Proving the Correctness of Multiprocess Programs

LESLIE LAMPORT

Abstract—The inductive assertion method is generalized to permit formal, machine-verifiable proofs of correctness for multiprocess programs. Individual processes are represented by ordinary flowcharts, and no special synchronization mechanisms are assumed, so the method can be applied to a large class of multiprocess programs. A correctness proof can be designed together with the program by a hierarchical process of stepwise refinement, making the method practical for larger programs. The resulting proofs tend to be natural formalizations of the informal proofs that are now used.

Index Terms—Assertions, concurrent programming, correctness, multiprocessing, synchronization.

I. Introduction

The prevalence of programming errors has led to an interest in proving the correctness of programs. Two types of proof have been used: formal and informal. A formal proof is one which is sufficiently detailed, and carried out in a sufficiently precise formal system, so that it can be checked by a computer. An informal proof is one which is rigorous enough to convince an intelligent, skeptical human, and is usually done in the style of "journal mathematics proofs."

The need for correctness proofs is especially great with multiprocess programs. The asynchronous execution of several processes leads to an enormous number of possible execution sequences, and makes exhaustive testing impossible. A multiprocess program which has not been proved to be correct will probably have subtle errors, resulting in occasional mysterious program failures.

We have written several multiprocess algorithms to solve synchronization problems, and have given informal proofs of their correctness. Although the proofs were simple and convincing, they were ultimately based on the method of considering all possible execution sequences. This method is not well-suited for formal proofs. Other formal methods seemed either too difficult to be practical, or else were not applicable because they were based upon special synchronization primitives.

In this paper, we present a simple generalization of Floyd's inductive assertion method [9] which seems to be practical for proving the correctness of multiprocess programs. Using it, we have been able to translate our informal correctness proofs into formal ones. We feel that it can provide the basis for a general system for proving the correctness of most types of multiprocess programs.

Programs are simply represented by ordinary flowcharts, and no particular synchronization primitive is assumed. Any desired primitive can easily be represented. The method is practical for larger programs because the proof can be designed together with the program by a hierarchical process of stepwise refinement.

To prove the correctness of a program, one must prove two essentially different types of properties about it, which we call *safety* and *liveness* properties.[1] A safety property is one which states that something will *not* happen. For example, the partial correctness of a single process program is a safety property. It states that if the program is started with the correct input, then it cannot stop if it does not produce the correct output. A liveness property is one which states that something *must* happen. An example of a liveness property is the statement that a program will terminate if its input is correct.

The techniques used to prove safety and liveness are quite different from one another. They are therefore described in separate sections. Each of these two sections begins with an informal description of the technique in terms of a simple example, then gives a formal exposition, and concludes with a longer example. For a short introduction, the reader can omit all but the first parts of these sections. A final section discusses some general aspects of the method, including its applicability and its relation to previous work.

II. Safety

The Producer/Consumer Example

Before describing our formal axiom system, we illustrate the basic proof procedure with a simple example. The producer/consumer problem has been used to illustrate different synchronization primitives [5], [10]. It consists of a producer process which puts messages into a buffer and a consumer process which removes the messages. We assume that the buffer can hold at most b messages, $b \geqslant 1$. The processes must be synchronized so the producer does not try to put a message into the buffer if there is no room for it, and the consumer does not try to remove a message that has not yet been produced.

We will give a very simple solution which uses no complicated synchronization primitive, and might thus be more efficient than the solutions described in [5], [10]. Although we have not seen precisely this algorithm in print before, it has undoubtedly been discovered countless times by programmers implementing buffered input/output.

In our solution, we let k be a constant greater than b, and let s and r be integer variables assuming values between 0 and

Manuscript received August 1, 1975; revised April 1, 1976.
The author is with Massachusetts Computer Associates, Inc., Wakefield, MA 01880.

[1] These terms are not equivalent to their counterparts in Petri net theory, but they are used to describe somewhat similar properties.

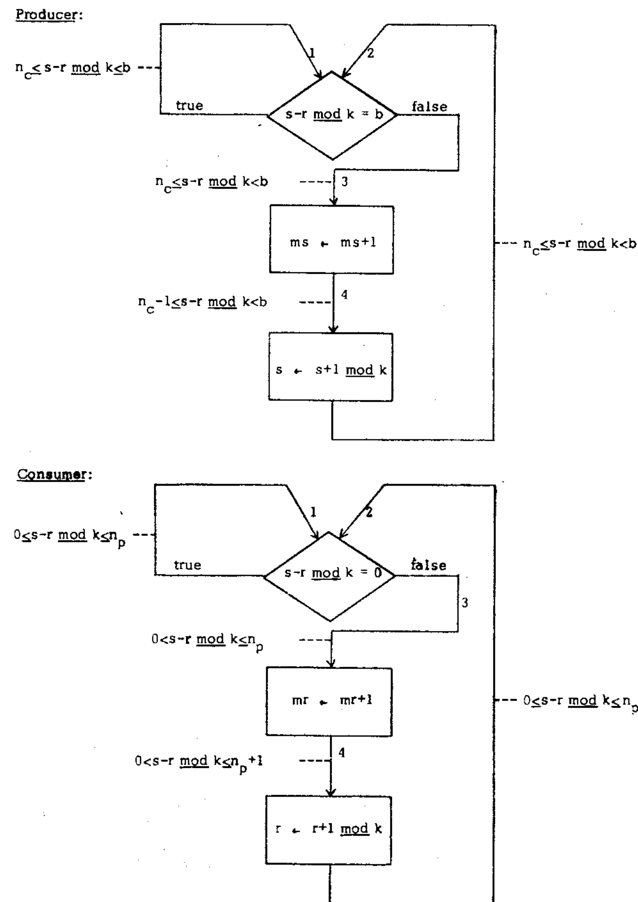


Fig. 1. Producer/consumer program.

$k - 1$. We assume that initially $s = r$ and the buffer is empty. The algorithm is given below. (It is easier to understand by first letting $k = \infty$. Then s represents the number of messages sent by the producer, and r represents the number of messages received by the consumer.)

Producer:

L: **if** $s - r$ *mod* $k = b$ **then goto** *L* **fi**;
put message in buffer;
$s := s+1$ *mod* k;
goto *L*;

Consumer:

L: **if** $s - r$ *mod* $k = 0$ **then goto** *L* **fi**;
take message from buffer;
$r := r+1$ *mod* k;
goto *L*;

We assume that reading and setting the variables r and s are indivisible operations. (The algorithm is easily modified to be correct under the weaker assumption A of [14], but the correctness proof is more complicated.) By choosing k to be a multiple of b, the buffer can be implemented as an array $B[0 : b-1]$. The producer simply puts each new message into $B[s \bmod b]$, and the consumer takes each message from $B[r \bmod b]$. However, proving this would complicate the proof, and it is left as an exercise for the interested reader. The only correctness property we will prove is that the producer never puts a message into a full buffer, and the consumer never takes one from an empty buffer.

We represent the two processes with the flowcharts of Fig. 1. For now, the reader should ignore the expressions attached to the arcs and consider only the nodes and the arcs. Note that each arc is numbered. We have represented the operations of sending and receiving a message by incrementing the fictitious variables ms and mr, respectively. We assume that initially $ms = mr$.

Each node of the flowchart is assumed to represent an indivisible operation. We assume that initially a token is placed on arc 1 of each process. The program is executed by arbitrarily choosing one of the processes and executing one step of that process, where a step consists of moving the token through one node onto another arc and changing the values of the variables in the obvious way. For now, we allow executions which always choose to execute the same process. Such executions must be disallowed to prove liveness properties, but they do not affect safety.

For simplicity, we have represented the producer's entire Algol statement $s := s+1$ as a single flowchart box. We could have more faithfully represented an actual implementation by splitting it into the two assignments $temp \leftarrow s + 1$ and $s \leftarrow temp$. However, it is easy to see that this would make no essential difference because s is not set by the consumer. We have also represented the **if** statement by a single decision box. We can do this by pretending that the entire execution of the statement occurs when the value of r is read. Similar remarks apply to the consumer.

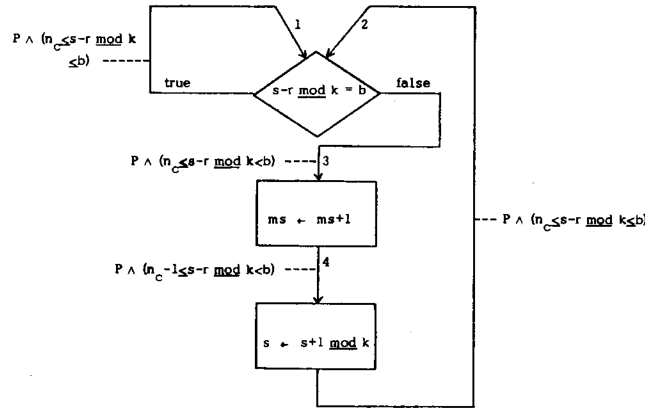


Fig. 2. Interpretation of producer for monotonicity proof.

Let n equal $ms - mr$. Then n obviously represents the number of messages in the buffer. To prove the desired safety property, we must prove that $0 \leqslant n \leqslant b$ holds throughout the execution of the program. We define the functions n_p and n_c as follows.

$$n_p = \begin{cases} n - 1 & \text{if the producer's token is on arc 4} \\ n & \text{otherwise} \end{cases}$$

$$n_c = \begin{cases} n + 1 & \text{if the consumer's token is on arc 4} \\ n & \text{otherwise} \end{cases}$$

An *assertion* is a logical function of program variables and token positions. An assertion is said to be *invariant* if it is always true during execution of the program. We want to prove that the assertion $0 \leqslant n \leqslant b$ is invariant for our producer/consumer program.

An *interpretation* of a program is an assignment of an assertion to each arc of the flowchart. Fig. 1 gives an interpretation of our producer/consumer program. We say that the program is in a legitimate state if each token is on an arc whose assertion is true.[2] The interpretation is said to be *invariant* if the program always remains in a legitimate state throughout its execution.

To prove the correctness of our program, we will prove that the interpretation of Fig. 1 is invariant. This is done by verifying the following two conditions: 1) the program's initial state is legitimate, and 2) if the program is in a legitimate state, then any execution step leaves it in a legitimate state. An interpretation which satisfies condition 2) is said to be *consistent.*

Since we start the program with each token on arc 1 and $s-r = n = 0$, it is easy to see that condition 1) is satisfied. The proof of condition 2) is done in two steps. First, we show that the interpretation of each process is consistent, i.e., we show that for each process: if the process' token is on an arc whose assertion is true and an execution step moves that token, then it is moved to an arc whose assertion is now true.

[2]It would be more precise to say that the variables and tokens are in a legitimate state, but for brevity we simply say that the program is in a legitimate state.

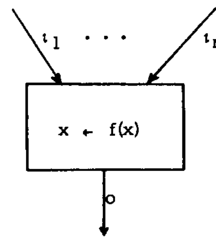
The proof that an interpretation of a single process is consistent is done essentially the same way as described by Floyd in [9] for a single process program. For each flowchart node, we prove that if the token is on an input arc of the node whose assertion is true, then executing the node moves the token to an output arc whose assertion is then true.

Proving the consistency of the producer's interpretation therefore requires verifying a consistency condition for each node of its flowchart. Verifying the condition for the decision box is simple, e.g., if the token is on arc 1 or 2 and $n_c \leqslant s-r \bmod k \leqslant b$, then execution of the decision node moves the token to arc 3 only if $s-r \bmod k \neq b$. This implies that $n_c \leqslant s-r \bmod k < b$, so the assertion on arc 3 is then true. Verifying the consistency condition for the ms assignment node is also simple, since executing it increases n_c by one and does not change s or r.

To prove the consistency condition for the s assignment node, we need the following result from number theory: if $s-r \bmod k < k-1$, then $(s+1 \bmod k) - r \bmod k = (s-r \bmod k) + 1$. Since we have assumed that $b < k$, this easily implies the required consistency condition.

We have thus shown that the producer's interpretation is consistent. A similar proof establishes the consistency of the consumer's interpretation. However, the consistency of each process' interpretation does not by itself imply the consistency of the entire program's interpretation. There still remains the possibility that an assertion attached to an arc was true when the process' token was moved to the arc, but was made false by the execution of the other process. The second step in proving the consistency of the interpretation is to show that this cannot happen.

Let P denote one of the assertions attached to the consumer's flowchart. We must show that if P is true, then it cannot be made false by executing one step of the producer. In that case, we say that P is *monotone* under the producer. To prove this, it suffices to show that if the producer's token is on an arc whose assertion is true, and P is true, then executing the producer's next flowchart node will leave P true. This is shown by proving the consistency of the interpretation of the producer shown in Fig. 2, which is formed by *and*ing P with each assertion of the producer's original interpretation. If the interpretation of Fig. 2 is consistent, then we say that P is *mono-*



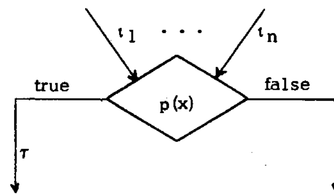


Fig. 3. Flowchart nodes.

tone under the interpretation of the producer shown in Fig. 1. The second step in proving the consistency of the program's interpretation therefore consists of proving that each assertion in the consumer's interpretation is monotone under the producer's interpretation, and vice-versa.

At this point, we urge the reader not to form any conclusion about how practical this step is for larger programs. This will be discussed later.

We now prove that each assertion P attached to the consumer's flowchart is monotone under the producer's interpretation. Instead of doing this separately for each of the three different assertions in the consumer's interpretation, we will do it all in one step by proving the monotonicity of the following assertion P for arbitrary constants u and $v: 0 < (s-r \bmod k) + u < n_p + v$.

To prove this, we must prove the consistency of the interpretation in Fig. 2 for this choice of P. As before, this requires proving consistency at each flowchart node. Since we have already proved the consistency of the producer's original interpretation, we must only show that if the assertion on the input arc of the node is true, then executing the box leaves P true.

The proof of consistency at the decision box is trivial. The consistency condition for the *ms* assignment box follows from the observation that by the definition of n_p, executing this node leaves n_p unchanged. The consistency at the *s* assignment node follows from the fact that executing it increments both $s-r \bmod k$ and n_p by one.

To complete the proof that the program's interpretation is consistent, we must prove that each assertion in the producer's interpretation is monotone under the consumer's interpretation. The proof is similar to the one we just did, and is left to the reader.

We now use the invariance of the program's interpretation to prove the invariance of the assertion $0 \leq n \leq b$. From the producer's interpretation, we see that regardless of which arc the producer's token is on, we must have $n_c \leq b$. Hence, the assertion $n_c \leq b$ is invariant. Similarly, the consumer's interpretation shows that $0 \leq n_p$ is invariant. Since $n_p \leq n \leq n_c$ by definition, the invariance of these two assertions implies that $0 \leq n \leq b$ is invariant. This completes our correctness proof.

The Formalism

We now formalize the method of proof used in our example. First, we formally define what a program is, and introduce some notation.

Definition 1: A *program* Π consists of a *value set* X, a finite collection or *processes* $\Pi_1, \cdots, \Pi_N$, and an *initial assertion* A_0. Each process is a directed graph composed of the two types of nodes shown in Fig. 3, where $f: X \to X$ and $p: X \to \{true, false\}$ are multivalued functions and $n \geq 1$. We let Γ_k denote the set of arcs of Π_k, and let Γ denote $\Gamma_1 \times \cdots \times \Gamma_N$. An *assertion* A is a single-valued function $A: X \times \Gamma \to \{true, false\}$. □

The value set X denotes the set of all possible values of the vector x of program variables. If $f(x)$ has more than one value for the current value of x, then executing the assignment node consists of setting x to any arbitrary one of those values. Similarly, if $p(x)$ has two values, then executing the decision node may move the token to either of its output arcs. Thus, we are allowing our processes to be nondeterministic.

The assumption that $n \geq 1$ means that each node has at least one input arc. An assignment node always has one output arc, and a decision node always has two.

We can think of Γ as the set of all possible ways to place one token on an arc of each process. The state of the program at any time during its execution is described by an element of

$X \times \Gamma$. The initial assertion A_0 specifies the possible initial states of the program.

We explicitly note two things we are *not* assuming about the program:

1) We do not assume X to be a finite set. The vector x may be composed of infinitely many component variables, and a variable can have an infinite set of possible values.

2) We do not assume that every arc joins two nodes. An arc is called an exit (entry) arc if it is not an input (output) arc of any node. A process may have entry or exit arcs.

We will state some simple axioms about invariant assertions. We could define an execution of the program Π in the obvious way as a sequence of elements of $X \times \Gamma$, and then define an assertion to be invariant if it has the value *true* for every element of every execution. We could then prove that invariant assertions satisfy our axioms. However, this would be a tedious exercise in proving the obvious. Instead, we will consider our axioms to provide a definition of invariance. This will make it unnecessary to introduce formally the concept of an execution.

We let $\wedge, \vee, \supset, \sim, \equiv$ denote the usual logical operations *and*, *or*, *implication*, *negation*, and *equivalence*. If $\mathfrak{A}$ is a set, then $\bigwedge \mathfrak{A} (\bigvee \mathfrak{A})$ denotes the logical *and* (*or*) of all the elements of $\mathfrak{A}$. ($\mathfrak{A}$ will be either a subset of $\{true, false\}$ or else a set of assertions.) We define $\bigwedge \phi \equiv true$ and $\bigvee \phi \equiv false$ for the empty set ϕ.

Definition 2: A Π_k-assertion A_k is an assertion which is independent of Π_k's token position–i.e., such that the value of $A(x, \gamma_1, \cdots, \gamma_N)$ does not depend upon the value of γ_k. If A is any assertion, and α is an arc of Π_k, then $A_{(k)}^{(\alpha)}$ is the Π_k-assertion defined by $A_{(k)}^{(\alpha)}(x, \gamma_1, \cdots, \gamma_N) = A(x, \gamma_1, \cdots, \gamma_{k-1}, \alpha, \gamma_{k+1}, \cdots, \gamma_N)$. For any assertion A and any multivalued function $f: X \rightarrow X$, the assertion $A \circ f$ is defined by $A \circ f(x, \gamma) = \bigwedge \{A(z, \gamma) : z = f(x)\}$. We let $\pi_k : X \times \Gamma \rightarrow \Gamma_k$ denote the obvious projection mapping. □

Note that $\pi_k = \alpha$ denotes the mapping which takes (x, γ) into the Boolean value $\pi_k(x, \gamma) = \alpha$, i.e., $\pi_k = \alpha$ is the assertion which is true iff (if and only if) Π_k's token is on arc α. Similarly, if Λ is any set of arcs in Π_k, then $\pi_k \in \Lambda$ is the assertion which is true iff Π_k's token is on some arc in Λ.

We assume some formal system for proving theorems which includes the propositional calculus. To prove correctness, it must be capable of proving useful theorems about the program's value set X. However, the details of the system will not concern us.

We let $\vdash [\mathfrak{T}]$ denote that $\mathfrak{T}$ is a provable theorem in our formal system. An assertion A is considered to represent the theorem $\forall (x, \gamma) \in X \times \Gamma : A(x, \gamma) = true$. We therefore write $\vdash [A]$ or $\vdash [A(x, \gamma)]$ to denote that this theorem is provable. Note that the theorem $A_{(k)}^{(\alpha)}$ is equivalent to $\forall (x, \gamma) \in \pi_k^{-1}(\alpha) : A(x, \gamma) = true$.

We let $\Vdash [A]$ or $\Vdash [A(x, \gamma)]$ denote that the assertion A is invariant.

Definition 3: a) An *interpretation* I_k of the process Π_k is a mapping which assigns to each arc $\alpha \in \Gamma_k$ a Π_k-assertion I_k^α. The interpretation is said to be *consistent at a node* of Π_k if the appropriate one of the following conditions is satisfied, where the notation is as in Fig. 3 and I_k^ι is defined to be the assertion $\bigvee \{I_k^\alpha : \alpha = \iota_1, \cdots, \iota_n\}$.

assignment node: $\Vdash [I_k^\iota \supset I_k^o \circ f]$

decision node: $\Vdash [(I_k^\iota \wedge p \supset I_k^\tau) \wedge (I_k^\iota \wedge \sim p \supset I_k^\varphi)]$.

The interpretation I_k is *consistent* if it is consistent at every node of Π_k.

b) An assertion A is said to be *monotone under* I_k if the interpretation $A \wedge I_k$, which assigns to each arc $\alpha \in \Gamma_k$ the assertion $A_{(k)}^{(\alpha)} \wedge I_k^\alpha$, is consistent.

c) An *interpretation* I of the program Π consists of an interpretation I_k of each process Π_k of Π. The interpretation I is *consistent* if: i) each I_k is consistent, and ii) for each j and k with $j \neq k$ and every $\beta \in \Gamma_j$, the assertion $I_j^\beta \wedge (\pi_j = \beta)$ is monotone under I_k. The interpretation I is *invariant* if it is consistent and $\vdash [A_0 \supset \bigwedge \{(\pi_k = \alpha) \supset I_k^\alpha : \text{all } k, \text{and all } \alpha \in \Gamma_k\}]$. □

Part a) of the definition is a straightforward formalization of the informal definition used in the above example. In part b), the use of the assertions $A_{(k)}^{(\alpha)}$ allows us to make use of the knowledge of Π_k's token position. This is illustrated by considering how the proof of the monotonicity of P in our example is formalized. The assertion $P_{(c)}^{(\alpha)}$ is obtained from P by replacing n_c by n if $\alpha \neq 4$ and by $n-1$ if $\alpha = 4$. In part c), proving the monotonicity of $I_j^\beta \wedge (\pi_j = \beta)$ is easier than proving the monotonicity of I_j^β because we can use the knowledge of Π_j's token position. This was not necessary in our example. Observe that this makes the monotonicity condition trivial at a node of Π_k if its input assertion I_k^ι implies that $\pi_j = \beta$ is false.

Instead of requiring the consistency condition at a node to be provable, we have made the weaker requirement that it be invariant. This allows us to assume the truth of assertions which have already been proved to be invariant. We could actually replace "$\Vdash$" by "$\vdash$" in this definition, and then prove the original weaker definition as a metatheorem.[3] We will not bother to do this.

Having made our definitions, we can now state our two axioms.

S1: For any assertions A and B,

a) if $\vdash [A]$, then $\Vdash [A]$.

b) if $\Vdash [A]$ and $\Vdash [A \supset B]$, then $\Vdash [B]$. □

S2: If I is an invariant interpretation, then for all k and all $\alpha \in \Gamma_k$: $\Vdash [(\pi_k = \alpha) \supset I_k^\alpha]$. □

These two simple axioms are sufficient to provide a formal foundation for proving safety properties of programs. However, it is convenient to have some other theorems which can be used to simplify the proofs. We now state a few such theorems. Their proofs are simple and are omitted.

Theorem 1: Let I be an invariant interpretation of the program Π and let A be an assertion. If for some k and all $\alpha \in \Gamma_k$: $\Vdash [(I_k^\alpha \wedge (\pi_k = \alpha)) \supset A]$, then $\Vdash [A]$. □

The next theorem is what makes our method practical for larger programs. It is stated somewhat informally to make it

[3] A metatheorem is a theorem *about* a formal system. Unlike a theorem *in* the system, it could become false if another axiom is added to the system.

easier to understand. We leave the rigorous formal statement of part a) to the reader.

Theorem 2: Let I_k be a consistent interpretation of process Π_k and let A be any assertion.

a) To prove that A is monotone under I_k, it suffices to prove that the interpretation $A \wedge I_k$ is consistent at each node whose execution can change the value of A.

b) Let $\Lambda \subset \Gamma_k$. To prove that the assertion $(\pi_k \in \Lambda) \supset A$ is monotone under I_k, it suffices to prove that the interpretation $[(\pi_k \in \Lambda) \supset A] \wedge I_k$ is consistent at each node which has an element of Λ as an output arc. □

Before stating the next theorem, we need the following definition.

Definition 4: An assertion A is said to be *monotone* if there is an invariant interpretation I of Π such that A is monotone under each I_k. □

Theorem 3: If an assertion A is monotone and $\vdash [A_0 \supset A]$, then $\Vdash [A]$. □

To prove the following theorem, we need to be able to prove that for any set of assertions $\mathfrak{A}$: if $\vdash [A]$ for each $A \in \mathfrak{A}$, then $\vdash [\bigwedge \mathfrak{A}]$ and $\vdash [\bigvee \mathfrak{A}]$. Thus, if the theorem is to be true for infinite sets of assertions, then our formal proof system must be stronger than the simple predicate calculus.

Theorem 4: Let $\mathfrak{A}$ be a set of assertions and I_k an interpretation of a process Π_k.

$$\text{If each } A \in \mathfrak{A} \text{ is } \left\{\begin{array}{l}\text{invariant}\\ \text{monotone}\\ \text{monotone under } \Pi_k\end{array}\right\},$$

$$\text{then } \bigwedge\mathfrak{A} \text{ and } \bigvee\mathfrak{A} \text{ are also } \left\{\begin{array}{l}\text{invariant}\\ \text{monotone}\\ \text{monotone under } \Pi_k\end{array}\right\}. \quad \square$$

Subroutines

The basic method of proving the safety of a program is to prove a certain interpretation I to be invariant. This requires proving that $I_j^\beta \wedge (\pi_j = \beta)$ is monotone under I_k for all j, k, β with $j \neq k$ and $\beta \in \Gamma_j$, which requires proving a consistency condition for each node of Π_k. If the program has $O(m)$ nodes and arcs, this seems to imply that $O(m^2)$ individual proofs are needed to prove I invariant. If this were true, then our method would be impractical for large programs. To be practical, the number of operations must be an approximately linear function of the program size.

We will see that in practice, using Theorem 2 enables us to prove the monotonicity of $I_j^\beta \wedge (\pi_j = \beta)$ under I_k by proving consistency conditions for only a small number of nodes. Hence, only $O(m)$ proofs are needed. This is true for essentially the same reason that large programs can be written in the first place. If designing one step of a program required consideration of all other steps, then the effort of writing a program would grow as the square of its size, making large programs impractical. However, large programs can be designed, and their proofs of correctness designed along with them. What makes this possible is the method of designing by stepwise refinement. In order to discuss this method for our flowchart programs, we introduce some terminology. (In the following definition, a "subroutine" should be thought of as an *open* subroutine.)

Definition 5: A *subroutine* of a process Π_k is a subgraph composed of a nonempty set of nodes together with all the arcs of Π_k attached to those nodes. A *decomposition* of Π_k into subroutines is defined in the obvious way to be a directed graph whose nodes are subroutines of Π_k such that each node of Π_k belongs to exactly one subroutine. A decomposition of the program Π consists of a decomposition of each process Π_k of Π. A decomposition $\overline{\Sigma}$ of Π is a refinement of a decomposition Σ if Σ can be obtained in the obvious way by decomposing $\overline{\Sigma}$. □

The coarsest decomposition of a process Π_k is the trivial graph containing a single node, formed by decomposing Π_k into one subroutine. The finest decomposition consists of Π_k itself, and is formed by decomposing Π_k into subroutines each containing a single node.

To design a program, one designs a sequence of decompositions starting with the coarsest one and proceeding through successively finer decompositions until reaching the finest decomposition: the complete program. At each stage, one has both a decomposition and a specification of what each of its subroutine nodes is to do. To go to the next stage, one uses the specification of each node to refine it into simpler subroutine nodes. The refinement of each subroutine must only depend upon the refinement of a small number of other subroutines; otherwise the design procedure becomes too difficult.

With our method, the programmer designs his formal proof along with the program by designing an appropriate invariant interpretation. At each stage, he attaches assertions to the arcs of the decomposition. These assertions may not be fully specified, since the value set X need not be completely defined until later. In the next stage of the design, in which the programmer refines this decomposition, these assertions are more fully specified, and assertions are attached to the newly introduced arcs.

The programmer designs each refinement so that at the final stage, the interpretation of the complete program will be consistent. To show that this can be done in a hierarchical manner which makes it practical for larger programs, we will describe how the method is used in solving a more difficult problem.

The Bakery Algorithm

As our example, we will show how the solution to the mutual exclusion problem given in [13] could be constructed. Assume that we have N processes, each with a critical section. The problem is to find an algorithm which guarantees that at most one of the processes is in its critical section at any time. The other conditions which must be satisfied will not concern us here.

The first stage decomposition of each process is shown in Fig. 4. Note that subroutines are denoted by boxes drawn with dashed lines. Each subroutine is numbered, and we enclose its number in a box to denote the set of all arcs of the subroutine except its entry and exit arcs. Thus, $\pi_k \in \boxed{4}$ means that process Π_k is inside its critical section. We assume that initially, each process' token is in subroutine 8.

This decomposition is rather trivial. Subroutines 4 and 8 are

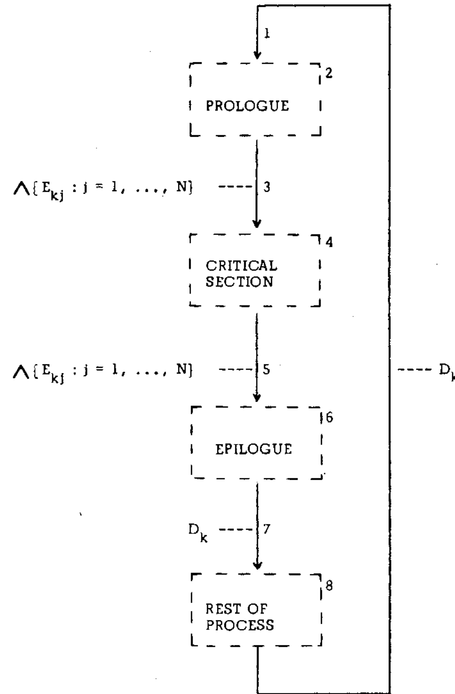


Fig. 4. Stage 1 decomposition of Π_k.

given. Our problem is to design subroutines 2 and 6. To prove correctness, we must show that two different processes cannot both be in subroutine 4 at the same time. We will do this by proving the invariance of an interpretation containing the indicated assertions, where D_k and E_{kj} will be defined later. We assume that D_k is also attached to all arcs of subroutine 8, and $\bigwedge\{E_{kj} : j = 1, \cdots, N\}$ is attached to all arcs in subroutine 4. The idea is to define E_{kj} so that if $j \neq k$, then $E_{kj} \wedge E_{jk} \wedge \pi_k \in \boxed{4} \wedge \pi_j \in \boxed{4} \equiv$ *false*. The invariance of the interpretation will imply that Π_k and Π_j cannot both be in their critical sections.

We will use the following idea for our solution. Let each process Π_k have a variable $n[k]$ which is initially zero. Before entering its critical section, Π_k sets $n[k]$ to some number greater than the value of $n[j]$ for all $j \neq k$. It then waits until for each $j \neq k$ with $n[j] > 0$: either $n[k] < n[j]$ or $n[k] = n[j]$ and $k < j$ whereupon it enters its critical section. Let us define the predicate $k \lll j$ to equal $(0 < n[k] < n[j]) \vee (0 = n[j] < n[k]) \vee (n[k] = n[j] \wedge k \leqslant j)$. (Note that $\lll$ is a total ordering of $\{1, \cdots, N\}$.) Then Π_k waits until $k \lll j$ for all j before entering its critical section.

This leads us to the stage 2 decomposition shown in Fig. 5, in which we have decomposed subroutine 2 and more precisely specified subroutine 6. In subroutine 6, we are allowing the possibility that setting $n[k]$ to zero may be a complex operation. Note the numbering of the subroutines and arcs in the decomposition of subroutine 2. In our notation for sets of arcs, we have $\boxed{2} = \boxed{2.1} \cup \{2.2\} \cup \boxed{2.3}$.

We have defined $D_k \equiv (n[k] = 0)$. Our first thought is to let E_{kj} be the assertion $(n[k] > 0) \wedge (k \lll j)$. However, this is not monotone under Π_j because Π_j might have decided to choose a value for $n[j]$ which would make $j \lll k$, but not yet have set $n[j]$ to this value. Hence, we could initially have

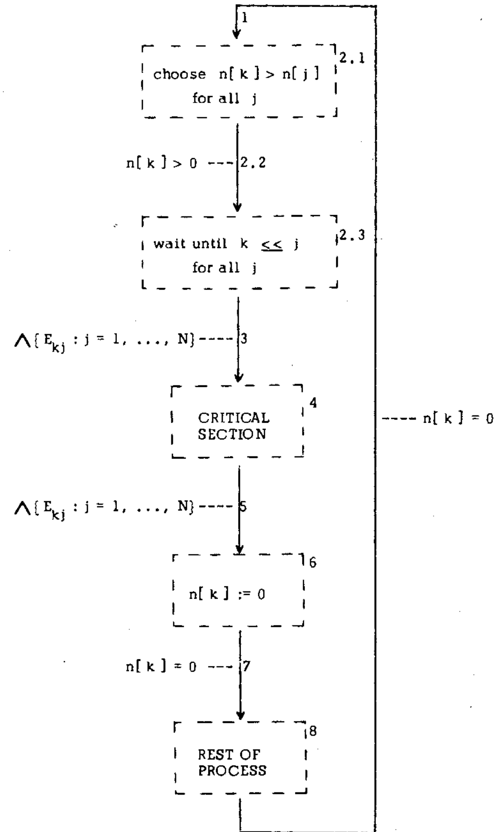


Fig. 5. Stage 2 decomposition of Π_k.

$n[j] = 0$ and $k \lll j$, but then $k \lll j$ becomes false after $n[j]$ is set to its new value. Therefore, we must let E_{kj} be an assertion which states that $n[k] > 0$ and either $k \lll j$ or else Π_j is

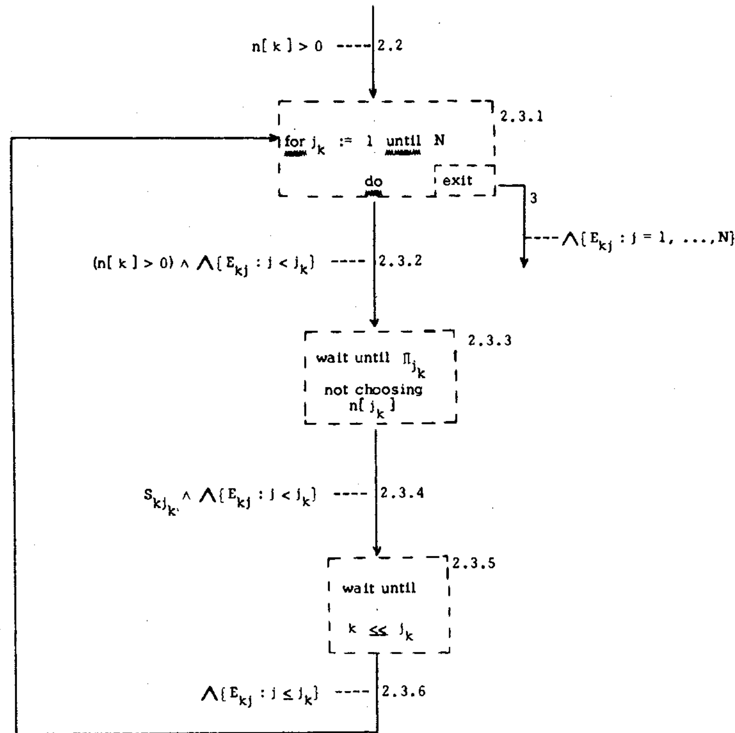


Fig. 6. Stage 3 decomposition of subroutine 2.3.

currently choosing a value of $n[j]$ which will make $k \ll j$. This assertion will be monotone under Π_j.

In the next stage, we decompose subroutine 2.3 as shown in Fig. 6. (The subscript k indicates that the variable j_k is only used by Π_k.) After Π_k executes subroutine 2.3.3, it then knows that any subsequently chosen value of $n[j_k]$ must be greater than $n[k]$—until Π_k completes its critical section and resets $n[k]$ to zero.

Let R_{kj} be an assertion which states that $n[k] > 0$ and if Π_j is not changing the value of $n[j]$, then $k \ll j$. Let S_{kj} be an assertion which states that $n[k] > 0$ and if Π_j is choosing a new value of $n[j]$, then it will choose one greater than the current value of $n[k]$. (We will specify R_{kj} and S_{kj} more precisely later.) We define $E_{kj} \equiv R_{kj} \wedge S_{kj}$. Our idea of why the program works is embodied by the interpretation of subroutine 2.3 shown in Fig. 6.

The next stage of the design procedure is to further specify subroutines 2.1 and 2.3.3. In Fig. 7 we have decomposed subroutine 2.1, introducing the Boolean array *cf*. The variable *cf*[k] is initially false, and is modified only in subroutines 2.1.1 and 2.1.5. We then define R_{kj} and S_{kj} as follows:

$$R_{kj} \equiv [n[k] > 0] \wedge [(\pi_j \notin \boxed{2.1.3} \cup \boxed{6}) \supset (k \ll j)]$$

$$S_{kj} \equiv [n[k] > 0] \wedge [(\pi_j \in \boxed{2.1.3}) \supset T_{kj}].$$

(Note that we have taken into consideration the fact that $n[j]$ might assume arbitrary values while Π_j executes subroutine 6.) We will define T_{kj} to be a function of process Π_j's local variables. If $T_{kj} = true$, it will mean that either $n[k]$ has not yet been read by subroutine 2.1.3 of Π_j, or else its current value was read. We can now describe subroutine 2.3.3 as a "wait until *cf*[j] = *false*" operation.

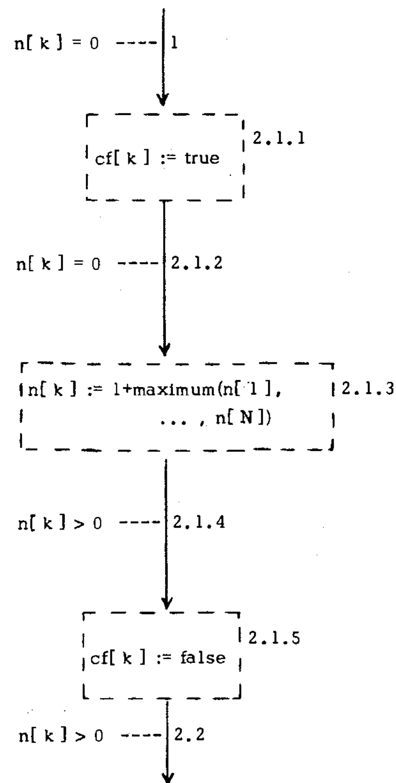


Fig. 7. Stage 4 decomposition of subroutine 2.1.

At this stage, we have described our program to the same

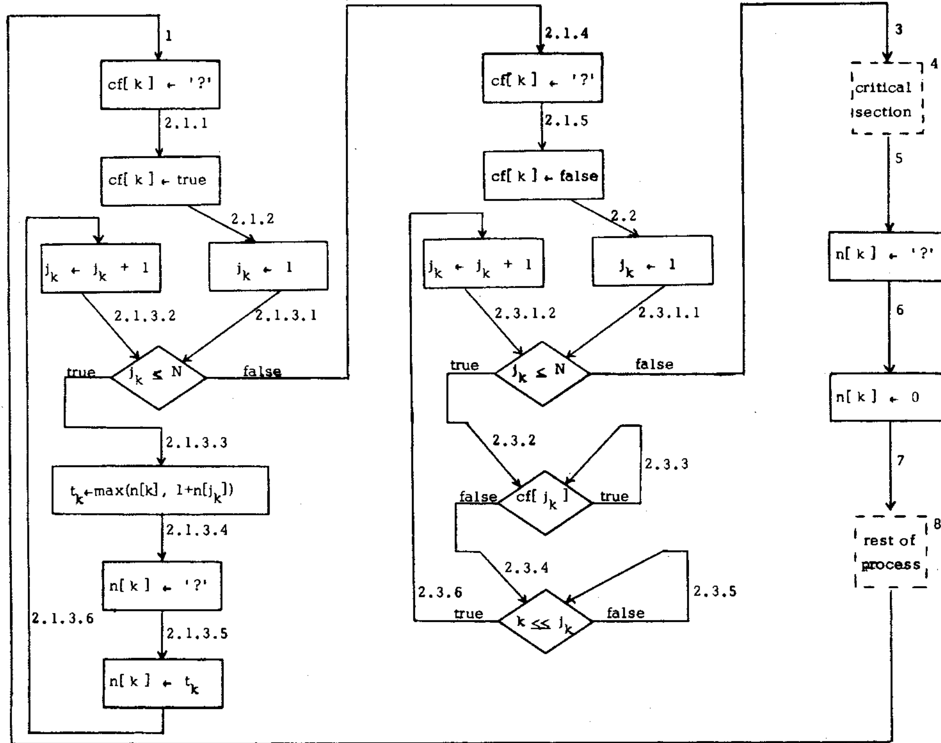


Fig. 8. The final program.

level of detail as its Algol representation in [13], and have essentially designed an informal proof of correctness. This proof involves showing that our "partial interpretation" of Π is invariant, and is done by proving consistency at each subroutine node and monotonicity of each assertion attached to Π_k under every other process Π_j. Theorems 2 and 4 imply that to verify the monotonicity conditions, it suffices to prove that R_{kj} is monotone under subroutine 6, and R_{kj} and E_{kj} are monotone under subroutine 2.1.3 of Π_j. Note that there are just three monotonicity conditions to be proved, despite the fact that the 11 arcs and 9 nodes of our flowchart imply 99 monotonicity conditions.

This informal proof of correctness must be done in order to check our design and make sure that it can be completed to a correct program. It is very similar to the proof of Assertion 2 of [13]. The latter proof involves sequences of events. We have essentially translated its ideas into assertions which must be true at different points in the flowchart.

At this stage, the further refinement of our design is quite straightforward. Each subroutine can be designed independently of any other subroutines. The final program is shown in Fig. 8. Its initial assertion is

$$\bigwedge\{(\pi_k \in \boxed{8}) \wedge (n[k] = 0) \wedge (\sim cf[k]) : 1 \leq k \leq N\}.$$

Observe that setting the value of $cf[k]$ or $n[k]$ is represented by a two-step procedure. The variable is first set to the transient value '?'. We assume that if the variable is read by another process while its value is '?', then the read operation may return any integral value. Hence, any function of these variables is a multivalued function, and our program is nondeterministic.

This program represents an implementation in which the operations of reading and writing a variable may overlap, in which case no assumption is made about the value returned by the read. We assume that concurrent read operations to a variable do not interfere with one another. Concurrent write operations are impossible in this program.

Note that if $n[j]$ = '?', then the expression $n[j]$ assumes all integral values. Hence, the *predicate* $n[j] > 0$ (which is a subexpression of the predicate $k \ll j$) is multivalued and equals both *true* and *false*. However, the *assertion* $n[j] > 0$ is false, as is the assertion $n[j] = 0$.

The program still has flowchart nodes which require several operations when implemented by any real processor—for example, the "$k \ll j_k$" decision node. However, the implementation of each node requires at most one read to one variable which can be set by another process. It is easy to see that this allows us to consider these nodes to be indivisible operations.

We can now define the assertion T_{kj} as follows:

$$\begin{aligned} T_{kj} \equiv \;& ([(\pi_j \in \{2.1.3.2, 2.1.3.3\}) \wedge (j_j > k)] \\ & \supset [n[j_j] > n[k]]) \wedge ([(\pi_j \in \{2.1.3.4, 2.1.3.5\}) \\ & \wedge (j_j \geq k)] \supset [t_j > n[k]]) \wedge ([(\pi_j = 2.1.3.6) \\ & \wedge (j_j \geq k)] \supset [n[j_j] > n[k]]). \end{aligned}$$

The complete interpretation is defined by attaching assertions to the arcs as indicated below:

arcs 1, 2.1.1, 2.1.2, 7, 8: $n[k] = 0$
arcs 2.1.3.1, 6: *true*
arcs 2.1.3.4, 2.1.3.5: $(j_k \geq k) \supset (t_k > 0)$

arcs 2.1.3.2, 2.1.3.3: $(j_k > k) \supset (n[k] > 0)$
arc 2.1.3.6: $(j_k \geqslant k) \supset (n[k] > 0)$
arcs 2.1.4 - 2.3.1.1: $n[k] > 0$
arcs 2.3.1.2 - 2.3.3: $(n[k] > 0) \wedge \bigwedge\{E_{kj} : 1 \leqslant j < j_k\}$
arcs 2.3.4, 2.3.5: $S_{kj_k} \wedge \bigwedge\{E_{kj} : 1 \leqslant j < j_k\}$
arc 2.3.6: $\bigwedge\{E_{kj} : 1 \leqslant j \leqslant j_k\}$
arcs 3 - 5: $\bigwedge\{E_{kj} : 1 \leqslant j \leqslant N\}$.

It is a straightforward task to prove the invariance of this interpretation. We must assume that subroutines 4 and 8 of Π_k do not change the values of n, cf, j_i, or t_i for $i \neq k$. Proving the required consistency conditions is then tedious but quite simple, except for the consistency of the "$cf[j_k]$" decision node. For this, we must first prove that for all k, the assertion $\sim cf[k] \supset (\pi_k \notin \boxed{2.1.3})$ is invariant. This is easily done by proving that attaching this assertion to all arcs of the flowchart gives an invariant interpretation.

Note that Theorem 2 reduces the number of monotonicity conditions to be verified from a possible 418 (22 arcs, 19 nodes) to only 11.

Discussion of the Proof

Our example showed how the program and the proof can be designed together in a top-down hierarchical fashion. At each stage, our proof required that the following conditions be satisfied by each subroutine: 1) the assertions attached to its input and output arcs are consistent, and 2) certain assertions are monotone under that subroutine. We then checked that our design was correct by showing that the informally defined subroutines satisfied these conditions. We can also regard these conditions as part of the formal specification of the subroutine. This idea can be developed into a design methodology for large programming projects, but it is beyond the scope of this paper to do so.

Observe that much of the bookkeeping in our design procedure can be automated–for example, keeping track of the monotonicity conditions. Automatic theorem provers can also do most of the work in proving the invariance of the final interpretation, since most of the consistency conditions that must be proved are very simple. However, designing the interpretation required considerable human ingenuity. Like most simple proofs, this one was hard to make simple.

Designing multiprocess programs and proving them to be correct is a difficult art. We do not claim that our method makes it easy. With this example, we have tried to show that the method makes formal correctness proofs feasible. Using automated aids to handle the simple drudgery, constructing and verifying this formal proof would not have been much harder than writing a careful informal proof.

The reader may feel that our proof was too difficult for such a short program. However, although it is short, the algorithm is quite subtle. This is indicated by the fact that it implements mutual exclusion without assuming any hardware implemented mutual exclusion. Hence, even so simple an assertion as $n[k] = 0$ turned out to require careful definition.

Its subtlety means that the algorithm is more complex than it appears. If we were to try to prove it correct by a brute force analysis of all possible program states, then its complexity would manifest itself in the enormous number of cases we would have to consider. For example, not only can the value of $n[k]$ become arbitrarily large, but $n[j]$, $n[k]$, and $n[j] - n[k]$ can all become arbitrarily large if $j \neq k$ and $N > 3$. (We do not know if this is true for $N = 3$.)

We have spent considerable time studying this algorithm, trying to find an easy way to modify it so $n[k]$ need only assume a finite number of values. We have been consistently frustrated by the amazing number of possible execution sequences. This experience has led us to conclude that it would be impractical to try to prove the algorithm correct by exhaustive case analysis.

Another Property of the Program

There is one safety property of the bakery algorithm proved in [13] which we have not mentioned: processes enter their critical sections on a first come, first served basis. More precisely, if Π_k reaches arc 2.2 before Π_j enters subroutine 2.1, then Π_k will enter its critical section before Π_j does. For the sake of completeness, we now indicate how this is proved.

The correctness property is phrased in terms of sequences of events, so it cannot be directly expressed as the invariance of an assertion. The obvious method is to introduce some fictitious variables, including a "clock" which is incremented each time it is read. A process reads the "clock" at the beginning and end of subroutine 2.1, and by using the "times" read we can define an assertion whose invariance implies the desired property.

However, there is an easier way. It suffices to prove that if we reach a state with Π_k's token on arc 2.2 and Π_j's token between arcs 7 and 1, then Π_k must reach arc 6 before Π_j can enter its critical section. To do this, we simply delete arc 6 from Π_k and show that then Π_j can *never* enter its critical section. Let us therefore define a new program in which all processes except Π_k are the same as before, and Π_k consists only of subroutines 2.3 and 4. We define an interpretation of this new program by attaching the same assertions to the arcs as before. The initial assertion is that $\pi_k = 2.2$, $\pi_j \in \boxed{7} \cup \boxed{8} \cup \boxed{1}$, and every process' token is on an arc whose assertion is true.

The invariance of the interpretation is an easy consequence of the invariance of the interpretation of our original program and our initial assertion. The invariance of the assertion E_{kj} then follows easily from Theorem 3. Using the fact that $\vdash [(\pi_j \in \boxed{4}) \supset \sim(E_{kj} \wedge E_{jk})]$ (since Π_k no longer changes $n[k]$), we can prove that $\Vdash [\pi_j \notin \boxed{4}]$, which is the desired result.

III. Liveness

Introduction

We now turn to the proof of liveness properties of programs. Recall that a liveness property is one which states that something *must* happen. For example, in the bakery algorithm we want to prove that if a process' token reaches arc 1, then that process will eventually enter its critical section.

We defined an execution of a flowchart program to be a se-

quence of steps, where each step consists of moving a single process' token from one arc through a node to another arc (possibly the same one). A process is halted if its token reaches an exit arc. So far, we have allowed executions which can prevent anything from happening—for example, executions of the bakery program in which some process' token sits forever on an arc of its critical section. Obviously, no other process will ever enter its critical section in such an execution. To prove liveness properties, we must rule out executions which "freeze" a process in this way.

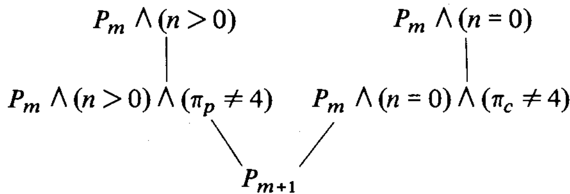
Let a legitimate execution be one in which every process that does not halt has its token moved infinitely many times. For assertions A and B, we let $A \rightsquigarrow B$ mean that if a legitimate execution reaches a state in which A is true, then it will subsequently reach a state in which B is true. (The assertion B might later become false.) To prove liveness properties, we will prove that $A \rightsquigarrow B$ for the appropriate assertions A and B. Note that the relation $\rightsquigarrow$ is transitively closed ($A \rightsquigarrow B$ and $B \rightsquigarrow C$ imply $A \rightsquigarrow C$) and reflexive ($A \rightsquigarrow A$).

The Producer/Consumer Program

To illustrate our method, we return to the producer/consumer program of Fig. 1. We will show that no deadlock can occur, so the producer and consumer never stop sending and receiving messages. More precisely, we prove that ms and mr become arbitrarily large. Since we have already proved that $\Vdash [0 \leqslant ms - mr \leqslant b]$, it suffices to prove that $ms + mr$ becomes arbitrarily large.

Let P_m denote the assertion $ms + mr = m$. We will prove that $A_0 \rightsquigarrow P_m$ for every nonnegative integer m. Since P_0 is true initially, by the transitivity of the $\rightsquigarrow$ relation it suffices to prove that $P_m \rightsquigarrow P_{m+1}$ for all m.

Consider the following diagram.



where π_p and π_c denote the locations of the producer's and consumer's tokens, respectively. Suppose that for each assertion in this diagram, we can show that if it is true, then one of the assertions beneath it must subsequently become true. Then this implies that $P_m \rightsquigarrow P_{m+1}$ because if P_m is true, then one of the two top assertions $P_m \wedge (n > 0), P_m \wedge (n = 0)$ must be true.

We have split the problem of proving $P_m \rightsquigarrow P_{m+1}$ into proving four other $\rightsquigarrow$ relations: one for every assertion in the diagram except P_{m+1}. We illustrate the method by proving one of these relations—namely, $[P_m \wedge (n > 0) \wedge (\pi_p \neq 4)] \rightsquigarrow P_{m+1}$. To do this, we assume that $P_m \wedge (n > 0) \wedge (\pi_p \neq 4)$ is initially true, but that P_{m+1} never becomes true, and we obtain a contradiction.

We first generalize our definition of an interpretation to one which attaches two assertions to a single arc: an input assertion which should be true if the token is initially placed on the arc, and an output assertion which should be true if the token is moved onto the arc during the program execution. The basic idea is to find such an interpretation satisfying the following conditions:

1)
   a) If $P_m \wedge (n > 0) \wedge (\pi_p \neq 4)$ is true, then the token of each process must lie on an arc whose input assertion is true.
   b) The invariance of $\sim P_{m+1}$ implies that this interpretation is consistent.
2) The consumer's token must eventually reach an arc whose output assertion is the constant assertion *false*.

This yields the necessary contradiction, since 1) implies that each process' token is always on an arc whose assertion is true, and 2) implies that the consumer's token must reach an arc whose assertion is false.

We will use the interpretation shown in Fig. 9, where Q denotes the assertion $P_m \wedge (n > 0) \wedge (\pi_p \neq 4)$. All arcs have their input and output assertions equal, except for arcs 1 and 4 of the consumer. Each of those arcs has the input assertion Q and the output assertion *false*.

It is easy to see that this interpretation satisfies condition 1a). We next show that it satisfies 1b). The consistency of the producer's interpretation is easy to verify. Note that the proof

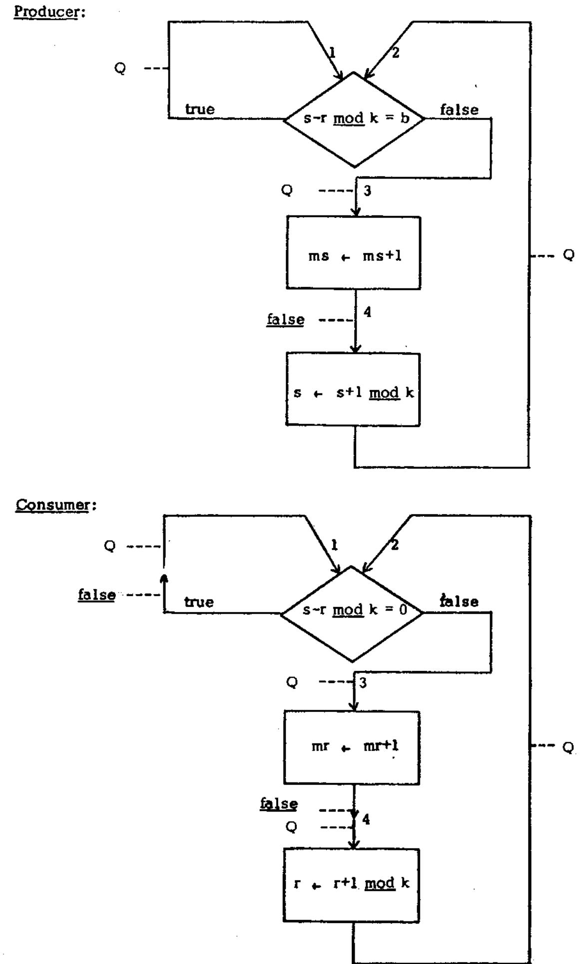


Fig. 9. Generalized interpretation of producer/consumer program.

of the consistency at the *ms* assignment node uses the hypothesis that $\sim P_{m+1}$ is invariant, i.e., the producer's token cannot reach arc 4 unless P_{m+1} becomes true. To prove the consistency of the consumer's interpretation, the only nontrivial condition to verify is the one for the decision node. We first observe that the invariance of the interpretation of Fig. 1 implies that the assertion $(\pi_p \neq 4) \supset (n_c \leqslant s - r \bmod k)$ is invariant. Using this and the fact that $(\pi_c \in \{1, 2\}) \supset (n = n_c)$, we can easily prove consistency at the consumer's decision node.

Proving the monotonicity conditions for the attached assertions is now trivial. Hence, condition 1b) is satisfied. Condition 2) is clearly satisfied, since every loop in the consumer's flowchart contains arc 1 or 4. Thus, our proof that $Q \leadsto P_{m+1}$ is completed.

We have proved one of the required four $\leadsto$ relations for the assertion diagram. The proofs of the remaining three relations are similar.

The Formalism

We now formalize the method used in the above example.

Definition 6: A *generalized interpretation* I_k of a process Π_k is a mapping which assigns to each arc $\alpha \in \Gamma_k$ an *input* Π_k-*assertion* ${}_iI_k^\alpha$ and an *output* Π_k-*assertion* ${}_oI_k^\alpha$ such that $\vdash [{}_oI_k^\alpha \supset {}_iI_k^\alpha]$. The definitions of consistency and monotonicity are the same as in Definition 3, except that in the consistency conditions for a node, the input assertions are used for the node's input arcs and the output assertions for its output arcs.

A generalized interpretation I of a program Π consists of a generalized interpretation I_k of each process Π_k. It is *consistent* if: i) each I_k is consistent, and ii) for each j and k with $j \neq k$ and every $\beta \in \Gamma_j : {}_iI_j^\beta \wedge (\pi_j = \beta)$ is monotone under I_k. □

We will only need generalized interpretations for which each ${}_oI_k^\alpha$ equals either ${}_iI_k^\alpha$ or *false*.

Definition 7: A set of arcs Λ of a process Π is called an *inevitable* set if for some process Π_k: i) every closed path in Π_k contains an arc in Λ, and ii) Λ contains all the exit arcs of Π_k. □

We now state some simple axioms for the relation $\leadsto$. We could define $\leadsto$ in terms of executions as indicated before, and then prove that it satisfies the axioms. As with S1 and S2, we will not bother to do this, but will consider our axioms to define the relation $\leadsto$.

L1: Let A and B be assertions about a program Π, and let I be a generalized interpretation of Π. *If*

1) by assuming $\Vdash [\sim B]$ we can prove that
   a) $\Vdash [A \supset \bigwedge \{(\pi_k = \alpha) \supset {}_iI_k^\alpha : \text{all } k, \text{and all } \alpha \in \Gamma_k\}]$
   b) I is consistent
2) $\{\alpha : {}_oI_k^\alpha \equiv \mathit{false}\}$ is an inevitable set.

Then $A \leadsto B$. □

L2:

a) The relation $\leadsto$ is transitively closed.

b) If $\mathcal{A}$ is a [finite] set of assertions, and $A \leadsto B$ for each $A \in \mathcal{A}$, then $\bigvee \mathcal{A} \leadsto B$. □

We have indicated two possible choices for axiom L2, depending upon whether or not $\mathcal{A}$ is assumed to be finite. These two choices give slightly different meanings to the relation $\leadsto$. This will be discussed later. In practice, one will seldom use L2 for infinite sets of assertions.

Axiom L1 is stated in terms of a proof by contradiction. It might be more elegant to restate the axiom in terms of a positive proof. However, proofs by contradictions seem to be easier in practice. Rather than considering all the things that *might* happen, we can simply postulate that none of them do happen and obtain a contradiction.

The following theorems are consequences of the axioms, and are useful in proofs. We will not bother to prove them.

Theorem 5: If $\Vdash [A \supset B]$, then $A \leadsto B$. □

Theorem 6: If $\Vdash [\sim C]$ implies $A \leadsto B$, then $A \leadsto B \vee C$. □

Theorem 7: If C is monotone and $\Vdash [C]$ implies $A \leadsto B$, then $A \wedge C \leadsto B \wedge C$. □

The following definition abstracts the property of the positive integers which is used in "counting down" proofs.

Definition 8: Let $\mathcal{A}$ be a set with an irreflexive partial ordering $<$. For any $A \in \mathcal{A}$, we define $\mathcal{A}_A$ to be $\{B \in \mathcal{A} : B < A\}$. The set $\mathcal{A}$ is called *well-founded* if for each $A \in \mathcal{A}$, $\mathcal{A}_A$ is a finite set. □

The following theorem generalizes the "assertion diagram" method used in the producer/consumer example. It is an easy consequence of L2, and its proof is omitted. The hypothesis that $\mathcal{A}$ is finite is needed if only the weaker form of L2 is assumed. Note that any finite partially ordered set is well-founded.

Theorem 8: Let $\mathcal{A}$ be a [finite] well-founded set of assertions, and B any assertion. If $A \leadsto B \vee \bigvee \mathcal{A}_A$ for all $A \in \mathcal{A}$, then $\bigvee \mathcal{A} \leadsto B$. □

Proofs of liveness properties can be designed in a hierarchical manner along with the program. This is done by designing the set of assertions $\mathcal{A}$ of Theorem 8 by successive refinement. With each step of the design process, the elements of $\mathcal{A}$ are decomposed as the union of other assertions. This is made more precise by the following definition.

Definition 9: Let $\mathcal{A}$ and $\mathcal{A}'$ be partially ordered sets of assertions. We say that $\mathcal{A}'$ is a *refinement* of $\mathcal{A}$ if there exists a surjective, order-preserving, single-valued mapping $\pi : \mathcal{A}' \to \mathcal{A}$ such that $A = \bigvee \pi^{-1}(A)$ for each $A \in \mathcal{A}$. □

It is easy to see that the refinement $\mathcal{A}'$ satisfies the hypothesis of Theorem 8 only if $\mathcal{A}$ does. Hence, we can design a formal proof by successively refining informal proofs.

The Bakery Program

To illustrate the hierarchical design of a liveness proof, we return to the bakery program and its design process shown in Figs. 4-8. The basic liveness property we want to prove is that any process which reaches arc 1 of its flowchart must eventually enter its critical section. Since we have already proved that processes enter their critical sections on a first come, first served basis, we will simplify our task by proving the following property: if some process is in subroutine 2, then some process must subsequently be in its critical section.

Before doing this, we introduce some notation. For any set Λ of arcs, we let $\gamma(\Lambda) = \{k : \pi_k \in \Lambda\}$. For example, $\gamma(\boxed{4})$ represents the set of processes in their critical sections. If m and n are arc numbers, then $\boxed{m:n}$ denotes the set of all arcs

numbered from m through n. Referring to Fig. 7, we have $\boxed{1:2.1.3} = \{1\} \cup \boxed{2.1.1} \cup \{2.1.2\} \cup \boxed{2.1.3}$. We also let $\overline{N}$ denote $\{1, \cdots, N\}$.

We can now state our desired liveness property as follows: $[\gamma(\boxed{1:2}) \neq \phi] \leadsto [\gamma(\boxed{3:4}) \neq \phi]$. The method of proof is to construct an appropriate set $\mathfrak{A}$ of assertions with $\bigvee \mathfrak{A} \equiv [\gamma(\boxed{2}) \neq \phi]$, and then apply Theorem 8. By Theorem 6, it suffices to assume that $\Vdash [\gamma(\boxed{3:4}) = \phi]$ and then prove that $A \leadsto \bigvee \mathfrak{A}_A$ for every $A \in \mathfrak{A}$. Therefore, we will assume throughout this proof that $\Vdash [\gamma(\boxed{3:4}) = \phi]$.

We first consider the stage 1 decomposition of the program shown in Fig. 4. Since a process in subroutine 6 might prevent any other process from entering its critical section, we want to show that all processes must eventually leave subroutine 6. To do this, we make the following definitions:

$$A(S) \equiv [\gamma(\boxed{1:2}) \neq \phi] \wedge [\gamma(\boxed{5:6}) \subset S], \text{ for } S \subset \overline{N}$$

$$\mathfrak{A}^{(1)} = \{A(S) : S \subset \overline{N}\}$$

$$A(S) < A(S') \text{ iff } S \subsetneq S'.$$

Lemma 1:

a) For each k: the assertions $\pi_k \in \boxed{1:2}$ and $\pi_k \notin \boxed{5:6}$ are monotone.

b) For all $S \subset \overline{N}$: the assertion $A(S)$ is monotone.

Proof:

a) The monotonicity of each of these assertions is easily proved by attaching it to every arc of the flowchart, and then using the assumption that $\Vdash [\gamma(\boxed{3:4}) = \phi]$ to prove that this interpretation is consistent.

b) This follows from part a) and Theorem 4, since $A(S) \equiv \bigvee\{(\pi_k \in \boxed{1:2}) : k \in \overline{N}\} \wedge \bigwedge\{\pi_k \notin \boxed{5:6} : k \notin S\}$. □

Lemma 2: Assume that for each k: $[\pi_k \in \boxed{5:6}] \leadsto [\pi_k = 7]$. If $S \neq \phi$, then $A(S) \leadsto A(S')$ for some $S' \subsetneq S$.

Proof: Let $k \in S$. By definition of $A(S)$, we have $\vdash [(A(S) \wedge A(S - \{k\})) \supset (\pi_k \notin \boxed{5:6})]$ and $\vdash [(A(S) \wedge \sim A(S - \{k\})) \supset (\pi_k \in \boxed{5:6})]$. Applying Theorem 5, the hypothesis, and L2 b), we easily show that $A(S) \leadsto [\pi_k \notin \boxed{5:6}]$. Since $A(S) \wedge [\pi_k \notin \boxed{5:6}] \equiv A(S - \{k\})$, Lemma 1 b) and Theorem 7 imply $A(S) \leadsto A(S - \{k\})$. □

Note how defining $A(S)$ by the condition $\gamma(\boxed{5:6}) \subset S$ rather than $\gamma(\boxed{5:6}) = S$ simplified the proof of Lemma 2. When trying to apply Theorem 8, it is often easier if we define $\mathfrak{A}$ so that $A < A'$ implies $A \supset A'$.

To use Lemma 2, we will have to show later that $[\pi_k \in \boxed{5:6}] \leadsto [\pi_k = 7]$. This must wait until a later stage of the design when subroutine 6 is more fully specified. In fact, this condition can be considered to be part of the specification of subroutine 6.

Since $\bigvee \mathfrak{A}^{(1)} \equiv [\gamma(\boxed{1:2}) \neq \phi]$, Lemma 2 shows that to complete our proof, we need only prove that $A(\phi) \leadsto$ *false*. To do this, we must decompose $A(\phi)$. Using the idea that either a process must enter its critical section or else another process must enter subroutine 2, we make the following definitions:

$$B(T) \equiv A(\phi) \wedge [T \subset \gamma(\boxed{1:2})]$$

$$\mathfrak{A}^{(2)} = \{A(S) : S \neq \phi\} \cup \{B(T) : T \neq \phi\}$$

$$B(T) < A(S) \text{ for all } S, T$$

$$B(T') < B(T) \text{ iff } T \subsetneq T'.$$

Since $A(\phi) = \bigvee\{B(T) : T \neq \phi\}$, we need only prove that $B(T) \leadsto \bigvee\{B(T') : T \subsetneq T'\}$ for all $T \neq \phi$. This will require a further decomposition of $B(T)$.

We next turn to the stage 2 decomposition of the program shown in Fig. 5. We expect any process in subroutine 2.1 to leave that subroutine and enter subroutine 2.3. This leads us to make the following definitions.

$$B(T, U) \equiv B(T) \wedge [U \subset \gamma(\boxed{2.2:2.3})]$$

$$\mathfrak{A}^{(3)} = \{A(S) : S \neq \phi\} \cup \{B(T, U) : U \subset T \text{ and } T \neq \phi\}$$

$$B(T, U) < A(S) \text{ for all } S, T, U$$

$$B(T', U') < B(T, U) \text{ iff : i) } T \subsetneq T' \text{ or ii) } T = T' \text{ and } U \subsetneq U'.$$

Lemma 3: For all T, U with $U \subset T$: $B(T, U)$ is monotone.

Proof: The proof is similar to that of Lemma 1 b). We first prove that $[\pi_k \in \boxed{2.2:2.3}]$ is monotone for all k, then express $B(T, U)$ in terms of these assertions and the ones of Lemma 1 a). The details are left to the reader. □

Lemma 4: Assume that for all k: $[\pi_k \in \boxed{1:2.1}] \leadsto [\pi_k = 2.2]$. If $U \subsetneq T$, then $B(T, U) \leadsto B(T, U')$ for some U' with $U \subsetneq U'$.

Proof: Let $k \in T - U$. Then Lemma 3, Theorem 7, and the hypothesis imply $[\pi_k \in \boxed{1:2.1}] \wedge B(T, U) \leadsto [\pi_k = 2.3] \wedge B(T, U)$. But $[\pi_k \in \boxed{1:2.1}] \wedge B(T, U) \equiv B(T, U)$ and $[\pi_k = 2.2] \wedge B(T, U) \supset B(T, U \cup \{k\})$. Hence, Theorem 5 and L2 a) imply $B(T, U) \leadsto B(T, U \cup \{k\})$. □

The condition $[\pi_k \in \boxed{1:2.1}] \leadsto [\pi_k = 2.2]$ must be proved later, and becomes part of the specification of subroutine 2.1. All that remains to be shown is that $B(T, T) \leadsto \mathfrak{A}^{(3)}_{B(T,T)}$. This requires decomposing the assertions $B(T, T)$. The essential idea is that if $\gamma(\boxed{2.2:2.3}) = T$ and no more processes enter subroutine 2, then the process Π_k with $k \lll j$ for all $j \in T$ must enter its critical section. We therefore make the following definitions.

$$B(T, T, k) \equiv B(T, T) \wedge \bigwedge\{k \lll j : j \in T\}$$

$$\mathfrak{A}^{(4)} = \{A(S) : S \neq \phi\} \cup \{B(T, U) : U \subsetneq T\} \cup \{B(T, T, k) : T \neq \phi, k \in T\}.$$

$$A \lesssim B(T, T, k) \text{ iff } A \lesssim B(T, T), \text{ for any } A \in \mathfrak{A}^{(4)}.$$

Observe that $T \neq \phi$ implies $B(T, T) \equiv \bigvee\{B(T, T, k) : k \in T\}$ since $\bigwedge\{k \lll j : j \in T\}$ must be true for some $k \in T$. Therefore, $\mathfrak{A}^{(4)}$ is a refinement of $\mathfrak{A}^{(3)}$.

Lemma 5: For each $T \neq \phi$, $k \in T$: $\Vdash [\sim \bigvee \mathfrak{A}^{(4)}_{B(T,T,k)}]$ implies that

a) $B(T, T, k)$ is monotone

b) $\Vdash [B(T, T, k) \supset (\alpha(\boxed{1:2.1}) = \phi) \wedge \bigwedge\{k \lll j : j \in \overline{N}\}]$.

Proof: Let $Q \equiv \sim \bigvee \mathfrak{A}^{(4)}_{B(T,T,k)}$. Then $Q \equiv \bigwedge\{\sim B(T', U') : T \subset U' \subset T' \text{ and } T \neq T'\}$. Therefore, $Q \wedge B(T, T, k)$ implies $\gamma(\boxed{2.2:2.3}) = \gamma(\boxed{1:2}) = T$, so $\gamma(\boxed{1:2.1}) = \phi]$. Also, $B(T, T, k)$ implies $\gamma(\boxed{5:6}) = \phi$. Since we are assuming $\Vdash [\gamma(\boxed{3:4}) = \phi]$, we have therefore proved that $\Vdash [Q]$ implies

$$(*) \quad \Vdash [B(T, T, k) \supset (\gamma(\boxed{1:2.1}) = \gamma(\boxed{3:6}) = \phi)].$$

It is easy to prove part a) by attaching $B(T, T, k)$ to each arc of the flowchart, and then using (*) to prove that this interpretation is consistent. To prove part b), we first observe that $\Vdash [B(T, T, k) \supset (n[k] > 0)]$ and $\Vdash [(\pi_j \notin \boxed{2:6}) \supset (n[j] = 0)]$. Since $\vdash [(n[k] > 0) \wedge (n[j] = 0) \supset (k \leqslant\!\!\leqslant j)]$, part b) follows easily from (*). □

Lemma 6: Assume that for all k: if $\Vdash [(\gamma(\boxed{1:2.1}) = \phi) \wedge \bigwedge\{k \leqslant\!\!\leqslant j : j \in \overline{N}\}]$, then $[\pi_k \in \boxed{2.2:2.3}] \leadsto [\pi_k = 3]$. Then for all $T \neq \phi$ and $k \in T$: $B(T, T, k) \leadsto \bigvee \mathcal{A}^{(4)}_{B(T,T,k)}$.

Proof: By Theorems 6 and 7 and Lemma 5, it suffices to assume that $\Vdash [\sim \bigvee \mathcal{A}^{(4)}_{B(T,T,k)}]$ and $\Vdash [B(T, T, k)]$, and then prove that $B(T, T, k) \leadsto$ *false*. Lemma 5 then implies that $\Vdash [\bigwedge\{k \leqslant\!\!\leqslant j : j \in \overline{N}\}]$. We now have

$B(T, T, k) \leadsto [\pi_k \in \boxed{2.2:2.3}]$ (by Theorem 5)

$[\pi_k \in \boxed{2.2:2.3}] \leadsto [\pi_k = 3]$ (by hypothesis)

$[\pi_k = 3] \leadsto$ *false* (by Theorem 5 and $\Vdash [\gamma(\boxed{3:4}) = \phi]$).

By the transitivity of $\leadsto$, this proves the lemma. □

Combining Lemmas 2, 4, and 6, we see that Theorem 8 applied to $\mathcal{A}^{(4)}$ proves the desired liveness property—if the following conditions are satisfied:

1) $[\pi_k \in \boxed{5:6}] \leadsto [\pi_k = 7]$
2) $[\pi_k \in \boxed{1:2.1}] \leadsto [\pi_k = 2.2]$
3) $\Vdash [(\gamma(\boxed{1:2.1}) = \phi) \wedge \bigwedge\{k \leqslant\!\!\leqslant j : j \in \overline{N}\}]$ implies $[\pi_k \in \boxed{2.2:2.3}] \leadsto [\pi_k = 3]$.

To prove these conditions, we have to turn to the final program of Fig. 8. The proof of 1) requires a trivial application of L1. The proof of 2) is done with the following sequence of steps:

a) $[\pi_k \in \boxed{1:2.1}] \leadsto [\pi_k \in \boxed{2.1.3}]$
b) $\Vdash [(\pi_k \in \boxed{2.1.3}) \supset \bigvee\{\pi_k \in \boxed{2.1.3}) \wedge (j_k = i) : i \in \overline{N}\} \vee ((\pi_k = 2.1.3.2) \wedge (j_k = N + 1))]$
c) if $1 \leqslant i \leqslant N$, then $[(\pi_k \in \boxed{2.1.3}) \wedge (j_k = i)] \leadsto [(\pi_k = 2.1.3.2) \wedge (j_k = i + 1)]$
d) $[\pi_k = 2.1.3.2] \wedge [j_k = N + 1] \leadsto [\pi_k = 2.2]$.

Step b) is a safety property which is easily proved by the methods described before, and the other steps are easily proved using L1. By L2 and Theorem 5, they prove 2).

The proof of 3) is similar to that of 2), except that it requires the additional step of proving the following trivial safety property:

$$\Vdash [(\gamma(\boxed{1:2.1}) = \phi) \supset \bigwedge\{\sim cf[j] : j \in \overline{N}\}].$$

Discussion of the Proof

Let us consider the informal reasoning underlying the above proof. We must assume that no process is ever in its critical section, and then derive a contradiction. We first observe that this assumption implies that all processes must leave subroutine 6 (Lemmas 1 and 2). Next, we see that we may further assume that no more processes enter subroutine 2.1, so eventually there will be no processes in that subroutine (Lemmas 3 and 4). We cannot suppose all processes will eventually reach subroutine 2, since a process may remain forever in subroutine 8. Finally, we show that if subroutines 2.1 and 6 are empty, then the process Π_k with $k \leqslant\!\!\leqslant j$ for all j must eventually enter its critical section (Lemmas 5 and 6).

This reasoning assumes certain properties of the subroutines 2.1, 2.3, and 6: namely, the properties formalized in conditions 1)-3). The proofs of these conditions were quite straightforward, and were essentially the same as the usual proofs of termination for sequential programs. Of course, the formal proofs that we sketched required a considerable amount of detail for such simple results. However, that was because we had to reduce everything to our fundamental axioms L1 and L2. A few basic theorems—e.g., that a process must eventually exit from a loop-free subroutine—would greatly simplify the proofs. There is no real difficulty in constructing complete formal proofs of conditions 1)-3).

The situation is not quite as satisfactory with the rest of the proof. If we consider Lemmas 1-6 and their proofs to constitute a very rigorous informal correctness proof, then it is doubtful that this proof inspires any more confidence in the program than the simple informal reasoning upon which it was based. We can also consider our proof to be a sketch for a complete formal proof. It is certainly rigorous enough to be formalized, but it would be rather tedious to do so without help from an automated proof verifier. A general-purpose proof verifier would have to be quite sophisticated in order to provide this help.

Fortunately, a sophisticated verifier should not be necessary. Examining the proofs of the lemmas shows that they follow a similar pattern. Further experience constructing liveness proofs should enable us to ritualize the procedure. It would then be possible to design a relatively unsophisticated automated system to aid in constructing the proofs and to verify them.

Observe that we never assumed that a process which enters its critical section must eventually leave it. We very carefully formulated our liveness property so that assumption was unnecessary. Of course, the complete program would undoubtedly be considered incorrect if a process remained forever in its critical section. However, we were really only interested in proving the correctness of subroutines 2 and 6.

Real Time Considerations

We now examine the meaning of the relation $\leadsto$ in terms of program execution times. Let us assume that each process has a run time clock which is advanced by a positive quantity every time one of its nodes is executed. (This quantity may depend upon the node and the values of variables.) Let us also assume a single real time clock which is arbitrarily advanced, but must always run at least as fast as each process' run time clock. This represents our intuitive idea of how processes are actually executed by real machines.

Let us now make the further assumption that there are positive constants $T, T_1, \cdots, T_N$ such that whenever the real time clock advances at least T units of time, each process Π_k's run time clock will advance by at least T_k units. For processes being executed by separate physical processors, this means that the ratio of the execution speeds of any two processors is

bounded. For processes implemented by time-sharing a single processor, this represents a reasonably weak fairness condition on the scheduler. (Processes which are in a "waiting loop" can be considered to be running even though they are not actually executed.) We will call the T_k, T and the execution times of the nodes the *implementation parameters.*

We now define the relation $A \overset{t}{\rightsquigarrow} B$ to mean that if A is true, then B will become true within t units of real time, where $0 \leqslant t \leqslant \infty$. If $t = \infty$, then B will eventually become true, but it could take an arbitrarily long time. We can restate axioms L1 and L2 for the relation $\overset{t}{\rightsquigarrow}$. We can modify L1 by changing its conclusion to $A \overset{t}{\rightsquigarrow} B$, where t is a finite number which is an obvious but complicated function of the topology of the flowcharts, the inevitable set of hypothesis 2), and the implementation parameters. Axiom L2 can be restated as follows:

L2':

a) $A \overset{s}{\rightsquigarrow} B$ and $B \overset{t}{\rightsquigarrow} C$ imply $A \overset{s+t}{\rightsquigarrow} C$.

b) If $A \overset{t_A}{\rightsquigarrow} B$ for all $A \in \mathcal{A}$, then $\bigvee \mathcal{A} \overset{t}{\rightsquigarrow} B$, where $t = sup \; \{t_A : A \in \mathcal{A}\}$.

Note that in part b), if $\mathcal{A}$ is an infinite set, then t can equal ∞ even if each t_A is finite. We also need the additional axiom that $A \overset{t}{\rightsquigarrow} B$ implies $A \overset{t'}{\rightsquigarrow} B$ for all $t' \geqslant t$.

When designing real programs, it is not sufficient to prove that something must eventually happen. One wants to prove that it must happen within some reasonable length of time. By proving the appropriate relation $A \overset{t}{\rightsquigarrow} B$, we prove that what we want to have happen will happen within time t, provided the implementation parameters are satisfied. This method might be useful in designing programs with real time response requirements. However, it is not clear whether such a simple approach can successfully represent the complexities of a real implementation.

The difference between the strong and weak forms of axiom L2 can now be described by the following result. (The weak form assumes $\mathcal{A}$ is finite.)

Metatheorem:

a) $A \rightsquigarrow B$ under the strong form of L2 iff $A \overset{\infty}{\rightsquigarrow} B$.

b) $A \rightsquigarrow B$ under the weak form of L2 iff $A \overset{t}{\rightsquigarrow} B$ for some $t < \infty$. □

A rigorous proof of this result would be long and unrewarding, so we will not bother to prove it. Note that the value of t in part b) depends upon the implementation parameters, but its existence does not.

We will illustrate this metatheorem with the single process program of Fig. 10. We want to prove that if $n > 0$, then the program will eventually halt, i.e., we want to prove that $[n > 0] \rightsquigarrow [\pi = 3]$. It is easy to prove, using the weak form of L2, that for any $i > 0$: $[n = i] \rightsquigarrow [\pi = 3]$. However, we need the strong form of L2 (with $\mathcal{A} = \{n = i : i > 0\}$) to conclude that $[n > 0] \rightsquigarrow [\pi = 3]$. We cannot prove this using only the weak form of L2 because although $n > 0$ implies that the process eventually reaches arc 3, it can take an arbitrarily long time to do so. Hence, $[n > 0] \overset{t}{\rightsquigarrow} [\pi = 3]$ holds only for $t = \infty$.

As we have already stated, the strong form of L2 is rarely

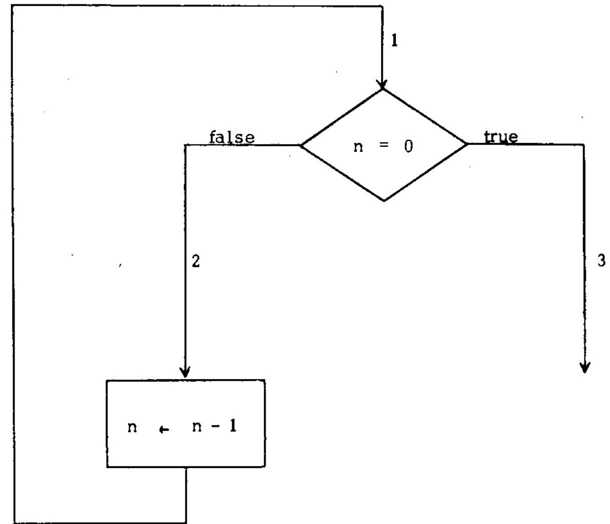


Fig. 10. A single process program.

needed. Even in programs such as our bakery algorithm in which the value set X is infinite ($n[k]$ can become arbitrarily large), we still only need the weak form of L2.

IV. Further Remarks

Applicability

Our multiprocess flowchart programs are sufficiently general to allow a convenient representation of most single-site multiprocess systems.[4] The only apparent lack of generality is the assumption of a fixed number of processes. However, the creation and destruction of processes can be represented by having processes leave and enter waiting loops. Since the actual number of processes need not be specified (our bakery program was proved correct for any N), this is a satisfactory approach. Another approach will be described below.

It seems possible to represent any desired correctness property as either the invariance of an assertion, or a relation of the form $A \rightsquigarrow B$. This may require the introduction of fictitious variables.

Our formalism does not rely upon any particular form of interprocess synchronization. Indeed, we easily represented the bakery algorithm in which no *a priori* synchronization mechanism was assumed. This means that the programmer can choose the type of synchronization most appropriate to his problem, and is not forced to introduce unnecessary synchronizing operations in order to prove the correctness of his program. A good example of a problem that can and must be solved without costly synchronization is given in [8].

Any desired synchronization primitive can easily be represented. For example, Fig. 11 shows one possible representation of the semaphore operations $P(s)$ and $V(s)$ in a process Π_k. Although the formalism requires that the semaphore primitives be represented by busy waiting, they need not actually be implemented in this way. The representation is a formal specification of these primitives. Having to provide

[4]They are probably not suitable for distributed networks of processors.

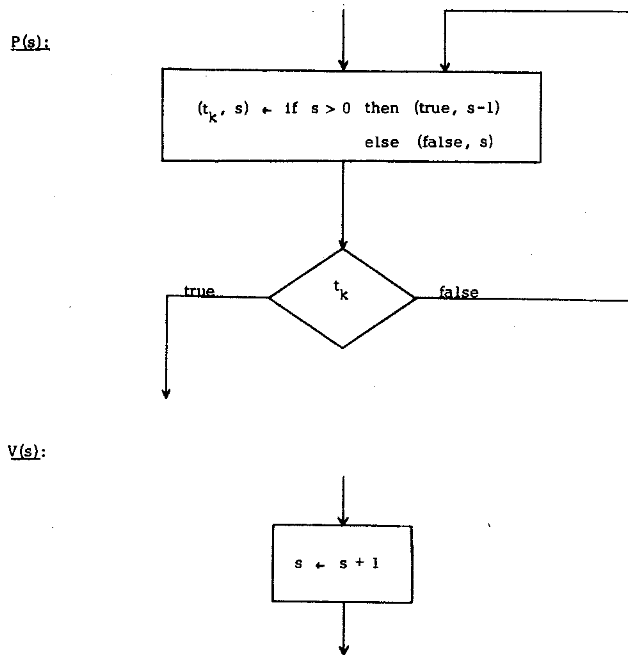


Fig. 11. Representation of the semaphore operations.

such a specification quickly leads one to realize that the usual informal description does not completely define the semaphore primitives.

The method of hierarchical decomposition allows the programmer to prove the correctness of as detailed a description of the program as he desires. To obtain a formal proof, he must reach a level of detail at which each individual node can be considered to be an indivisible operation. Any subroutine which $\left\{\begin{matrix}\text{sets}\\ \text{uses}\end{matrix}\right\}$ no variables which may be concurrently $\left\{\begin{matrix}\text{used}\\ \text{set}\end{matrix}\right\}$ by another process can be considered an indivisible operation. Hence, a single flowchart node may represent a complex operation. For example, in many applications of the mutual exclusion problem, the critical section can be represented by a single assignment node.

A subroutine which can be represented as an indivisible operation may also be further decomposed. By Theorem 2, the proof of correctness of this decomposition becomes independent of the rest of the program, and is essentially the same as the correctness proof for a sequential program as described in [9]. The decomposition can be carried down to the lowest level of detail at which a process maintains its identity. This level is usually that of the individual instruction in some programming language.

When proving the correctness of a program, there are two things that must be verified which are external to our formalism:

1) The flowchart program must be a correct representation of the actual program.

2) The formal correctness properties that we prove really do imply that the program will exhibit the desired behavior.

We may view the total system as a hierarchy of virtual machines. The highest level machine is the one seen by the user; the lowest level one consists of the logical design of the hardware. Each level of the hierarchy is a separate multiprocess program which implements one virtual machine in terms of the next lower level one. Condition 1) states that the program is correctly implemented by the lower level virtual machine, and condition 2) states that it correctly implements the next higher level machine. Hence, 1) and 2) state the correctness of the "coupling" of the program to the other levels of virtual machines. We know of no satisfactory way to formally prove the correctness of this coupling, although a first step has been taken in [3]. For now, this must be proved informally. In any case, correctness of the lowest level coupling (the correct implementation of the hardware design) and of the highest level coupling (the satisfaction of the user's *desires*) can never be formally proved.

An Experiment

Upon discussing our method with others, we were asked how long it takes actually to construct a correctness proof. Our experience with the above proofs did not provide an answer for two reasons: we were already quite familiar with the algorithms before we started, and we worked very hard to make the proofs as simple and elegant as possible. We therefore decided to construct correctness proofs for the solutions to the problem given by Courtois, Heymans, and Parnas in [7]. These were algorithms we had only casually read before.

We first proved the fundamental safety properties of the algorithm. We did this by writing down the appropriate interpretation, and informally checking the conditions needed to prove its invariance. (Essentially the same proof works for both of their solutions.) Starting from when we began examining the algorithms, this took about $1\frac{3}{4}$ hours.

We next sketched a proof of the following liveness property of their second solution: if a writer wants to write, then some writer will eventually enter its writing section.[5] The level of detail in this sketch was analogous to specifying the set of assertions $\mathcal{A}^{(4)}$ and the statements of Lemmas 1-6 and conditions 1)-3) in our liveness proof for the bakery algorithm, but not writing down the proofs of the lemmas or the conditions. Starting from when we chose our informal liveness property, this took about $1\frac{1}{2}$ hours.

Of course, most people would take longer to construct these proofs than we did. One can expect to take at least twice as long in his first attempt at this type of formal proof. He should not even make the attempt until he is sure he could write a rigorous informal proof. However, these times do indicate what can be expected from one who is experienced at writing informal proofs and has some practice with our formal method.

Extensions to the Method

There are two ways to extend our method: by specialization and by generalization. We begin with specialization. Our formalism placed no restriction on the value set X. A useful extension would be to formally define some sort of structure on S. For example, we can assume that X is a product of the

[5] This requires a different definition of the semaphore operations than the one described in Fig. 11.

value sets of the component variables. We could then formally define local variables. Such a structure would make it possible to give a simple formal statement of Theorem 2.

Another reason for placing a structure on X is to allow us to formalize the concept of a closed subroutine. This would enable one to use a library subroutine without having to prove it correct every time it is used. The specification of the subroutine would contain certain theorems which would permit a formal correctness proof for the program without having to decompose the subroutine. These theorems would be of two kinds: i) safety properties stated by sufficient conditions on the assertions to guarantee the existence of a consistent interpretation of the subroutine, and ii) liveness properties stating conditions which guarantee termination. For example, the $P(s)$ semaphore operation could be represented by a subroutine whose specification consists of the following two theorems: i) a consistent interpretation is obtained if the output assertion is implied by the conjunction of $s \geqslant 0$ and the input assertion with $s-1$ substituted for s, and ii) if $s > 0$ and some process is inside a $P(s)$ subroutine, then some process will eventually exit from a $P(s)$ subroutine. By defining an appropriate structure on X, these ideas can be formalized.

Another useful specialization would be to restrict the type of flowcharts allowed. Arbitrary flowchart processes were the natural setting for describing the formalism. However, progress in structured programming has shown that they are not the best way to think about programs. Of course, our method of designing by successive refinement is the essence of structured programming. However, it would be better to have another method of representing processes which would more forcefully encourage a properly structured design, and would exhibit the structure in the final program. Such a representation just requires a change of syntax, and is irrelevant to our basic formalism. However, this does not diminish its importance for a practical programming tool.

We now consider some possible generalizations of our formalism. First, it would be trivial to combine our two types of flowchart nodes into a single more general one: an assignment node having an arbitrary number of exit arcs. This would allow the $P(s)$ operation in Fig. 11 to be represented by a single node with two input and two output arcs. The necessary modification to our definitions should be obvious.

Another possible generalization is to allow recursive subroutines. A recursive subroutine is represented by a flowchart containing one or more nodes which represent another instance of the entire subroutine. Our methods cannot prove the consistency of a single interpretation of such a flowchart or a single $\rightsquigarrow$ relation. However, we can use our methods combined with mathematical induction to prove theorems of the type described above about the subroutine. Working out the details is a formidable task which we will not even contemplate here.

Probably the most significant generalization is to allow the creation and destruction of processes. Equating a process with a token (rather than a flowchart), this is easily done by defining new nodes which create and destroy tokens. These nodes, called *fork* and *join* nodes, were first defined in [6]. Their meaning is illustrated by Fig. 12, which shows a very precise

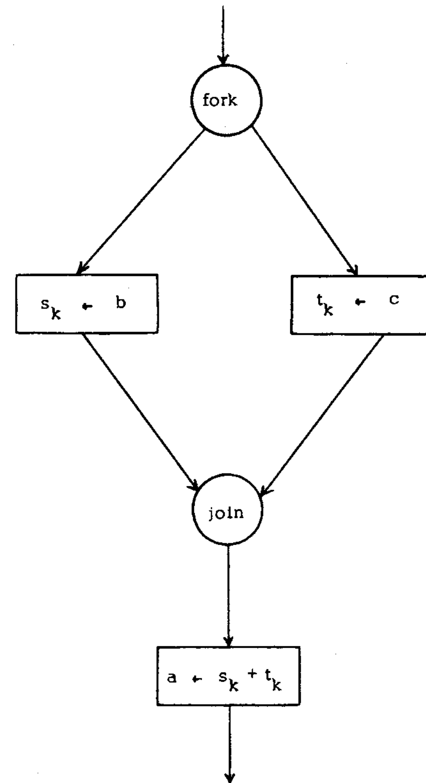


Fig. 12. Representation of $a := b + c$.

translation of the Algol statement $a := b + c$ where the fetches of b and c are not ordered. Executing the fork node places a token on each of its output arcs. Execution of the join node can only take place when there is a token on each input arc, and it replaces these tokens by a single token on the output arc.

Incorporating fork and join nodes into our formalism is disarmingly simple. The consistency conditions for these nodes are simple and obvious. We need only change Definition 3 c) so that for an interpretation I to be consistent, $I_j^\beta \wedge (\pi_j = \beta)$ must be monotone under I_k for all j, k including $j = k$.[6] Although this seems like a minor change, it turns out to be disastrous. It destroys the conceptual separation of the processes, making the design of a proof much more difficult.

Fortunately, there is no problem if the fork and join nodes are used in a properly disciplined fashion. We can restrict their use to the type of structure shown in Fig. 13, in which a process splits into concurrently executed subprocesses. The subprocesses can in turn split into subprocesses, and so on. This yields a hierarchical tree structure of processes and their subprocesses. In the definition of a consistent interpretation, an assertion attached to an arc of a process need not be monotone under the interpretation of that process or of any processes above or below it in the tree of processes. The definition of an inevitable set (Definition 7) must also be modified. The details are easy and are left to the reader.

[6]The assertion $\pi_j = \beta$ now means that at least one of Π_j's tokens is on arc β.

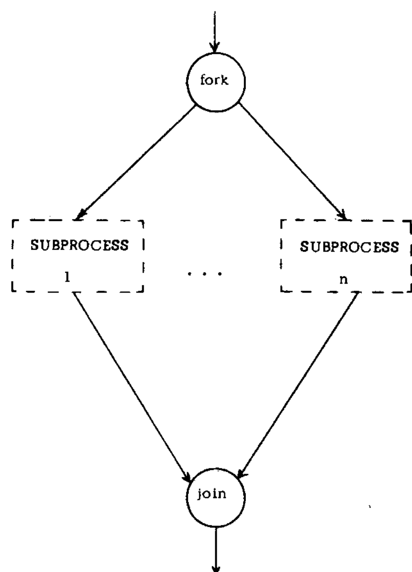


Fig. 13. Subprocess structure.

Comparison with Previous Methods

There have been several methods previously proposed for proving the correctness of multiprocess programs. They can be divided into two general classes: very general formal methods, and less general, usually less formal approaches. The earlier general methods such as [2] represented the multiprocess program as a single nondeterministic sequential program. This approach is unsatisfactory because it uses a very unnatural representation. The later formal methods of Ashcroft [1] and Keller [12] permitted explicit representation of multiple processes, where a process can be identified with a movable token on a directed graph.

Our method of proving safety properties can be viewed as a special case of the methods of Ashcroft and Keller. Although this may seem to diminish the significance of our method, it is precisely its more specific nature which makes the method useful. All methods of proving safety can be viewed as special cases of the following theorem: if $F: S \to S$ is multivalued function, $D \subset S$, and $F(D) \subset D$, then $F^n(D) \subset D$ for all $n > 0$ (where F^n denotes the composition of F with itself n times). We merely let F be the function which takes the current program state into its next state. However, its generality renders this theorem useless because it tells us very little about how to construct a proof for an actual program. By being more specific than Ashcroft and Keller, we have provided a method which gives more guidance to the programmer. We have not found the greater generality of these other methods to offer any significant advantage to compensate for their lack of guidance.

We differ considerably from Ashcroft and Keller in our method of proving liveness properties. Liveness is not considered at all by Ashcroft. Keller's concept of liveness is a weaker one than ours, and essentially states that some condition never becomes impossible. For example, it would consider a mutual exclusion condition to be suitably "live" so long as a waiting process always retained the possibility of entering its critical section—even though it might also be possible for it to wait forever. Although such a liveness property is often adequate in practice, there always lurks the possibility that some unforeseen "resonance" phenomenon in the implementation might make a process wait much longer than seemed likely.

Other proof procedures have been proposed which are more specific than ours, and assume some type of synchronization primitive [4], [10], [11], [15]. The basic reason for their assumptions is to insure that data cannot be accessed by one process while they might be modified by another process. It seems to have been generally accepted that this was necessary in order to allow a correctness proof [5, p. 241]. Such methods obviously cannot be used to design the lower level virtual machines—the ones which implement the synchronizing primitives. However, they can be useful for designing the higher level ones. Fortunately, these methods can all be carried out in terms of our formalism. They just require using the appropriate synchronizing primitives and process structuring techniques when designing the multiprocess flowchart programs. It seems useful to have a simple common formalism upon which to base these different methods.

After writing the initial version of this paper, we learned of the recent work of Owicki [16], [17]. Her method of proving safety properties is very similar to ours. Apart from syntactic details, the basic difference between her method and ours is that she uses assertions which may depend only on variable values, and not on token positions. The restriction requires the use of fictitious variables in most proofs. We find it more elegant to use assertions about token positions rather than introducing extraneous variables, but that is a matter of taste. Owicki does not prove liveness properties. Instead, she proves that programs written in terms of a special *await condition* primitive cannot become deadlocked by having all processes waiting at the same time.

Conclusions

We have presented a method for constructing formal, machine-verifiable proofs of correctness for multiprocess programs. It allows a reasonably simple formalization of informal proofs, and is practical for proving the correctness of the short algorithms commonly published in journals. With the type of automated aids which can now be built, the method should provide the basis for a system for designing and machine-verifying large real programs. This is because the program and its proof can be designed together in a top-down, hierarchical fashion. Even without the automated aids needed to carry out the proof at the levels of greatest detail, the method provides an informal proof of the correctness of the higher level design.

Since the method does not rely upon any synchronizing primitive, it can be used for any kind of multiprocess program—even the lowest layers of an operating system. The proof procedure does not require the introduction of any unnecessary synchronization into the program.

Although the method makes correctness proofs practical, it does not make them easy. Designing proofs is still a poorly understood art. Good proofs, like good programs, cannot be produced by bad programmers.

REFERENCES

[1] E. A. Ashcroft, "Proving assertions about parallel programs," *J. Comput. Syst. Sci.*, vol. 10, pp. 110-135, Jan. 1975.

[2] E. A. Ashcroft and Z. Manna, "Formalization of properties of parallel programs," *Machine Intelligence*, vol. 6, Edinburgh Univ. Press, 1970.

[3] G. Belpaire and J. P. Wilmotte, "Correctness of realization of levels of abstraction in operating systems," in *Operating Systems*, E. Gelenbe and C. Kaiser, Eds., *Lecture Notes in Computer Science 16*. New York: Springer Verlag, 1974.

[4] P. Brinch Hansen, "A comparison of two synchronizing concepts," *Acta Informatica*, vol. 1, pp. 190-199, 1972.

[5] P. Brinch Hansen, "Concurrent programming concepts," *Computing Surveys*, vol. 5, pp. 223-245, Dec. 1973.

[6] M. E. Conway, "A multiprocessor system design," in *1963 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 23. Washington, DC: Spartan Press, 1963, pp. 139-146.

[7] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent control with 'readers' and 'writers'," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 667-668, Oct. 1971.

[8] E. W. Dijkstra *et al.*, "On-the fly garbage collection: An exercise in cooperation," Burroughs Corporation, Rep. EWD595, submitted for publication.

[9] R. W. Floyd, "Assigning meanings to programs," in *Proc. Symp. Appl. Math.*, vol. 19, Amer. Math. Soc., pp. 19-32, 1967.

[10] A. N. Habermann, "Synchronization of communicating processes," *Commun. Ass. Comput. Mach.*, vol. 15, pp. 177-184, Mar. 1970.

[11] C. A. R. Hoare, "Parallel programming–An axiomatic approach," Stanford A. I. Lab., Memo AIM-219, Oct. 1973.

[12] R. M. Keller, "Formal verification of parallel programs," *Commun. Ass. Comput. Mach.*, vol. 19, pp. 371-384, July 1976.

[13] L. Lamport, "A new solution of Dijkstra's concurrent programming problem," *Commun. Ass. Comput. Mach.*, vol. 17, pp. 453-455, Aug. 1974.

[14] —, "On concurrent reading and writing," Massachusetts Computer Associates, Inc., CA-7409-0511, Sept. 1974, to be published in *Commun. Ass. Comput. Mach.*

[15] K. N. Levitt, "The application of program-proving techniques to the verification of synchronization processes," in *1972 Fall Joint Comput. Conf., AFIPS Conf. Proc.*, vol. 41. Montvale, NJ: AFIPS Press, 1972, pp. 33-47.

[16] S. Owicki, "Axiomatic proof techniques for parallel programs," Ph.D. dissertation, Cornell University, Ithaca, NY, Aug. 1975.

[17] S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs," to be published in *Acta Informatica*.

Leslie Lamport received the B.S. degree from the Massachusetts Institute of Technology, Cambridge, MA, and the M.S. and Ph.D. degrees from Brandeis University, Waltham, MA, all in pure mathematics.

He has been employed by the Mitre Corporation and Marlboro College, and is currently a Senior Analyst at Massachusetts Computer Associates, Inc., Wakefield, MA. His primary research interest is in the theoretical aspects of concurrent multiprocessing.