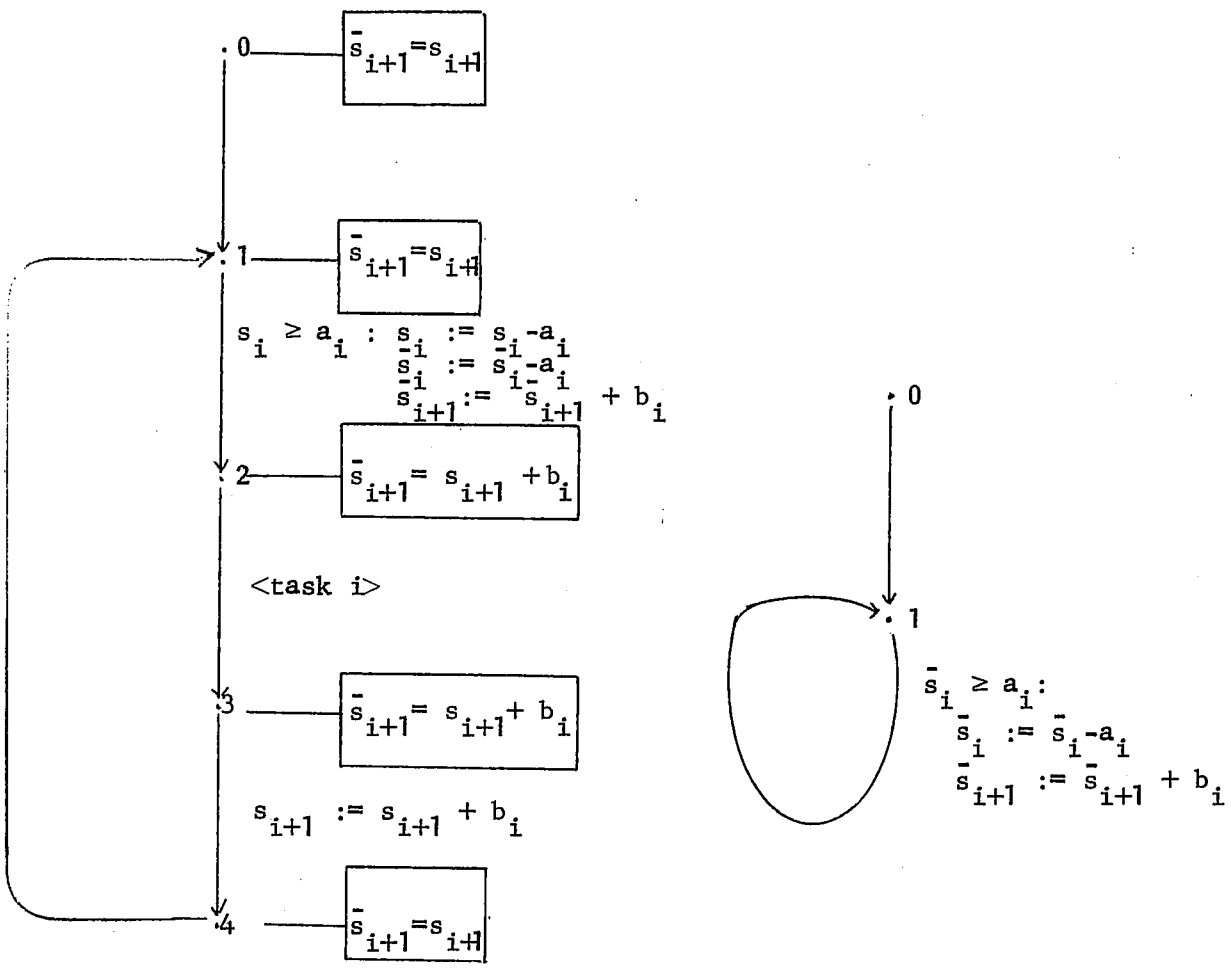
To discover this assertion let $(\bar{s}_1^0, \bar{s}_2^0, \ldots \bar{s}_n^0)$ be some state of the pseudo variables $(\bar{s}_1, \bar{s}_2, \ldots, \bar{s}_n)$, and consider the effect of executing the first idealized process until it becomes blocked. It would cycle approximately $\bar{s}_1^0/a_1$ times and increment $\bar{s}_2$ by approximately

$$\bar{s}_1^0 \frac{b_1}{a_1} .$$

Then execute the second process until it is blocked: it cycles approximately

$$(\bar{s}_2^0 + \bar{s}_1^0 \frac{b_1}{a_1}) \frac{1}{a_2}$$

times and increments $\bar{s}_3$ by approximately



(Note: For i = N, replace "i+1" by "1".)

(a) (b)

Figure IV.5

$$(\bar{s}_2^0 + \bar{s}_1^0 \frac{b_1}{a_1}\frac{b_2}{a_2} = \bar{s}_2^0 \frac{b_2}{a_2} + \bar{s}_1^0 \frac{b_1 b_2}{a_1 a_2}$$

Similarly executing each process in turn until it becomes blocked produces the result that $\bar{s}_2, \bar{s}_3, \ldots, \bar{s}_n$ are all less than the corresponding a's and $\bar{s}_1$ would be approximately equal to

$$\bar{s}_1^0 \frac{b_1 b_2 \ldots b_n}{a_1 a_2 \ldots a_n} + \bar{s}_2^0 \frac{b_2 b_3 \ldots b_n}{a_2 a_3 \ldots a_n} + \bar{s}_n^0 \frac{b_n}{a_n} .$$

If the system is not to deadlock, this must be large enough to allow the above procedure to be repeated indefinitely.

Thus to define a suitable assertion let the function f be defined as

$$f(x_1,x_2,\ldots,x_n) = s_1 \frac{b_1b_2\ldots b_n}{a_1a_2\ldots a_n} + x_2 \frac{b_2b_3\ldots b_n}{a_2a_3\ldots a_n} + \ldots + x_n \frac{b_n}{a_n}.$$

Then define the assertion $A(\bar{s}_1,\bar{s}_2,\ldots,\bar{s}_n)$ to be the relation

$$f(\bar{s}_1,\bar{s}_2,\ldots,\bar{s}_n) > f(a_1\text{-}1,a_2\text{-}1,\ldots,a_n\text{-}1).$$

Clearly this relation is false if for each i, $\bar{s}_i < a_i$. For in this case, we would have $\bar{s}_i \leq a_1\text{-}1$ and thus

$$f(\bar{s}_1,\bar{s}_2,\ldots,\bar{s}_n) \leq f(a_1\text{-}1,a_2\text{-}1,\ldots,a_n\text{-}1)$$

contradicting $A(\bar{s}_1,\bar{s}_2,\ldots,\bar{s}_n)$.

Now we will prove that the product of the n augmented processes of Figure IV.5a is correct with respect to the assertion $A(\bar{s}_1,\bar{s}_2,\ldots\bar{s}_n)$ on every node, for suitable initial states. Suppose that each initial state $(s_1^0,s_2^0,\ldots,s_n^0)$ is such that

$$A_v(s_1^0,s_2^0,\ldots,s_n^0) = \underline{\text{true}}$$

Then since $\bar{s}_i = s_i$ (initially) for each i = 1,2,...,n, we also have

$$A(\bar{s}_1^0,\bar{s}_2^0,\ldots,\bar{s}_n^0) = \underline{\text{true}}.$$

The only assigmments to the $\bar{s}_i$ occur in arc <1,2> of each component process. Thus we must show for each i = 1,2,...,n-1 and each $(\bar{s}_1,\ldots,\bar{s}_n)$ that

$$A(\bar{s}_1,\ldots,\bar{s}_n) \wedge s_i \geq a_i \supset A(\bar{s}_1,\ldots,\bar{s}_i\text{-}a_i,\bar{s}_{i+1} + b_i,\ldots,\bar{s}_n)$$

and that

$$A(\bar{s}_1,\ldots,\bar{s}_n) \wedge s_n \geq a_n \supset A(\bar{s}_1 + b_n,\ldots,\bar{s}_n - a_n).$$

For $i = 1,2,\ldots,n-1$, observe that

$$f(\bar{s}_1,\ldots,\bar{s}_i - a_i,\bar{s}_{i+1} + b_i,\ldots,\bar{s}_n) =$$

$$f(\bar{s}_1,\ldots,\bar{s}_i,\bar{s}_{i+1},\ldots,\bar{s}_n) - a_i \frac{b_i b_{i+1}\cdots b_n}{a_i a_{i+1}\cdots a_n} + b_i \frac{b_{i+1}\cdots b_n}{a_{i+1}\cdots a_n} =$$

$$f(\bar{s}_1,\ldots,\bar{s}_i,\bar{s}_{i+1},\ldots,\bar{s}_n) - b_i \frac{b_{i+1}\cdots b_n}{a_{i+1}\cdots a_n} + b_i \frac{b_{i+1}\cdots b_n}{a_{i+1}\cdots a_n} =$$

$$f(\bar{s}_1,\ldots,\bar{s}_i,\bar{s}_{i+1},\ldots,\bar{s}_n).$$

Thus, $A(\bar{s}_1,\ldots,\bar{s}_i-a_i,\bar{s}_{i+1} + b_i,\ldots,\bar{s}_n) \equiv A(\bar{s}_1,\ldots,\bar{s}_n)$.

For $i = n$,

$$f(\bar{s}_1 + b_n,\ldots,\bar{s}_n - a_n) =$$

$$f(\bar{s}_1,\ldots,\bar{s}_n) + b_n \frac{b_1 b_2\cdots b_n}{a_1 a_2\cdots a_n} - a_n \frac{b_n}{a_n} =$$

$$f(\bar{s}_1,\ldots,\bar{s}_n) + b_n \frac{b_1 b_2\cdots b_n}{a_1 a_2\cdots a_n} - b_n.$$

By hypothesis, $a_1 a_2\cdots a_n \leq b_1 b_2\cdots b_n$, so that

$$b_n \frac{b_1 b_2\cdots b_n}{a_1 a_2\cdots a_n} \geq b_n.$$

Thus

$$f(\bar{s}_1+b_n,\ldots,\bar{s}_n- a_n) \geq f(\bar{s}_1,\ldots,\bar{s}_n),$$

and therefore

$$A(\bar{s}_1,\ldots,\bar{s}_n) \supset A(\bar{s}_1+b_n,\ldots,\bar{s}_n-a_n).$$

Thus the formula of the Induction Theorem is satisfied and the system of augmented processes is correct with respect to the assertion

$A(\bar{s}_1,\ldots\bar{s}_n)$ on every node.

Finally to conclude that the system does not deadlock, consider node (1,1,...1) of the product transition graph. We have already seen that the $\underline{i}^{th}$ augment process is correct (in the context of the product) with respect to the assertion

$$\bar{s}_{i+1} = s_{i+1} \quad (\text{for } i = 1,2,\ldots,n-1)$$

or

$$\bar{s}_1 = s_1 \quad (\text{for } i = n).$$

Therefore, $A(\bar{s}_1,\bar{s}_2,\ldots,\bar{s}_n) = A(s_1,s_2,\ldots,s_n)$ on node (1,1,...1), and thus the product process is correct with respect to the assertion $A(s_1,s_2,\ldots,s_n)$ on this node. But this assertion is false if $s_i < a_i$ for all $i = 1,2,\ldots,n$. Thus, the product process never reaches such a state and hence the system never deadlocks.

Comment

This example illustrates both some strengths and weaknesses of using assertions to prove properties of cooperating processes. On the positive side, we have been able to give a conceptually simple proof that the two conditions

(1) $a_1a_2\ldots a_n \leq b_1b_2\ldots b_n$, and

(2) the initial state $(s_1^0,s_2^0,\ldots s_n^0)$ satisfies

$$\sum_{i=1}^{n} s_i^0 \frac{b_ib_{i+1}\ldots b_n}{a_ia_{i+1}\ldots a_n} > \sum_{i=1}^{n} (a_i-1) \frac{b_ib_{i+1}\ldots b_n}{a_ia_{i+1}\ldots a_n}$$

are sufficient to guarantee that the system of $\underline{n}$ processes do not deadlock.

After performing appropriate transformations and simplifications, our proof consisted solely of showing that a certain inequality is preserved. It completely avoids the notions of execution sequences which were inherent in the original proof.

On the negative side, however, our result is weaker than the result obtained by Bruno et al because our assertion is not strong enough. Consider the following case for n = 3. Let

$$a_1 = a_2 = a_3 = b_1 = b_2$$

and let $b_3 = a_1+3$. Let an initial state be

$$s_1^0 = a_1-3$$
$$s_2^0 = a_2-1$$
$$s_3^0 = a_3$$

Then $f(s_1^0,s_2^0,s_3^0) =$

$$s_1^0 \frac{b_1b_2b_3}{a_1a_2a_3} + s_2^0 \frac{b_2b_3}{a_2a_3} + s_3^0 \frac{b_3}{a_3} =$$

$$(a_1-3) \frac{b_1b_2b_3}{a_1a_2a_3} + (a_2-1) \frac{b_2b_3}{a_2a_3} + a_3 \frac{b_3}{a_3} =$$

$$(a_1-1) \frac{b_1b_2b_3}{a_1a_2a_3} + (a_2-1) \frac{b_2b_3}{a_2a_3} + (a_3-2) \frac{b_3}{a_3} <$$

$$f(a_1-1,a_2-1,a_3-1),$$

since $\frac{b_1b_2}{a_1a_2} = 1.$

That is, the initial state does not satisfy our assertion A. But the system is not blocked in this state, since $s_3^0 \geq a_3$. Furthermore, by the

test of Bruno et al, the system will not reach a deadlock from this initial state - something which is apparent after process 3 executes its cycle just once. For then, the new values of the semaphores are

$$s_1 = s_1^0 + b_3 = (a_1-2) + (a_1+3) \geq a_1$$

$$s_2 = s_2^0 = a_2-1$$

$$s_3 = s_3^0-a_3 = 0$$

and $f(s_1,s_2,s_3) =$

$$(2a_1+1)\ \frac{b_1b_2b_3}{a_1a_2a_3} + (a_2-1)\ \frac{b_2b_3}{a_2a_3} =$$

$$(a_1+1)\ \frac{b_1b_2b_3}{a_1a_2a_3} + (a_2-1)\ \frac{b_2b_3}{a_2a_3} + a_3\ \frac{b_3}{a_3} =$$

$$f(a_1+1,a_2-1,a_3)$$

since $a_1 = a_3$ and $\frac{b_1b_2}{a_1a_2} = 1$. But this is greater than $f(a_1-1,a_2-1,a_3-1)$ so that the assertion A is satisfied and deadlock cannot occur.

The difficulty is that we have not provided strong enough assertions to completely characterize the system. This seems to be characteristic of the problems to which the author has tried to apply these methods - that the using assertions to describe the cooperation among processes, even in very idealized cases, is extremely difficult. In this particular problem, it took over a week of concentrated effort and many, many discussions with colleagues to discover an assertion which was both provable and which guaranteed no deadlock. The candidate assertions which were discarded were either too weak to imply that deadlock does not occur or too hard to prove correct (in fact, most of the latter were probably not

true). It was only several weeks later that the author discovered assertion A which allowed the example to be simplified to its present form.

Furthermore, the difficulties of discovering an assertion were not caused by the representation of the problem or the model of computation which we have used. They were inherent in the problem itself - that is, it was difficult to characterize precisely the states of the system which could arise from the cooperating of the processes of Figure IV.5b. If this problem is at all representative of the general case, we would expect that even if our methods are extended to permit verification of reasonable large system of cooperating processes in a reasonable amount of time, discovering and stating the properties to prove will still be a difficult problem. This is certainly a subject for continued research.

Strong Termination and Home States

We have seen that we can prove some properties of a system of cooperating processes by proving that no computation reaches a certain state. A complementary question is whether or not we can verify interesting properties of a system by showing that every computation reaches a particular state. In the remainder of this chapter we will briefly consider some variations of the Strong Termination Theorem of Manna [1968] for the purpose of proving such properties. We will see that, in principle, they can be proved by transforming the verification problem into a halting problem. Then the theorem provides a necessary and sufficient condition that every computation of the process "halts" at one of the identified states. However, there are still practical problems in the application

of this theorem, and these require more investigation. But we will see an example to illustrate its use.

For the purpose of defining the notion of "strong termination", it is useful to define a complete computation of a process as a computation which is not also an initial subsequence of another computation of that process (see Berry [1972]).

Definition: Let $p = (P,Y,\Sigma)$ be a process represented by a transition graph $G = (V,\Gamma,L)$ and let h be a node of G designated as the halt node. The p terminates strongly at h if and only if every complete computation of p is finite and ends at h.

That is, p terminates strongly if all finite computations which do not terminate at h can be extended to finite computations which do terminate at h.

Strong Termination Theorem (Manna [1968a]): Suppose the process p has no blocking states (other than, of course, its termination state at node h). Then p terminates strongly at node h if and only if there is no set of assertions $\{A_v\}_{v\in V}$ which satisfies the following formula in the intended interpretation:

$$(5)\quad \exists y[y \in \Sigma \wedge A_0(y)] \wedge \forall y[\neg A_h(y) \wedge \bigwedge_{v\in V} \cdot A_v(y) \supset \bigvee_{\langle v,w\rangle\in\Gamma} \cdot \varphi_{\langle v,w\rangle}(y) \wedge A_w(t_{\langle v,w\rangle}(y))].$$

The restriction that there be no blocking states can be relaxed by including in the disjunction the following term:

$$\bigwedge_{\langle v,w\rangle\in\Gamma} \neg\, \varphi_{\langle v,w\rangle}(y)$$

The principal implication in formula (5) then becomes

$$(5)\quad A_v(y) \supset \cdot \bigvee_{\langle v,w\rangle\in\Gamma} [\varphi_{\langle v,w\rangle}(y) \wedge A_w(t_{\langle v,w\rangle}(y))] \vee [\bigwedge_{\langle v,w\rangle\in\Gamma} \neg\, \varphi_{\langle v,w\rangle}(y)],$$

and it is repeated for each node v of the transition graph. The proof of this variation is nearly identical to Manna's proof of the original theorem, and so it will not be given here. Note that this theorem easily extends to apply to cases with more than one halt node, simply by requiring the assertion on every halt node to be identically <u>false</u>.

Now let us turn our attention again to problems related to deadlocks. Although we can sometimes associate a deadlock condition with a particular state as in previous sections, other times we cannot. That is, it may be inconvenient or impossible to identify a particular set of states and show that it is the cause of the system becoming deadlocked or a process remaining blocked indefinitely or some decision being postponed indefinitely. However, it might be possible to show instead that every computation of a system eventually reaches a state which can be regarded as "safe" in some sense. Then we could conclude that the deadlock and indefinite blocking and postponement cannot occur.

For a process which is designed to halt, an obvious "safe" state is its halting state - if it reaches that one, it could not have been delayed forever in some other state. But a cyclic process is not designed to halt. Thus, to apply this method we must look for other "safe" states. One possibility is illustrated by Figure IV.6: in a transition graph representing a cyclic process, one node - in this case v* - is designated

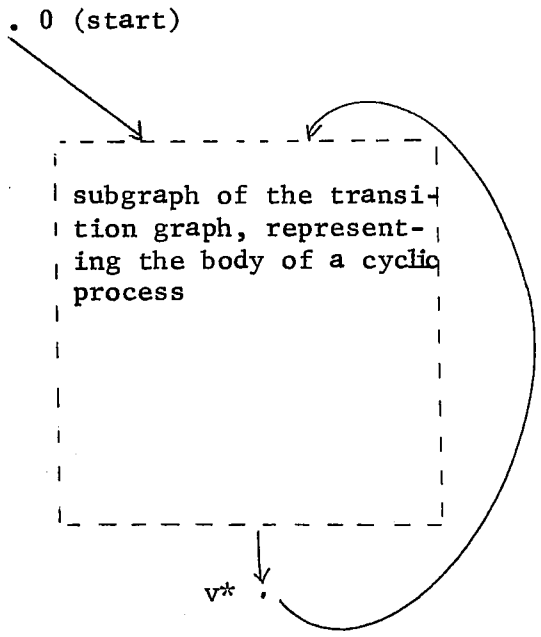


Figure IV.6

as the "home" state. That is, it represents the completion of some unit of work or some interaction with other processes, so it can be regarded as "safe". Thus, for a cyclic process, we break the cycle and consider the question of whether or not every computation which begins a cycle eventually completes it. In either case, it is a direct application of the Strong Termination Theorem to verify that this is true, and hence that deadlocks or indefinite delays do not occur.

The kind of processes of interest to us are those which are products of other processes and represented by product transition graphs. The verification method just outlined can be extended to apply to this kind of problem, as well. Suppose, for example, we wish to show that the j^{th} component process of a system always completes a cycle which it begins. We break its cycle at a node v_j^* representing a home state; then in the product transition graph, we designate as a halting state every node of which v_j^* is a component. That is, every node of the form

$$v^* = (v_1,\ldots,v_j^*,\ldots v_n)$$

is treated as a halt node. Then the Strong Termination Theorem is applied to the product transition graph to show that every computation of the system reaches one of these states, and hence that the component process in question completes its cycle.

In practice, applying the Strong Termination Theorem this way can lead to difficulties. Consider the system composed of N copies of the process of Figure III.2. To show that each process eventually completes its cycle, we would designate node 8 of the transition graph as the home state and apply the Strong Termination Theorem in the way we have described. Unfortunately, formula (5) is satisfiable - in fact, it is not difficult to show that the assertions of Table III.1 satisfy it. The problem is that there are computations in which some processors execute forever in infinite loops while others never execute at all.

In part, this problem arises from the model of computation which we are using. We have deliberately chosen the model to exclude information about relative speeds of the processors in a system and about hidden effects of one processor on the progress of another. This was done to provide an orderly framework for considering the product process. As a result, we do not have a convenient means of representing a very important piece of information about the system of Figure III.2: namely that every processor eventually executes its next operation if it is not blocked. Similarly, we do not have a convenient way of representing the kinds of information embodied in various implementations of synchronizing operations (see Habermann [1972]) such as:

> a process in a blocked state, waiting for some event, is unblocked and moved to its next state the instant that event is signalled.

This is a topic for further investigation. Either the model or the representations of systems of processes could be extended so that this kind of information can be incorporated. Alternatively, the Strong Termination Theorem itself could be extended to consider only the types of computations of interest. Another possibility is the addition of suitable axioms to the intended interpretation. A related topic for investigation is that of the usefulness and characteristics of the class of algorithms which do fit within our model. Dijkstra [1965a] and Courtois et al [1971] have shown interesting algorithms which do not assume any particular queueing discipline in the synchronization primitives but which do assume that signalling an event and unblocking a waiting process are together an indivisible action. Is there merit in studying the class of cooperating processes which do not even make the latter assumption?

Strong Termination and Finite Computations

Another variation of the Strong Termination Theorem provides an illustration of an application to a problem of cooperating processes, namely, the n-process system of Bruno et al considered earlier in this chapter. We have already shown that given suitable initial states, a sufficient condition for the system to avoid deadlock is that

$$a_1 a_2 \ldots a_n \leq b_1 b_2 \ldots b_n .$$

In this section we will show the condition is necessary in a very strong

way by showing that every computation of the system is finite (i.e., eventually reaches a deadlock) if the condition is not satisfied, independently of the initial state.

To do this, we will use the following version of the Strong Termination Theorem.

Corollary: Let $\underline{p} = (P,Y,\Sigma)$ be a non-deterministic process represented by a transition graph $G = (V,\Gamma,L)$. Then every computation of $\underline{p}$ is finite if and only if there is no set of assertions $\{A_v\}_{v\in V}$ which satisfies the following formula in the intended interpretation:

$$(6)\quad \exists y[(y,0) \in \Sigma \wedge A_0(y)] \wedge \forall y \bigwedge_{v\in V} \cdot A_v(y) \supset \bigvee_{\langle v,w\rangle\in\Gamma} \cdot \varphi_{\langle v,w\rangle}(y) \wedge A_w(t_{\langle v,w\rangle}(y))$$

The proof of this version is also very similar to Manna's original proof.

To apply the theorem we must show that formula (6) is unsatisfiable for the n-way product of the component transition graphs of Figure IV.4. That is, we must show that for every set of assertions on the product graph, there is a state which leads to a contradiction and hence prevents that set of assertions from satisfying (6). For this purpose, we will relabel the nodes of each component transition graph and then impose an ordering on the states of the semaphores and processors. Figure IV.7 shows the relabelling; note that the numbers labelling the nodes decrease in the direction of the arrows, except for the arc representing the P-operation.

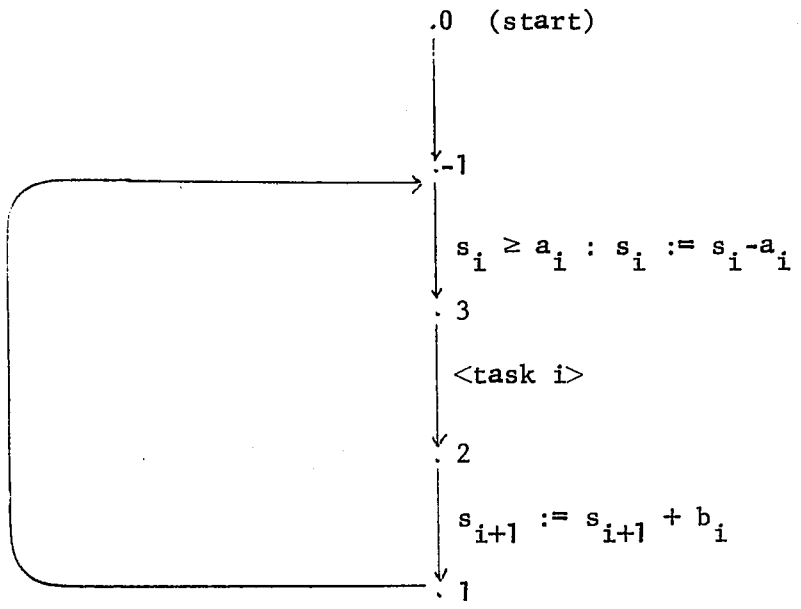


Figure IV.7

Now consider a state of the system of n processes as represented by the 2n-tuple

$$(s_1, s_2, \ldots, s_n, v_1, v_2, \ldots, v_n).$$

Form the sum $g(s_1, \ldots, s_n, v_1, \ldots, v_n)$ as follows:

for $i = 1,2,\ldots,n-1$: if $v_i = 2$ or 3, let

$\bar{s}_{i+1} = s_{i+1} + b_i$; otherwise let $\bar{s}_{i+1} = s_{i+1}$.

for $i = n$, if $v_n = 2$ or 3, let $\bar{s}_1 = s_1 + b_n$;

otherwise let $\bar{s}_1 = s_1$.

let $g(s_1, \ldots, s_n, v_1, \ldots, v_n) =$

$$\sum_{i=1}^{n} \bar{s}_1 \frac{b_1 b_{1+1} \cdots b_n}{a_i a_{i+1} \cdots a_n}$$

That is, $g(s_1, \ldots, s_n, v_1, \ldots, v_n)$ is defined in a very similar way to $f(s_1, \ldots, s_n)$ previously. Now to define an order relationship on states, we will say that

$$(s_1,\ldots,s_n,\ v_1,\ldots,v_n) < (s'_1,\ldots s'_n,\ v'_1,\ldots,v'_n)$$

if and only if either

$$g(s_1,\ldots s_n,\ v_1,\ldots,v_n) < g(s'_1,\ldots,s'_n,\ v'_1,\ldots,v'_n)$$

or

$$g_1(s_1,\ldots,s_n,\ v_1,\ldots,v_n) = g(s'_1,\ldots,s'_n,\ v'_1,\ldots v'_n)$$

and

$$(\bar{s}_1,\ldots,\bar{s}_n,\ v_1,\ldots,v_n) < (\bar{s}'_1,\ldots,\bar{s}'_n,\ v'_1,\ldots,v'_n)$$

in the lexicographic ordering. I.e., one state is less than another if and only if the sum constructed from the first is less than that of the second; or in the case the two sums are equal, if and only if some $\bar{s}_i < \bar{s}'_i$ and $\bar{s}_j = \bar{s}'_j$ for $1 \le j < i$; or in the case that all corresponding $\bar{s}$ are all equal, if and only if some $v_i < v'_i$, and for $1 \le j < i$, $v_j = v'_j$. For convenience, we will use v to denote the n-tuple $(v_1,\ldots,v_n)$.

Now let $\{A_v(s_1,\ldots,s_n)\}_{v\in V}$ be any set of assertions on the product transition graph. There are two principle cases to consider:

Case 1: $A_0(s_1^0,\ldots,s_n^0) \equiv$ false for every initial state $(s_1^0,\ldots,s_n^0)$ of the semaphores. Then the first term of formula (6) is not satisfied, and hence the whole formula is not satisfied.

Case 2: There exists an initial state $(s_1^0,\ldots,s_n^0)$ such that $A_0(s_1^0,s_2^0,\ldots,s_n^0) =$ true. Let

$$m = \min(0,s_1^0,s_2^0,\ldots,s_n^0)$$

and let $(s_1,s_2,\ldots,s_n,\ v_1,\ldots,v_n)$ be the smallest state of the system in our ordering such that for $1 \le i \le n$, $s_i \ge m$ and

$$A_v(s_1,\ldots,s_n) = \underline{\text{true}}.$$

(Such a state exists because our ordering relationship is also a well-ordering relationship for a set of semaphores bounded below by m and because the set of states with a true assertion is not empty.) We will show that the set of assertions does not satisfy

$$A_v(s_1,\ldots s_n) \supset \bigvee_{\langle v,w\rangle \in \Gamma} \cdot \varphi_{\langle v,w\rangle}(s_1,\ldots s_n) \wedge A_w(t_{<,w>}(s_1,\ldots s_n))$$

Since the transition graph is a product of n identical components, we need only consider the possible terms which the i^{th} component contributes to the disjunction:

Case a: $v_i = 0$, $w_i = -1$. Then $t_{\langle v,w\rangle}(s_1,\ldots s_n) = (s_1,\ldots,s_n)$ (i.e., the identity operation) but $v = (v_1,\ldots v_n)$ is lexicographically less than $w = (v_1,\ldots,v_{i-1},w_i,v_{i+1},\ldots v_n)$. Thus by assumption: $A_w(s_1,\ldots,s_n) = \underline{\text{false}}$, and hence

$$A_v \supset A_w(t_{\langle v,w\rangle}(s_1,\ldots s_n))$$

is false for this case.

Case b: $v_i = 3$, $w_i = 2$. This case is identical to case a.

Case c: $v_i = 1$, $w_i = -1$. This case is also identical to case a.

Case d: $v_i = 2$, $w_i = 1$. For this case the operation $t_{\langle v,w\rangle}(s_1,\ldots,s_n) = (s_1,\ldots,s_{i+1} + b_i,\ldots s_n)$ (for i = n, substitute 1 for i+1. But by our ordering relationship

$$(s_1,\ldots,s_n,\ v_1,\ldots v_n) < (s_1,\ldots s_{i+1} + b_i,\ldots,s_n,\ v_1,\ldots,\ w_i,\ldots v_n)$$

since all $\bar{s}_i$ are equal and $w_i < v_i$. Thus $A_w(t_{<v,w>}(s_1,\ldots s_n)) =$ false for this case.

Case e: $i \neq n$, $v_i = -1$, $w_i = 3$. Then the condition label $\varphi_{<v,w>}(s_1,\ldots s_n)$ is the predicate "$s_i \geq a_i$", and the operation is

$$t_{<v,w>}(s_1,\ldots s_n) = (s_1,\ldots s_i - a_i,\ldots s_n).$$

Since $g(s_1,\ldots s_n,\ v_1,\ldots v_n) =$

$$\sum_{i=1}^{n} s_i \ \frac{b_i\ldots b_n}{a_i\ldots a_n} =$$

$$\sum_{i=1}^{n} s_i \ \frac{b_i\ldots b_n}{a_i\ldots a_n} - a_i \ \frac{b_i\ldots b_n}{a_i\ldots a_n} + b_i \ \frac{b_{i+1}\ldots b_n}{a_{i+1}\ldots a_n} =$$

$$g(s_1,\ldots s_i - a_i \ldots s_n,\ v_1,\ldots,w_i,\ldots v_n),$$

we see that $(s_1,\ldots s_n,\ v_1,\ldots,v_n)$ is lexicographically greater than $(s_1,\ldots,s_i - a_i,\ldots s_n,\ v_1,\ldots,w_i,\ldots v_n)$ by virtue of

$$s_i - a_i < s_i,$$

Either the condition label $s_i \geq a_i$ is false, in which case

$$A_v(s_1,\ldots,s_n) \supset s_i \geq a_i \wedge A_w(t_{<v,w>}(s_1,\ldots s_n))$$

is false; or it is true, in which case

$\min(0, s_1,\ldots s_i - a_i,\ldots,s_n) \geq m$, guaranteeing that

$A_w(t_{<v,w>}s_1,\ldots s_n))$ is false by assumption, and hence that

$$A_v(s_1,\ldots s_n) \supset s_i \geq a_i \wedge A_w(t_{<v,w>}(s_1,\ldots,s_n))$$

is false.

Case f: $i = n$, $v_i = -1$, $w_i = 3$. This is the principle case and the only one to which the condition on the a's and b's is relevant. If $a_1a_2\ldots a_n > b_1b_2\ldots b_n$, then

$$g(s_1,\ldots,s_n, v_1,\ldots,v_n) =$$

$$\sum_{i=1}^{n} s_i \frac{b_i\ldots b_n}{a_i\ldots a_n} >$$

$$\sum_{i=1}^{n} s_i \frac{b_i\ldots b_n}{a_i\ldots a_n} - a_n \frac{b_n}{a_n} + b_n \frac{b_1\ldots b_n}{a_1\ldots a_n} =$$

$g(s_1,\ldots,s_n-a_n, v_1,\ldots,w_n)$. Thus be an argument similar to that of case e,

$$A_v(s_1,\ldots s_n) \supset s_n \geq a_n \wedge A_w(t_{<v,w>}(s_1,\ldots,s_n))$$

is false.

Since there is an arc representing an operation of the i^{th} process emanating from every node of the product transition graph, since we have considered every possible operation of the i^{th} component, and since all component processes are identical, we can conclude that

$$A_v(s_1,\ldots,s_n) \supset \bigvee_{\langle v,w\rangle \in \Gamma} \cdot \varphi_{\langle v,w\rangle}(s_1,\ldots s_n) \wedge$$

$$A_w(t_{\langle v,w\rangle}(s_1,\ldots s_n))$$

is false. Thus the set of assertions $\{A_v\}_{v\in V}$ does not satisfy (6) in the intended interpretation, and by the Strong Termination Theorem we conclude that every computation of the system is finite. That is, if

$$a_1a_2\ldots a_n > b_1b_2\ldots b_n,$$

then every computation of the system, no matter what initial state is chosen, will eventually reach a deadlock condition.

This completes our analysis of this problem and our illustration of an application of the Strong Termination Theorem to a system of co-operating processes. This verification, though tedious in detail, is conceptually a very simple one and yields a very powerful result.

CHAPTER V

ABSTRACTIONS OF PROCESSES

An important technique used in the design and analysis of programs (and, in particular, programs of operating systems) is that of forming "abstractions". That is, a program is mapped from the detailed environment of its implementation (i.e., its processor and information set) into less detailed and perhaps less restrictive environments in order to facilitate understanding of its various parts. The mappings are chosen to reduce the complexity of a design and description, to expose its essential features and subvert unessential ones, to eliminate or minimize the effect of some inherent non-determinism, and/or to impose order and structure on the processes and informations sets. Such use of abstractions in programming and system design has been discussed in the literature by Dijkstra [1970] and [1972], Zurcher and Randell [1968], Snowden [1971], and many others, and will not be considered here.

The basic idea of abstraction is, however, very important to our methods for verifying properties of cooperating processes. In view of the examples of the previous chapter, it is evident that such verification is difficult, even when the processes contain no extraneous detail. When we consider real programs, the effort would be greatly complicated by constraints of the design and implementation of the system which are not essential to proving the properties of interest. For example, the producer and consumer of Figure III.1 would probably not be implemented as simple loops but as basic building blocks or "atomic" objects within more complicated programs. However, we can see that repeated execution

of these objects would have the same effect as repeated traversals of our simple loop structure, at least with respect to the properties we proved. That is, the simple loops are abstractions of reality and are useful for analyzing certain characteristics of a system.

To actually infer that the system itself has those characteristics requires additional justification. That is, we must be able to show that proving properties about the abstract system allows us to draw similar conclusions about the real system. We have already done this for a simple kind of abstraction in Chapter III. There, we showed that under certain conditions, a subgraph of a transition graph may be replaced by a single arc so that subsequences of operations in a computation are replaced by single operations. That is, we abstracted a detailed part of a computation to a single operation with the same net effect, and we showed that to prove correctness of the original process it was sufficient to prove the correctness of the abstraction.

A general notion of abstraction ought to allow us to draw the same kinds of conclusions in more general cases, particularly when the details of cooperation among processes are being abstracted. This requires a precise definition of what is meant by "abstraction" and an effective method of determining whether one process is, indeed, an abstraction of another. To be useful in a discussion of cooperating processes, the definition must also provide a vehicle for considering the relationship between the abstraction of a product of processes and the product of abstractions of those processes. This would, in certain circumstances, permit us to design, analyze, and verify some properties of a system of cooperating processes at an abstract level rather than at a level dominated by other details.

We do not yet have such a precise definition of abstraction. However, in this chapter, we will consider a candidate for one, inspired by the notion of "image process" of Horning and Randell [1972]. Our definition is more restrictive than theirs in that every abstraction of a process is also a process in the sense of Chapter I. We will see that our definition allows us to infer the correctness of a process with respect to an assertion by verifying an abstraction of that process with respect to a corresponding assertion. The results of previous chapters, particularly the application of the Weak and Strong Termination Theorems, will suggest possible methods for determining if one process is an abstraction of another. But the definition still presents problems. As with the definition of Horning and Randell, it preserves neither the determinism-nondeterminism properties nor the termination-nontermination properties of a process. It also leaves unanswered the question about products of abstractions. Thus we present this chapter primarily as a framework for future investigation.

Mapping Information Sets

Our concept of abstraction is based on mapping the values of one information set into the values of another. A particular abstraction will be specified by specifying a function to map the values of an original, "detailed" information set into the values of an "abstract" information set. Then this function leads directly to a natural processor of the abstract set based on the given processor of the original information set. From this definition we will be able to formulate more precisely the problems which we have mentioned above.

Let us begin by considering a process $\underline{p} = (P,Y,\Sigma)$, and let Z be an information set. Consider a function $\underline{f}$ which maps values of Y into values of Z.

Example: Let Y be the set of two integer variables $\underline{a}$ and $\underline{b}$ and let Z be the set consisting of the single variable which can have any rational number as value. Then one such function would be

$$f(a,b) = \frac{a}{b} \text{ if } b \neq 0.$$

$$f(a,0) \text{ undefined.}$$

That is, f maps each value of Y (each pair of integers) into the quotient in Z, if that quotient is defined.

The function $\underline{f}$ may be either total (that is, defined for every value of Y) or partial (not defined on some values, as in the example). In the latter case we write $y \in D(f)$ (read Y is in the domain of f)* if and only if f(y) is defined. We call Z the image of Y under $\underline{f}$.

Given $\underline{f}$, we can define in a natural way a processor Q of Z based on the processor P of Y. Let us ignore for the moment internal states of P. Let y_0 and y_1 be any values of Y such that

$$y_0 \xrightarrow{P} y_1$$

and

$$y_0 \in D(f),\ y_1 \in D(f).$$

Then we define the action in Q

$$f(y_0) \xrightarrow{Q} f(y_1).$$

*Horning and Randell call such a state "observable".

That is, if a certain action is defined by P, the image of that action is defined by Q. More generally, consider any sequence $\{y_0,y_1,\ldots,y_n\}$ of values of Y such that

$$y_0 \xrightarrow{P} y_1,\ y_1 \xrightarrow{P} y_2,\ldots,y_{n-1} \xrightarrow{P} y_n$$

and

$$y_0 \in D(f),\ y_n \in D(f)$$

$$y_i \notin D(f) \text{ for } i = 1,2,\ldots,n-1.$$

Then we define the action

$$f(y_0) \xrightarrow{Q} f(y_n).$$

That is, a sequence of actions of P, such that the intermediate values of Y have no image under f, also defines a single action of Q.

The processor Q defined this way is a sort of image of P viewed through the mapping $\underline{f}$ (and as such, we will call it the <u>image processor</u>). That is, the actions of Q are to the information set Z what the actions of P are to Y.

In the case that the processor P has more than one internal state (i.e., the process $\underline{p} = (P,Y,\Sigma)$ is represented by a transition graph $G = (V,\Gamma,L)$ with more than one node), we can extend these definitions as follows: Let Z be an information set and W a finite set of nodes (W will be the set of nodes of a new transition graph). Let $\underline{f}$ be a function which maps value-node pairs of Y and V into value-node pairs of Z and W - i.e.,

$$\underline{f} : Y \times V \rightarrow Z \times W$$

Then the image processor, Q, is defined as a processor which has W as its set of internal states and which consists of the set of operations

$$f(y_0,v_0) \xrightarrow{Q} f(y_n,v_n)$$

for each computation of P on Y given (y_0,v_0)

$$\{(y_0,v_0),\ (y_1,v_1),\ldots,(y_n,v_n)\}$$

such that

$$(y_0,v_0) \in D(f)$$

$$(y_n,v_n) \in D(f)$$

$$(y_i,v_i) \notin D(f) \quad \text{for } 1 \leq i < n.$$

That is, an operation is included in Q if and only if it is the "image" under f of a finite computation of P on Y and that computation has the property that only its first and last states are in the domain of f. (We will call such a finite computation a basic sequence of states of p.)

We may assume without loss of generality that both P and Q have unique start nodes, labelled 0_v and 0_w respectively. Then we can use the notation $f(\Sigma)$ to denote the set of images of initial states of p - i.e.,

$$f(\Sigma) = \{(a,0_w) \mid (z,0_w) = f(y,0_v) \text{ for some } y \text{ and } (y,0_v) \in \Sigma\}$$

Definition: The process $q = (Q,Z,f(\Sigma))$ is called the abstraction of process $p = (P,Y,\Sigma)$ under the mapping f. Conversely, p is a realization of q. The mapping f is called the abstracting function.

By this definition, an abstraction of a process is a process in the sense of Chapter I. Thus, all of the verification methods of this thesis apply to it without modification.

Our definition of abstraction is based on mapping from the realization to the abstraction, instead of the other way, for several reasons. First, note that one purpose of forming abstractions is to obscure detail in the realization. Thus, one state of the abstract process could correspond to any of several in the realization. A mapping in our direction is a natural way to represent this. Second, the mapping makes it possible to define the image processor in terms of the given one. In the same way, an actual operating system creates the appearance of an abstract processor by simulating it at a less abstract level. Third, we will see in the next section that the computations of the abstract process correspond to a superset of those of the realization. Thus, properties of the abstraction apply automatically to the realization (subject to restrictions noted below). It would be very difficult to achieve these characteristics with a mapping in the other direction.

Computations of Abstractions

An important relationship between a process and an abstraction of it is that the computations of the former map into computations of the latter. To show this, let $\underline{p}$, $\underline{q}$, and $\underline{f}$ be defined as above and consider an arbitrary computation of $\underline{p}$, say

$$C = \{(y_0,0_v),\ (y_1,v_1),\ (y_2,v_2),\ldots\}.$$

Then the sequence

$$C' = \{f(y_0,0_v),\ f(y_1,v_1),\ f(y_2,v_2),\ldots\}$$

is a sequence of value-nodes pairs of $Z \times W$, possibly interleaved with undefined terms. Let C'' be the subsequence of C' consisting of exactly those elements which are defined. That is, let

$$C'' = \{f(y_{i_0},v_{i_0}),\ f(y_{i_1},v_{i_1}),\ f(y_{i_2},v_{i_2}),\ldots\}$$

where for each $j = 0,1,2,\ldots$

$$i_j < i_{j+1}$$

and

$$\{(y_{i_j},v_{i_j}),\ (y_{i_j+1},v_{i_j+1}),\ldots,(y_{i_{(j+1)}},y_{i_{(j+1)}})\}$$

is a basic sequence of states. Then by the definition of the image processor Q, C'' is a computation of Q on Z given $f(y_{i_0},v_{i_0})$. In particular, if $(y_0,0_v) \in D(f)$, then C'' is a computation of $\underline{q}$, and our claim is justified.

To prove a property about the computations of $\underline{p}$ it would be sufficient to prove the "image" of that property about the computations of $\underline{q}$. I.e., suppose $\underline{q}$ is correct with respect to an assertion $A_w(z)$. Consider an assertion $B_v(y)$ defined by

$$B_v(y) \equiv A_w(z) \text{ if } f(y,v) = (z,w)$$

whenever $(y,v) \in D(f)$. If the state (y,v) occurs in a computation of $\underline{p}$, then $(z,w) = f(y,v)$ occurs in a computation of $\underline{q}$. Hence $A_w(z)$ is true, and so is $B_v(y)$. I.e., $\underline{p}$ is correct with respect to $B_v(y)$. Thus to verify a process with respect to some criterion, that criterion could be

translated into an appropriate assertion about an abstraction of the process and verified there. Conversely, if a process is correct with respect to an assertion, then every realization of it is correct with respect to a suitable "realization" of that assertion. That is, it is sufficient to prove some properties in the abstraction rather than the realization.

It is not always true, however, that a property which can be proved about a realization holds in the abstraction (although it does hold for that subset of computations of the abstraction which actually occur as a result of the realization). This is because there may be computations of q which are not images of computations of p, and there may be finite computations of q which are images of infinite computations of p.

Example: (due to J. J. Horning): Let Y consist of two integer variables a and b, let Z consist of a single integer variable, and let P be a processor of Y which has only one internal state and which includes operations of the form

$$(a,b) \xrightarrow{P} (a+b,1-b)$$

for all integers a and b. Define a mapping f by f(a,b) = ab. Then the definition of Q includes the operations

$$0 \xrightarrow{Q} 1 \text{ (an image of say, } (1,0) \xrightarrow{P} (1,1))$$

$$1 \xrightarrow{Q} 0 \text{ (an image of say, } (1,1) \xrightarrow{P} (2,0))$$

and others. Thus one computation of Q on Z given as 0 an initial value is

$$\{0,1,0,1,0,1,\ldots\}$$

Clearly, there is no computation of P on Y which has this as image.

This example illustrates that there are computations of q which are not images of computations of p. Thus, in general, it is not possible to infer the correctness of q (with respect to some assertions) directly from that of p. This example also shows that a deterministic process can have non-deterministic abstractions, since the sequence

$$\{0,1,0,2,0,3,0,\ldots\}$$

is also a computation of q (in fact, it is the image of the computation

$$\{(1,0),\ (2,0),\ (2,1),\ (3,0),\ (3,1),\ldots\}$$

of p). I.e., q has more than one computation given the initial state 0, and hence it is non-deterministic.

Example: Let y consist of two integers, a and b, and let Z consist of a single variable which can have any rational number as value. Define f, as in the first example of this chapter, by

$$f(a,b) = \frac{a}{b}$$

if $b \neq 0$. Let P include the operations

$$(a,b) \xrightarrow{P} (a+b,\ 1-b)$$

$$(a,0) \xrightarrow{P} (a+1,0).$$

Then the computation

$$\{(0,1), (1,0), (2,0), (3,0),...\}$$

of p maps into the finite, incomplete computation

$$\{0\}$$

This example shows that infinite computations of p can map into finite computations of the abstraction. Thus it is not possible, in general, to infer properties regarding termination from an abstraction of a process.

In effect, forming an abstraction of a process loses structure and/or information - in the sense that properties such as determinism or termination may be obscured or altered, and that some properties are not observable in the realization. (E.g., in the example, it is impossible to tell by observing a value of 2 in Z whether it is the image of (2,1), (4,2), (-12,-6), or some other value of Y; this information is lost in the abstraction.) On the other hand, abstractions can be useful for suppressing unessential detail, information, non-determinism, and artificial restrictions; they can make a problem easier to understand and analyze; they facilitate the design of complex systems. This, as Horning and Randell, Dijkstra, and others have pointed out, is the whole point of forming abstractions.

Example: An Algol program describes an abstract process. The processor is defined by the statements of the program, and the information set consists of all possible incarnations of all variables. In principle, there can be infinitely variables and each can have

infinite precision. A realization of this process would have only finitely many incarnations of finitely many variables, each with finite precision. Thus an abstracting function would map each value of each variable of the realization into the same value of a corresponding variable of the infinite information set. The purpose of this abstraction is to facilitate the design and analysis of algorithms for which the restriction of finiteness is an unnecessary complication. The price which is paid is that certain information about the realization - particularly information about the effect of overflowing one of its finite constraints - is lost in the abstract process.

Abstractions of Cooperating Processes: An Example

We can explore the role of abstractions in verifying a set of cooperating processes by considering an example. This will introduce us to some new problems which will be the subject of the remainder of the chapter. The system we will examine is an implementation of a proposal by Lynch [1971] for a reliable, efficient communication system over a noisy, unreliable telecommunication link. The system multiplexes N logical communication channels on a single full-duplex line and provides each logical channel with a structure to allow it to cope with transmission errors in a manner described by Lynch [1968]. In principle, each logical channel acts like a half-duplex communication line, transmitting messages in alternate directions. Each message is accompanied by two bits which acknowledge receipt of the previous message and which identify the current one. A given message must be retransmitted as often

as necessary until an acknowledgment is returned that it has been received correctly.

The system consists of two processes at each end of the line, and both ends of the line are identical except for the initial value of one array. We already introduced one of the processes in Figure II.1. In Figures V.1 and V.2 we present a program and transition graphs describing the processes at one end of the line. The operation "transmit" is synchronized by the apparatus with a "receive" operation at the other end of the line. In the normal case the information received is exactly that transmitted, and "receive" returns a Boolean value of false indicating no errors. However, if noise on the line causes the information to be incorrect, the "receive" operation returns a Boolean value true, indicating that errors occurred. We assume that the apparatus can distinguish between correct and incorrect transmissions. In some cases, "receive" does not sense a transmission at all (these are called "line drops") - i.e., it completely misses the fact that a message was transmitted - and after an appropriate wait, it receives a later transmission.

The structure of the system and the function of the acknowledgment information can be better understood by considering abstractions of these processes. In particular, Figure V.3 shows how the two processes appear to the i^{th} logical channel. The abstract transmitter process is formed from the original transmitter process by the following mapping (v indicates a node of the transition graph of the transmitter).

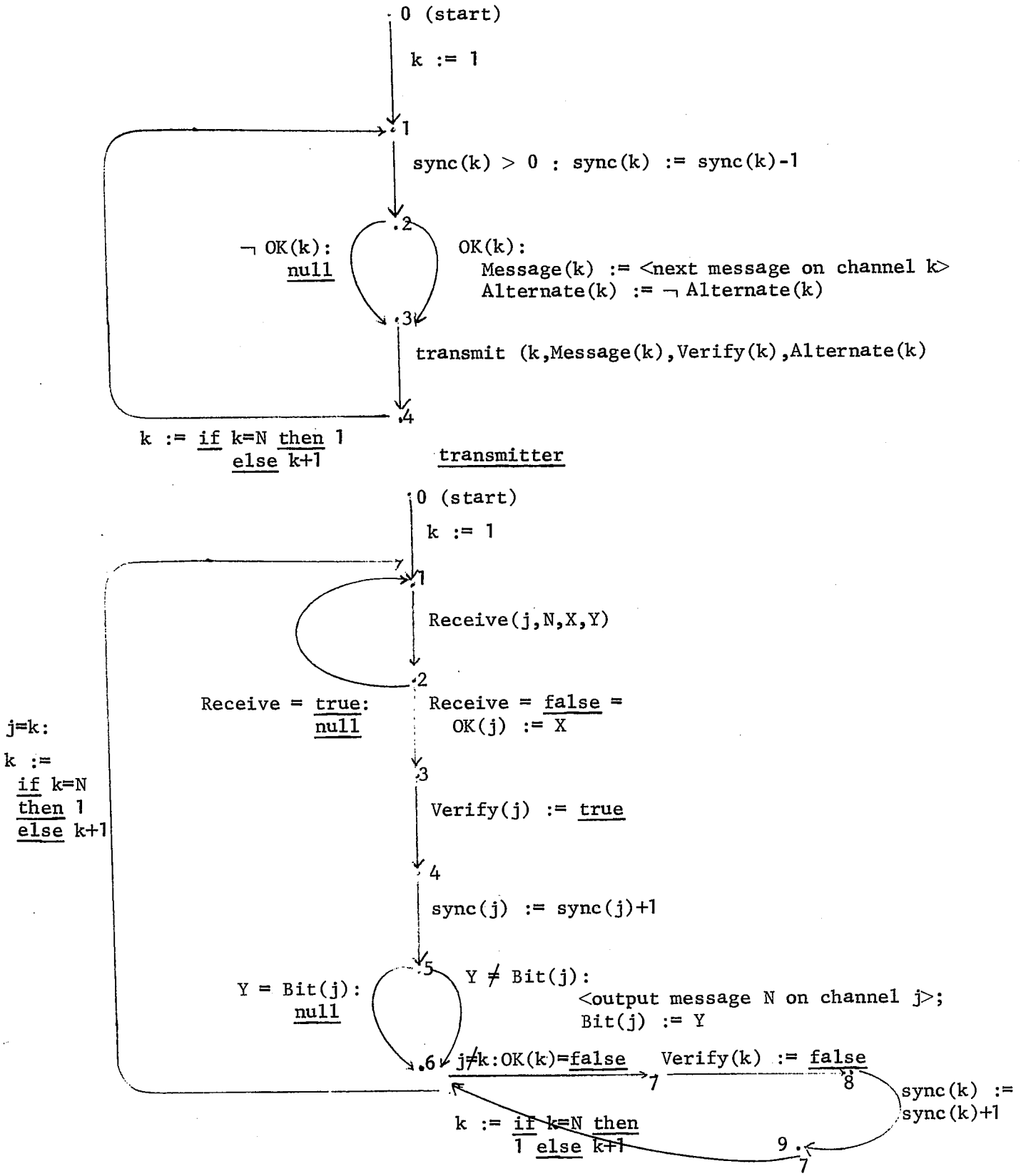
$$f(k, \text{Message}, \text{Verify}, \text{OK}, \text{Alternate}, \text{sync}, v) =$$

$$\begin{cases} \text{undefined if } k \neq i \\ \overline{\text{Message}}, \overline{\text{Verify}}, \overline{\text{OK}}, \overline{\text{Alternate}}, \overline{\text{sync}}, \bar{v} \text{ if } k = i \end{cases}$$

```
parbegin semaphore array sync (1:N);
         comment:  Each element of sync is initialized to 1 at one end
                   of the line but to zero at the other end;
         Boolean array  OK, Verify (1:N) = true;
transmitter:  begin integer k;
                   string array Message (1:N);
                   Boolean array Alternate (1:N) = true;
              for k = 1, (if k=N then 1 else k+1) while true do
                   begin P(sync(k));
                   if OK(k) then begin
                        Message(k) := <next message on logical channel k>;
                        Alternate(k) :=¬ Alternate(k) end;
                   transmit (k,Message(k),Verify(k),Alternate(k)) end;
              end of transmitter;
receiver:     begin integer j,k;
                   Boolean X,Y; string N;
                   Boolean array Bit (1:N) = true;
              for k = 1, (if k=N then 1 else k+1) while true do
                   if ¬ Receive(j,N,X,Y) then begin
                   OK(j) := X; Verify(j) := true; V(sync(j));
                   if Y ≠ Bit(j) then begin
                        <output message N on logical channel j>;
                        Bit(j) := Y end;
                   while j ≠ k do begin
                        OK(k) := Verify(k) := false; V(sync(k));
                        k := if k=N then 1 else k+1 end;
                   end;
              end of receiver;
parend.
```

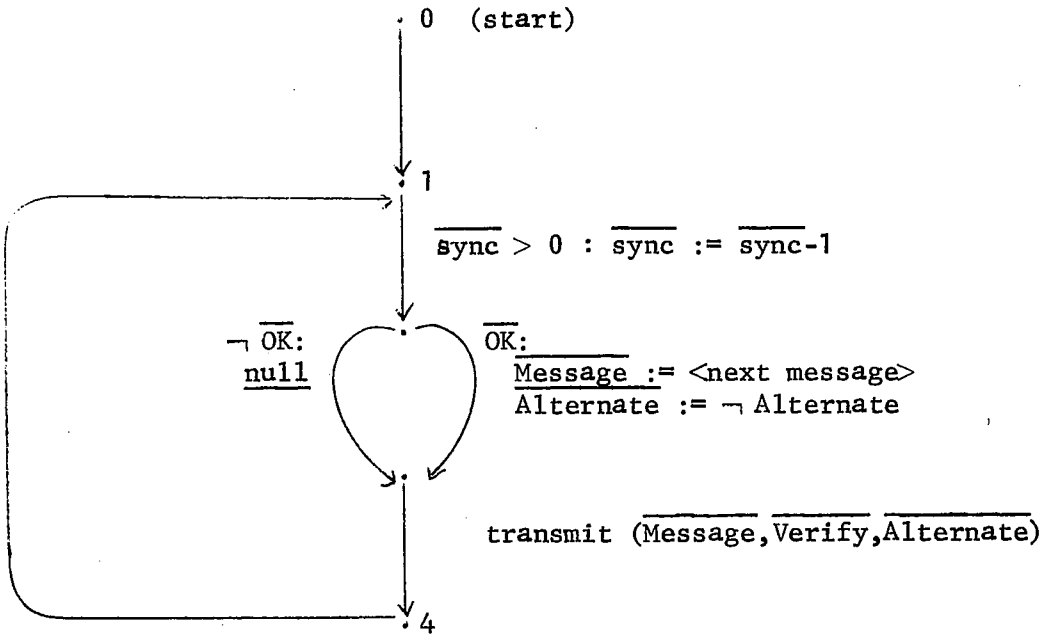
(Note: A general semaphore sync could be used in place of the array of binary semaphores sync(0:N-1), but this would complicate the discussion.)

Figure V.1

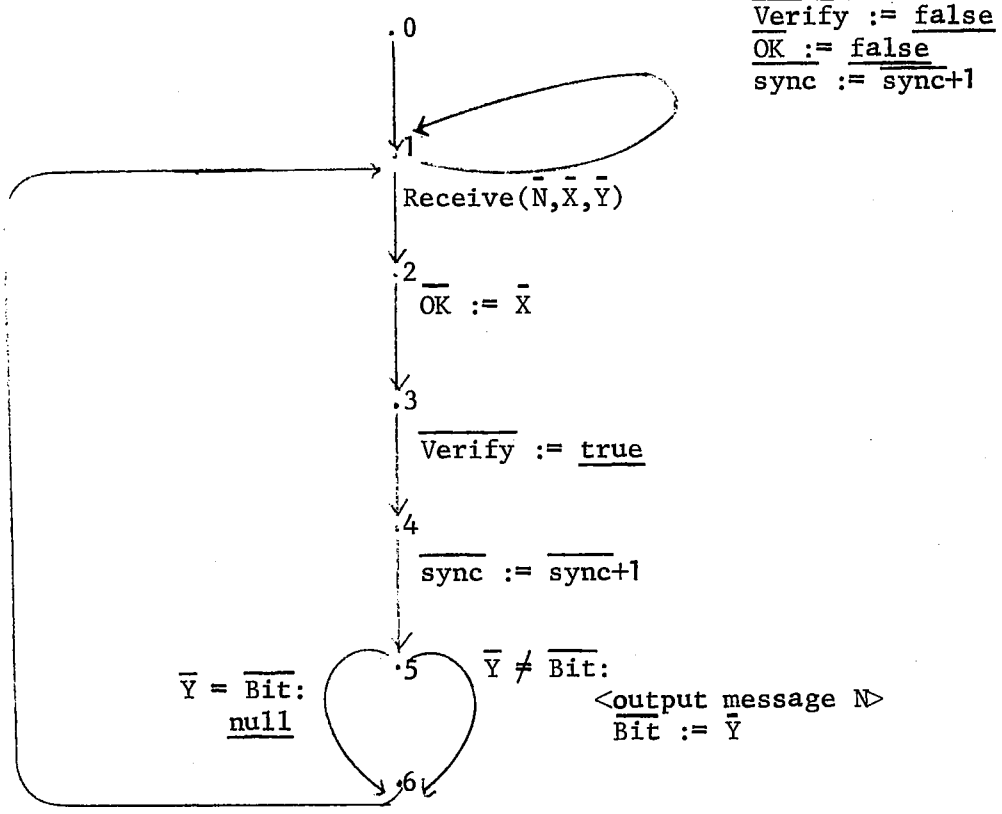


(Note that sequences of operations on local variables have been replaced by the equivalent single operations where possible.)

Figure V.2



Abstract Transmitter



Abstract Receiver

Figure V.3

where

$\overline{\text{Message}} = \text{Message}(i)$

$\overline{\text{Verify}} = \text{Verify}(i)$

$\overline{\text{OK}} = \text{OK}(i)$

$\overline{\text{Alternate}} = \text{Alternate}(i)$

$\overline{\text{sync}} = \text{sync}(i)$

$\bar{v} = v.$

The abstract receiver process is formed by a more complicated mapping, for it will eliminate the possibility of errors and line drops. It is defined by the following mapping (v indicates a node of the transition graph of the original receiver):

$$g(j,k,N,X,Y,OK,Verify,Bit,sync,v) = \begin{cases} \text{undefined if } j \neq i \\ \text{undefined if } j = i \text{ and } v = 6,7,8, \text{ or } 9 \\ \bar{N},\bar{X},\bar{Y},\overline{OK},\overline{Verify},\overline{Bit},\overline{sync},\bar{v} \quad \text{otherwise} \end{cases}$$

where

$\bar{N} = N,\ \bar{X} = X,\ \bar{Y} = Y$

$\overline{\text{OK}} = \text{OK}(i)$

$\overline{\text{Verify}} = \text{Verify}(i)$

$\overline{\text{Bit}} = \text{Bit}(i)$

$\overline{\text{sync}} = \text{sync}(i)$

$\bar{v} = v.$

We conjecture that the graphs of Figure V.3 are indeed transition graph representations of the abstract transmitter and receiver processes defined by these mappings. This raises two important questions:

(1) How can we verify the conjecture? I.e., how do we know that the transition graphs of Figure V.3 represent abstractions of Figure V.2?

(2) Is the product of these two (abstract) processes an abstraction of the product of the two given processes? More precisely, can we draw conclusions about the cooperation of the processes of Figure V.2 by considering the cooperation of Figure V.3?

We do not have complete answers to either question, but we will explore them briefly in the next two sections.

When is an Abstraction?

It is currently considered good programming practice to design a system "from the top down" - i.e., starting with the most abstract specification of its parts and building successively more detailed realizations (see, for example, Dijkstra [1970], Snowden [1971], Hoare [1971b], or Brinch Hansen [1972]). Thus in the course of developing a system, a designer might have in one hand a set of abstract processes about which he has verified some properties, and in the other hand, a set of processes alleged to be realizations of the first set. He must be able to convince himself that the allegation is true - i.e., that each process of the first set is an abstraction of processes of the second set. When he does, he can infer properties of the realizations from the properties he has already established in the abstraction.

Alternatively, the designer may proceed by using a given collection of processors and information sets to build new, "abstract" processors

and information sets. Then he repeats the procedure, over and over again, using the processes of one level to implement a more abstract set at the next level (e.g., the THE System of Dijkstra [1968b]). At each level, he can ignore the details of earlier ("less abstract") levels and concentrate on those of the given level, provided that he is confident that its processors and information sets are indeed images of the earlier levels.

In both cases it is necessary to show that one process is an abstraction of another. If the programs were developed by "stepwise refinement" (Wirth [1971]), then this amounts to showing that subprograms are equivalent to the abstract operations which they replace. (This is not often regarded as a difficult task, but Henderson and Snowden [1971] have shown that it is easy to develop a false sense of security about making refinements correctly.) When the abstracting functions are more complicated than mere refinements, as in our communication system example or in the THE system, it becomes more difficult and less obvious.

One method of proving that a process is an abstraction of another is suggested by the Weak and Strong Termination Theorems. Recall that the image processor includes an operation for every basic sequence of states in the realization.* Thus, if we have the transition graph representations of both a process and its alleged abstraction, it would be sufficient to show that every basic sequence of states in the former corresponds to an arc in the latter. That is, each computation of the given process which starts at state in the domain of the abstracting function must either

*An obvious generalization would allow the image processor to include more operations. Clearly this would not alter our ability to infer properties in the realization from those in the abstraction.

(1) lead directly to a state which has as its image an immediate successor of the starting state, or

(2) not lead to any state in the domain of the abstracting function.

In principle, the Weak Termination Theorem provides a means of testing this condition. We identify as a halt node each state of the given process which both is in the domain of the abstracting function and does not have a successor of the image of the given state as its own image. Then the Weak Termination Theorem provides a necessary and sufficient condition that no computation reach one of those states: namely, that there exists a set of assertions which satisifes a certain well-formed formula based on the transition graphs in the intended interpretations. If, instead, we designate as halt states all those states which have images which <u>are</u> successors of the image of the given state, then the Strong Termination Theorem provides a stronger condition: Each computation satisfies condition (1) above if and only if a certain formula is unsatisfiable in the intended interpretation.

In practice, more machinery is necessary before either of these theorems can be applied to show that one process is an abstraction of another. Such machinery should include a convenient way of representing "unobservable" states (i.e., states not in the domain of the abstracting function). It would also have to be particularly suited toward processes which are combined with other processes. Thus, our ability to verify the abstraction-realization relationship between two processes (or between two sets of processes) is still very much an open question.

Products of Abstractions

In a discussion about abstractions of operating systems processes, a fundamental question concerns conditions under which it is meaningful to consider the product of two abstract processes. The key property of an abstraction is that the set of its computations includes the image of each computation of its realization. But it is not always true that every computation of a product of two processes has an image which is a computation of the product of the abstractions of those processes - i.e., the operations of taking a product and taking an abstraction do not commute with each other.

Example: Let the information set Y consist of two integers, a and b, and let the processor P be defined to have a single internal state and to include the operation

$$(a,b) \xrightarrow{P} (a+b, 1-b)$$

for each pair of integers (a,b). Consider an abstracting function f which maps this information set into the single rational number c by

$$f(a,b) = \frac{a}{b} \text{ if } b \neq 0$$

$$f(a,0) \text{ undefined.}$$

Then the image processor (denoted by f(P)), includes only operations of the form

$$\frac{a}{b} \xrightarrow{f(P)} \frac{a+b}{1-b} \quad \text{for } b \neq 0,\ b \neq 1$$

$$\frac{a}{b} \xrightarrow{f(P)} \frac{a+b}{b} \quad \text{for } b = 1.$$

Now consider a second processor, Q on Y defined to have a single internal state and to include the operations

$$(a,0) \xrightarrow{Q} (2a,0)$$

for each integer $\underline{a}$. Let $\underline{g}$ be second abstracting function which maps values of Y into the integer $\underline{d}$, defined by

$$g(a,b) = \underline{a} + \underline{b}.$$

Thus the image processor g(Q) includes only operations of the form

$$a \xrightarrow{g(Q)} 2a$$

since $\underline{b}$ must be zero.

One computation of the product PX Q on Y given the initial state (1,1) is

$$\{(1,1),\ (2,0),\ (4,0),\ (4,1),\ldots\},$$

a result of alternating the actions of P and Q. In the abstract information set consisting of the variables $\underline{c}$ and $\underline{d}$, the image of a value $(\underline{a},\underline{b})$ of Y is

$$(\frac{a}{b},\ a+b)$$

for $b \neq 0$ and undefined if $b = 0$ (since f is undefined for $b = 0$). Thus the image of this computation is

$$\{(1,2),\ (4,5),\ldots\}.$$

But the operation

$$(1,2) \longrightarrow (4,5)$$

is not included in either image processor. Hence the product of two processes can define a computation with an image which is not a computation of the product of the abstractions of the two processes.

Thus, we are led to look for other ways in which it is meaningful to consider the products of abstractions of processes. There are some well-known cases - for example, the technique of simulating indivisible operations by using critical sections of code and a mechanism to ensure that only one process has access at any given instant. In this case, the abstracting function is the identity mapping except on states inside the critical section, where it is undefined. Then the abstraction of the product of two such processes is not the same as the product of the abstractions of the two processes, but it acts similarly. I.e., although some computations of the former are not computations of the latter, all of their states occur in computations of the latter.

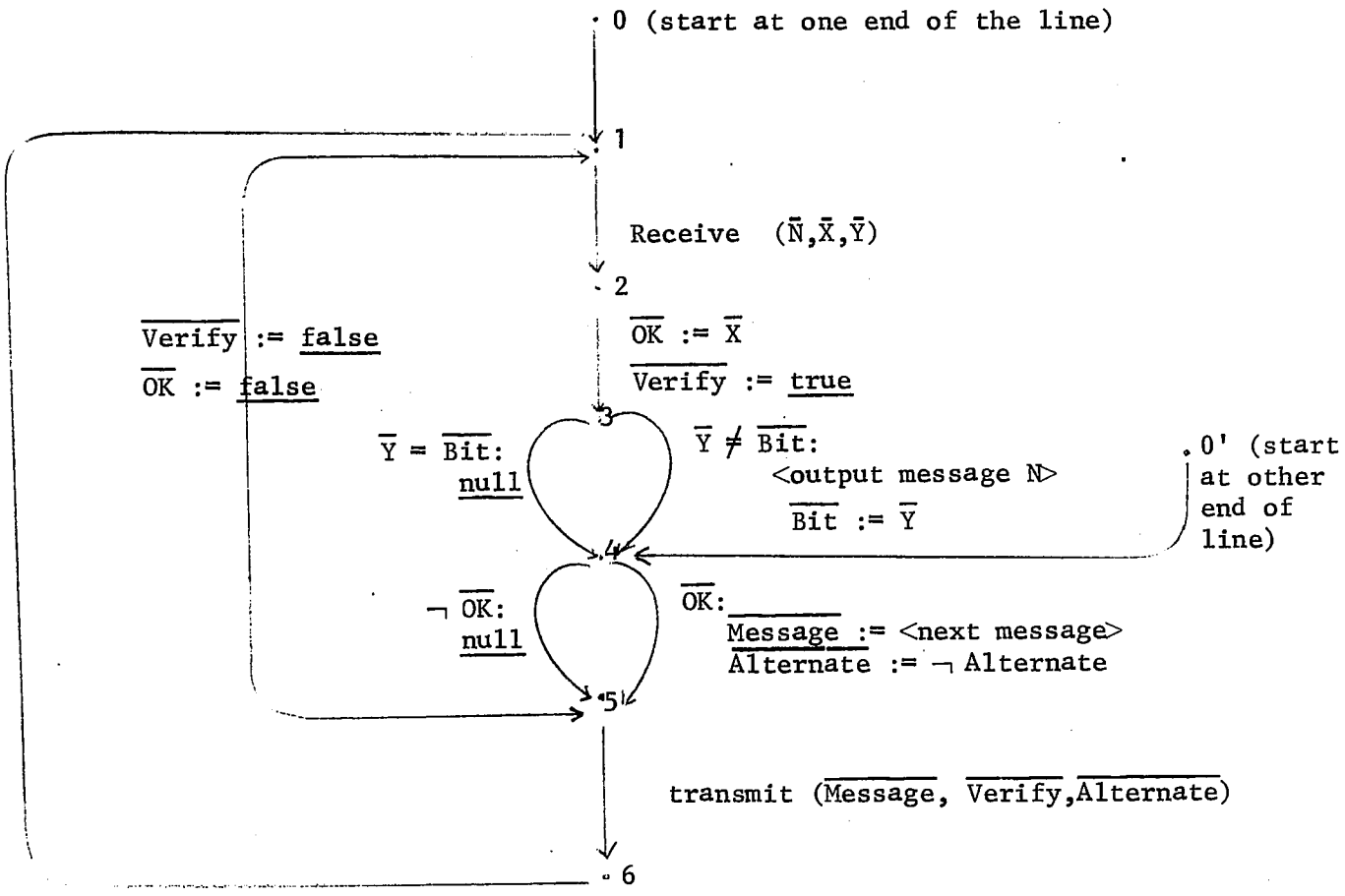
Example: If the product of the processes of Figure V.2 is abstracted by combining the functions we have presented earlier, many of its computations will contain long sequences of unobservable states. This is because one or the other of the component processes will be in an unobservable state - i.e., not working on logical channel i.

The images of such computations look like random selections of states which do not bear much relation to our expectations (as programmers) of the system. However, for each such computation, we can see intuitively that there is a computation of the

product of the abstractions which behaves in the same way with respect to channel i. This computation is a result of interleaving actions of the abstract receiver and transmitter in exactly the same way as the corresponding operations in the realizations are interleaved. Thus, we can still learn a lot about the system by studying the abstractions.

In particular, we would like to be able to conclude that the transition graph of Figure V.4 is an abstraction of the product of the process of Figure V.3, and hence an abstraction of the product of those of Figure V.2. But Figure V.4 is essentially a representation of Lynch's original communication system for a half-duplex line, a system which has been proven correct in both theory and practice (Lynch [1969]). Thus, if we could answer the two key questions of this chapter, we could conclude that the multiple channel system is correct with respect to the same criteria applied to each logical channel.

These examples suggest that a more precise formulation of the fundamental question of this section is as follows: What conditions are necessary and sufficient for an abstraction of the product of two processes to act like the product of the abstractions of those processes? Furthermore, what restrictions must be applied to assertions on the product of the abstractions in order to infer equivalent assertions about the realizations? These questions are of theoretical importance because we could not otherwise justify assertions about a system by proving them in abstractions of the system. In particular, we could not even justify assertions about a "real" system by proving them in our model of computation because



(Note: The start node at one end of the line is node 0, but at the other end, it is node 0'.)

Abstract Transceiver

Figure V.4

we have made the assumption of timelessness. This, itself, is an important abstraction of reality.

The questions are also of practical importance when we contemplate mechanical aids for system programmers such as an extension of the Snowden's PEARL System [1972]. We need to know what conditions to apply to refinements of abstract operations in order to be able to help the programmer keep them in proper perspective. Thus our question warrants a good deal more investigation.

COMMENTS AND CONCLUSIONS

In this thesis we have developed some potentially useful methods for verifying properties of processes which occur in operating systems. We have shown that by paying careful attention to representations we have been able to extend the work of Floyd, Manna, and King to apply to such processes. We have shown that knowledge about the structure of the processes can be used to reduce proofs of correctness to manageable proportions. Several examples taught us that finding suitable assertions is harder than verifying them. To cope with this we added information to the representation of the processes and used it to facilitate the statement of and verification of the properties of interest. We also showed that some interesting properties of operating system programs can be considered as variations on termination problems.

Our results suggest several directions for future work. On the purely mechanical side, there is need to adopt modern theorem-and program-proving methods to the formulas generated by our representations. This would bring us closer to the realization of a mechanical program verifier for cooperating processes. On the practical side, an important problem area is that of constructing suitable formal assertions about properties of cooperating processes - properties which are often expressed only in the programmer's intuition. That is, we need a structured body of knowledge and experience to lead us from problem statements to statements about the properties which solutions must possess. On the theoretical side, we must be able to factor a complex system into a hierarchical structure of subsystems in order to make it feasible to verify non-trivial

properties about non-trivial processes. To this end, we have introduced a potential framework for a formal definition of abstraction, and we have identified some problems which must be solved in order to make that definition usable.

How the results of this thesis might be used is something which cannot be predicted. Many of the author's programming colleagues have an aversion to program-proving af any sort - not from ignorance but from a well-reasoned, pragmetic view that whenever precise statements can be made about the execution of their programs, it is more profitable to incorporate them as redundancy checks at runtime rather than try to prove them. If these people are a representative sample, then program-proving will not achieve widespread use until and unless we have systems which are as available and as efficient as high-level language compilers are today.

An alternative, and perhaps more useful, application of our results is as an aid to the theoretical analysis of algorithms, operating system techniques, resource allocation mechanisms, etc. That is, we would expect that they would find more use in proving things about classes of programs rather than about individual programs.

Another promising application is in a mechanical "structured programming assistant" for cooperating processes. The problems of extending current systems (e.g, PEARL) to include non-deterministic and cooperating processes appear to be non-trivial and include those we have suggested above. But the potential of such a system in reducing design and programming costs, guaranteeing the reliability, and improving the understandability of operating systems is enormous. This would be ample reward for our efforts here.

We should make one final note about the non-determinism which we have built into our model of computation. A number of critics of this work have steadfastly maintained this non-determinism - and indeed, all non-determinism which we can observe in a multiprocessor computer system - is purely the result of not considering enough state information about the system. They assert that it could be reduced to determinism by describing more precisely the interactions of the various components. At one level, we note that even if the critics are right, it would be too hard to account for such information and much easier to live with the non-determinism. At a more fundamental level, though, such a Newtonian view of the universe and of the world of computing contradicts the accepted theories of physics, and it thus requires a degree of faith which this author has been unable to muster. We prefer to accept non-determinism as a fundamental fact of life and develop our programming and analytical skills around it - such as we have done in this thesis.

BIBLIOGRAPHY

Ashcroft and Manna [1971]
Ashcroft, E. and Z. Manna, "Formalization of Properties of Parallel Programs", Machine Intelligence 6, ed. Meltzer and Michie, University of Edinburgh Press, Edinburgh, 1971.

Barron [1969]
Barron, D. W., Assemblers and Loaders, MacDonald Press, London, 1969.

Bensoussan, Clingen, and Daley [1969]
Bensoussan, A., C. T. Clingen, and R. C. Daley, "The Multics Virtual Memory", Proceedings Second ACM Symposium on Operating Systems Principles, Princeton University, Princeton, New Jersey, October 1969.

Berry [1972]
Berry, Daniel, M., "The Equivalence of Models of Tasking", Proceedings of an ACM Conference on Proving Assertions about Programs, University of New Mexico, January 1972.

Brinch Hansen [1970]
Brinch Hansen, P., "The Nucleus of a Multiprogramming System", Communications of the ACM, Vol. 13, No. 4, p. 238, April 1970.

Brinch Hansen [1972]
Brinch Hansen, P., "Structured Multiprogramming", Communications of the ACM, Vol. 15, No. 7, p. 574, 1972.

Bruno, Coffman, and Hosken [1972]
Bruno, J. L., E. G. Coffman, and W. H. Hosken, "Consistency of Synchronization Nets Using P and V Operations", Technical Report 117, Computer Science Department, Pennsylvania State University, State College, Pa., June 1972.

Church [1956]
Church, A., Introduction to Mathematical Logic, Princeton University Press, Princeton, New Jersey, 1956.

Cooper [1968]
Cooper, D. C., "Program Scheme Equivalences and Second Order Logic", Machine Intelligence 4, ed. Meltzer and Michie, University of Edinburgh Press, Edinburgh, 1968.

Courtois, Heymans, and Parnas [1971]
Courtois, P. J., R. Heymans, and D. L. Parnas, "Concurrent Control with Readers and Writers", Communications of the ACM, Vol. 14, No. 10, p. 667, October 1971.

Daley and Dennis [1968]
Daley, Robert C. and Jack B. Dennis, "Virtual Memory, Processes, and Sharing in MULTICS", Communications of the ACM, Vol. 11, No. 5, p. 306, May 1968.

De Bruijn [1967]
De Bruijn,N. G., "Additional Comments on a Problem in Concurrent Programming Control", Communications of the ACM, Vol. 10, No. 3, pp. 137-138, March 1967.

Dijkstra [1965a]
Dijkstra, E. W., Cooperating Sequential Processes, Technological University of Eindhoven, Eindhoven, Netherlands, Report EWD123, September 1965.

Dijkstra [1965b]
Dijkstra, E. W., "Solution of a Problem of Concurrent Programming Control", Communications of the ACM, Vol. 8, p. 569, 1965.

Dijkstra [1968a]
Dijkstra, E. W., "Go To Statement Considered Harmful", Communications of the ACM, Vol. 11, No. 3, pp. 147-148, March 1968.

Dijkstra [1968b]
Dijkstra, E. W., "The Structure of the THE Multiprogramming System", Communications of the ACM, Vol. 11, No. 5, pp. 341-347, May 1968.

Dijkstra [1970]
Dijkstra, E. W., Notes on Structured Programming , Technical University of Eindhoven, Eindhoven, Netherlands, Report EWD249, 1970.

Dijkstra [1971]
Dijkstra, E. W., "Hierarchical Orderings of Sequential Processes", Acta Informatica, Vol. 1, No. 2, pp. 115-138, 1971.

Floyd [1967a]
Floyd, R. W., "Assigning Meaning to Programs", Proceedings of Symposia in Applied Mathematics, Vol. 19, American Mathematical Society, 1967.

Floyd [1967b]
Floyd, R. W., "Non-deterministic Algorithms", Journal of the ACM, October 1967.

Floyd [1971]
Floyd, R. W., "Toward Interactive Design of Correct Programs", Invited Lecture, IFIPS 1971 Congress, Ljubljana, Yugoslavia, 1971.

Good [1970]
Good, Donald I., Toward a Man-Machine System for Proving Program Correctness, Ph.D. thesis, University of Wisconsin, June 1970.

Good and London [1970]
Good, D. I., and R. L. London, "Computer Interval Arithmetic; Definition and Proof of Correct Implementation", Journal of the ACM, Vol. 17, pp. 603-612, 1970.

Habermann [1967]
Habermann, A. N., On the Harmonious Cooperation of Abstract Machines, Ph.D. thesis, Technical University of Eindhoven, Netherlands, 1967.

Habermann [1969]
Habermann, A. N., "Prevention of System Deadlocks", Communications of the ACM, Vol. 12, p. 373, July 1969.

Habermann [1972]
Habermann, A. N., "Synchronization of Communicating Processes", Communications of the ACM, Vol. 15, No. 3, p. 171, March 1972.

Havender [1968]
Havender, J. W., "Avoiding Deadlock in Multi-tasking Systems", IBM Systems Journal, Vol. 7, No. 2, p. 74, 1968.

Henderson and Snowden [1971]
Henderson, P. and R. Snowden, An Experiment in Structured Programming, University of Newcastle Upon Tyne Computing Laboratory, Newcastle Upon Tyne, Technical Report No. 18, 1971.

Hoare [1969]
Hoare, C. A. R., "An Axiomatic Basis for Computer Programming", Communications of the ACM, Vol. 12, pp. 576-583, 1969.

Hoare [1971a]
Hoare, C. A. R., "Proof of a Program FIND", Communications of the ACM, p. 31, January 1971.

Hoare, [1971b]
Hoare, C. A. R., "Towards a Theory of Parallel Programming", (Preliminary Draft of paper presented to the International Seminar on Operating System Techniques, Belfast, September, 1971.

Holt [1971a]
Holt, R. C., On Deadlock in Computer Systems, Ph.D. thesis, Cornell University, Ithaca, New York, January 1971.

Holt [1971b]
Holt, R. C., "Comments on Prevention of Systems Deadlocks", Communications of the ACM, Vol. 14, No. 1, January 1971.

Horning and Randell [1969]
Horning, J. J. and B. Randell, Structuring Complex Processes, Report Rc 2459 IBM Thomas J. Watson Research Center, Yorktown Heights, New York, 1969.

IBM Corporation [1970a]
IBM Corporation, Time Sharing System/360: Concepts and Facilities, Reference Manual, Poughkeepsie, New York, 1970.

IBM Corporation [1970b]
IBM Corporation, Operating System/360: Concepts and Facilities, Reference Manual, Poughkeepsie, New York, 1970.

IBM Corporation [1970c]
IBM Corporation, System/360: Principles of Operation, Reference Manual, Poughkeepsie, New York, 1970.

King [1969]
King, J. C., A Program Verifier, Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, Pa., 1969.

Knuth [1966]
Knuth, D. E., "Additional Comments on a Problem in Concurrent Programming Control", Communications of the ACM, Vol. 9, pp. 321-322, May 1966.

Knuth [1969]
Knuth, D. W., The Art of Computer Programming, Volume 1 (Fundamental Algorithms), Addison Wesley, Reading, Mass., 1969.

Ladner [1970]
Ladner, R., Verification of Transmission Algorithms, Ph.D. thesis, Case Western Reserve University, Cleveland, Ohio, September, 1970.

Lampson [1968]
Lampson, B., "A Scheduling Philosophy for Multiprocessing Systems", Communications of the ACM, Vol. 11, 5, p. 347, 1968.

London, [1970]
London, R. L., "C245: Proof of Algorithms - A New Kind of Certification", Communications of the ACM, Vol. 13, pp. 371-373, June 1970.

Luconi [1968]
Luconi, F. L., Asynchronous Computation Structures, Report MAC-TR-49, Project MAC (Ph.D. thesis), Massachussetts Institute of Technology, Boston, Mass., 1968.

Lynch [1968]
Lynch, W. C., "Reliable Full-duplex File Transmission over Half-Duplex Telephone Lines", Communications of the ACM, Vol. 11, No. 6, pp. 407-410, June 1968.

Lynch [1971]
Lynch, W. C., Reliable Data Transmission Sub-Channelization and Full Duplex, University of Newcastle Upon Tyne, Computing Laboratory, Technical Report No. 14, January 1971.

Manna [1968]
Manna, Z., Termination of Algorithms, Ph.D. thesis, Carnegie-Mellon University, Pittsburgh, Pa., 1968.

Manna and Pnueli [1970]
Manna, Z. and A. Pnueli, "Formalization of Properties of Functional Programs", Journal of the ACM, Vol. 17, No. 3, p. 555, July 1970.

Naur [1969]
Naur, P., "Programming by Action Clusters", BIT, Vol. 9, pp. 250-258, 1969.

Randell [1971]
Randell, B., "Operating Systems: The Problems of Performance and Reliability", Invited paper, 1971 IFIPS Congress, Ljubljana, Yugoslavia, 1971.

Rodriguez [1969]
Rodriguez, J. E., A Graph Model for Parallel Computations, D.Sc. thesis, Massachussetts Institute of Technology, Boston, Mass., MIT-Project MAC, Report MAC-TR-64, September 1969.

Saltzer [1966]
Saltzer, J. H., Traffic Control in a Multiplexed Computer System, Ph.D. thesis, Massachussetts Institute of Technology (Project MAC), Boston, Mass., June 1966.

Snowden [1971]
Snowden, R. A., PEARL: An Interactive System for the Preparation and Validation of Structured Programs", University of Newcastle Upon Tyne, Computing Laboratory Technical Report, 1971.

Van Horn [1966]
Van Horn, E. C., Computer Design for Asynchronously Reproducible Multiprocessing, Ph.D. thesis, Massachusetts Institute of Technology, Project MAC, 1966.

Wirth [1969]
Wirth, N., "On Multiprogramming, Machine Coding, and Computer Organization", Communications of the ACM, Vol. 12, No. 9, pp. 489-498, September 1969.

Wirth [1971]
Wirth, N., "Program Development by Stepwise Refinement", Communications of the ACM, Vol. 14, No. 4, p. 221, April 1971.

Zurcher and Randell [1968]
Zurcher, F. W. and B. Randell, "Iterative Multi-level Modelling - A Methodology for Computer Design", IFIPS 1968 Congress, Edinburgh, Scotland, 1968.