Proving Liveness Properties of Concurrent Programs

SUSAN OWICKI
Stanford University
and
LESLIE LAMPORT
SRI International

A liveness property asserts that program execution eventually reaches some desirable state. While termination has been studied extensively, many other liveness properties are important for concurrent programs. A formal proof method, based on temporal logic, for deriving liveness properties is presented. It allows a rigorous formulation of simple informal arguments. How to reason with temporal logic and how to use safety (invariance) properties in proving liveness is shown. The method is illustrated using, first, a simple programming language without synchronization primitives, then one with semaphores. However, it is applicable to any programming language.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Program Verification; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Temporal logic, liveness, fairness, proof of correctness, multiprocessing, synchronization

1. INTRODUCTION

This paper presents a method for proving properties of concurrent programs. By using the word "proving", we are committing ourselves to logical rigor—a method based upon an unsound logical foundation cannot be said to prove anything. However, our purpose is to develop a practical method for verifying that programs do what they are supposed to do, not to develop logical formalism. While we hope that logicians will find this work interesting, our goal is to define a method that programmers will find useful.

The first author's work was supported in part by the Defense Advanced Research Project Agency under contract MDA903-79-C-0680; the second author's work was supported in part by the National Science Foundation under grant MCS 78-16783.
Authors' addresses: S. Owicki, Computer Systems Laboratory, Stanford Electronics Laboratories, Department of Electrical Engineering, Stanford University, Stanford, CA 94305; L. Lamport, Computer Science Laboratory, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025.

There is a rather large body of sad experience to indicate that a concurrent program can withstand very careful scrutiny without revealing its errors. The only way we can be sure that a concurrent program does what we think it does is to prove rigorously that it does it. We have found that there are two kinds of properties one usually wants a concurrent program to satisfy:

—*Safety properties*, which state that something bad never happens—that is, that the program never enters an unacceptable state.
—*Liveness properties*, which state that something good eventually does happen—that is, that the program eventually enters a desirable state.

Some familiar safety properties are

Partial correctness: if the program begins with the precondition true, then it can never terminate with the postcondition false.
Absence of deadlock: the program never enters a state in which no further progress is possible.
Mutual exclusion: two different processes are never in their critical sections at the same time.

The only liveness property that has received careful formal treatment is program termination. However, concurrent programs are capable of many more sins of omission than just failure to terminate. Indeed, for many concurrent programs—operating systems are a prime example—termination is known by the less flattering name of "crashing", and we want to prove that it does not happen. For such programs other kinds of liveness properties are important, for example:

—Each request for service will eventually be answered.
—A message will eventually reach its destination.
—A process will eventually enter its critical section.

A number of methods have been proposed for proving safety properties of concurrent programs, but formal proof of liveness has received little attention. For sequential programs, termination is the one liveness property that has been studied extensively. It is typically proved by using some sort of inductive argument to show that every loop terminates, allowing us to conclude that the entire program terminates. In addition, the intermittent assertion method [1, 13] provides an informal approach to deducing more general liveness properties of sequential programs.

There has been some work on proving particular liveness properties of concurrent programs, such as absence of livelock [8] or the existence of cyclically recurring states [4]. Perhaps the first formal method for proving general liveness properties was given by Lamport in [11], where the idea of a proof lattice first appeared. However, his proofs of even simple liveness properties were unbearably long and hard to follow. The fundamental innovation of our method is the combining of Lamport's proof lattices with Pnueli's temporal logic [16]. This permits rigorous proofs that are easy to understand because they capture our intuitive understanding of how the program works. Flon and Suzuki [3] also present a formal proof system for liveness properties, but it is not clear how to use it in constructing program proofs.

Liveness properties involve the temporal concept "eventually". The most obvious way to formalize this concept is to use predicate calculus formulas containing a "time" variable. This approach was used by Francez and Pnueli [4] in their proof method for cyclic properties. However, the explicit introduction of time in this way leads to complicated formulas that tend to obscure the underlying ideas. To avoid this, temporal logic was used by Burstall [1] for reasoning about sequential programs and by Pnueli [15, 16] for concurrent programs.

Temporal logic is an extension of ordinary logic to include certain kinds of assertions about the future. The form that we use—the "linear time logic" described in [10]—has two temporal operators:

$\Box$ — meaning "now and forever";
$\Diamond$ — meaning "now or sometime in the future".

The following are some examples of temporal assertions using these operators:

$x > 0$—"the variable x is positive now";
$\Box(x > 0)$—"x is positive now and forever";
$\Diamond(x > 0)$—"x is positive now or will be positive some time in the future".

Temporal logic provides the logical foundation for our proof method.

One of the most important concepts in concurrent processing is *fairness.* Fairness means that every process gets a chance to make progress, regardless of what other processes do. Fairness is guaranteed by a truly concurrent system in which each process is run on its own processor: one process cannot halt the physical execution of a process running on a different processor. Multiprogramming systems, in which a single processor is shared by several processes, may or may not provide fairness, depending on the scheduling algorithm used for processor allocation.

In proving safety properties, it does not matter whether one assumes fairness—any safety property that holds under fair scheduling will also hold under unfair scheduling. However, many programs satisfy their liveness requirements only if the underlying implementation guarantees fairness. A methodology that cannot prove these properties will be inadequate for dealing with true concurrency, so fairness must be part of a general method for dealing with concurrent programs. Fairness has proved to be a stumbling block for formal systems.[1] Our temporal logic approach makes it easy to express fairness properties and use them in our proofs.

The paper is organized as follows. In the next section we present a simple programming language that serves to illustrate the method. We then introduce temporal logic and describe the proof lattices. Section 4 summarizes the proof of safety properties and their expression in temporal logic. Section 5 presents the basic axioms that define the liveness properties of our programming language, as well as derived inference rules and examples of proofs. Section 6 develops a more complex example: liveness proofs for a mutual exclusion algorithm. Section 7

[1] For example, attempts to apply denotational semantics have had to cope with the fact that fairness introduces discontinuity: the limit of a sequence of fair executions σ_n in which a certain process gets to run $(1/n)$th of the time is an unfair execution in which the process never runs.

illustrates how languages with synchronization primitives can be handled, presenting axioms for semaphore operations and an example of their use. Section 8 concludes with a discussion of what we have and have not done.

2. THE PROGRAMMING LANGUAGE

The programming language used in this paper is a very simple one, containing only assignment, **while**, and **cobegin** statements; concatenation (sequencing); and variable declarations at the beginning of the program. These language features are illustrated by the program of Figure 1, which is otherwise of no particular interest. Except for the **cobegin**, the language constructs are familiar and require no explanation. A **cobegin** statement

cobegin $S_1 \parallel \cdots \parallel S_n$ **coend**

causes the statements $S_1, \ldots S_n$ to be executed concurrently. The S_i's are often called *processes.* Since variable declarations can only appear at the beginning of a program, all variables are global to the entire program. In Section 7, we consider additional synchronizing statements.

Besides the **cobegin**s, the other novel feature of the program of Figure 1 is the angle brackets. In order to specify a concurrent program, one must state which actions are atomic. Atomic actions are indivisible and represent the finest grain of process interleaving. We indicate the atomic actions by enclosing them in angle brackets. In this paper, we require that each assignment and each test in a **while** statement be an atomic action. The angle brackets are therefore redundant, but we use them anyway to remind us of the grain of atomicity that we are assuming. Nonatomic assignments and tests are considered in [9] and cause no fundamental difficulty.

The purpose of this paper is to describe a method of proving things about programs. It is rather important that such a proof method allow one to prove only things that are true. To make sure that this is the case, one must be able to determine precisely what is true about the programs written in our language. To this end, we describe the semantics of the programming language somewhat informally, but carefully enough so that there should be no ambiguity about the meaning of programs.

We define the semantics of a program to be the set of all possible executions of that program. More formally, the semantics of a program is given by a set Σ of *execution sequences.* Each element of Σ is a sequence of *program states.* A program state s consists of two parts:

—An assignment of a value to each program variable.
—A control component $ready(s)$ consisting of a set of atomic actions. (When the program is being executed, the next action to be performed is chosen from $ready(s)$.)

One example of a state for the program of Figure 1 is

$$\langle x = 1, y = 3; ready = \{e, g\} \rangle.$$

Here the variables x and y are assigned the values 1 and 3, respectively, and the control component indicates that there are two possibilities for the next action to

```
integer x, y ;
a: < x := 0 > ;
b: cobegin
        c: < y := 0 >;
        d: cobegin
                e: < y := 2 * y >     ▮     f: < y := y + 3>
           coend
   ▮
        g: while < y = 0 > do    h: < x := x + 1> od
   coend ;
j: < x := 2 * y >
```

Fig. 1. A concurrent program.

be executed:

—the assignment statement $\langle y := 2 * y \rangle$;
—the $\langle y = 0 \rangle$ test in the **while** loop.

To save space, we omit the variable names when writing states, so the above state is written $\langle 1, 3; \{e, g\} \rangle$. Note how we use program labels to describe the *ready* component—in particular, how the label attached to a **while** statement denotes the test operation.

An execution sequence $s_0, s_1, \ldots$ in Σ represents a program execution that starts in state s_0, performs one atomic action to reach state s_1, performs another atomic action to reach state s_2, and so on. It simplifies our notation if all the elements of Σ are infinite sequences. Therefore, if the program execution terminates, we repeat the last state indefinitely to get an infinite sequence. This is purely a notational convenience and has no deep significance.

As an example, consider the following execution sequence for the program of Figure 1. It starts with initial values of 2 for x and 7 for y, with control at the beginning of the program.

$$s_0 = \langle 2, 7; \{a\} \rangle;$$
$$s_1 = \langle 0, 7; \{c, g\} \rangle;$$
$$s_2 = \langle 0, 0; \{e, f, g\} \rangle;$$
$$s_3 = \langle 0, 0; \{e, f, h\} \rangle;$$
$$s_4 = \langle 1, 0; \{e, f, g\} \rangle;$$
$$s_5 = \langle 1, 3; \{e, g\} \rangle;$$
$$s_6 = \langle 1, 6; \{g\} \rangle;$$
$$s_7 = \langle 1, 6; \{j\} \rangle;$$
$$s_8 = s_9 = s_{10} = \cdots = \langle 12, 6; \{\} \rangle.$$

The reader should observe the following things about the control component:

—There is no explicit "control point" at the beginning of a **cobegin**. Thus, after execution of statement a, the control component becomes $\{c, g\}$.

—The fact that one process in a **cobegin** has terminated is indicated by the absence of any action from that process in the *ready* component. Thus, the component $\{e, g\}$ in s_5 indicates that the second process of the inner **cobegin** has terminated.
—After execution of the body of a **while** statement, control returns to the beginning of the statement. There is no control point between the end of the body and the test, and the *ready* component does not distinguish the initial entry point from the "looping" point.
—Execution terminates when the *ready* component is empty.

In general, a *possible program state* is one in which

—All variables are assigned legal values—for example, an integer variable is not assigned a boolean value.
—The *ready* component consists of a set of concurrent atomic actions, where two actions are *concurrent* if they occur in different processes of some **cobegin** statement.

The second condition means that we exclude *ready* components like $\{a, c\}$ for the program of Figure 1, because the program control structure does not permit concurrent execution of statements a and c.

The set Σ of all possible execution sequences consists of all sequences $s_0, s_1, \ldots$ satisfying the following requirements:

—*Valid starting state*: s_0 is a possible program state.
—*Transition*: for each $i > 0$, s_i is obtained from s_{i-1} by executing one atomic action in $ready(s_{i-1})$. The only atomic actions in our language are the assignment statement and **while** test, which affect the program state in the obvious way.
—*Fairness*: if a is an atomic action in $ready(s_i)$, then, for some $j > i$, s_j is obtained by executing a.

Although in our example we happened to pick a starting state in which control was at the beginning of the program, this is not necessary. We allow an execution sequence to start in any possible program state. Thus, the sequence $s_4, s_5, \ldots$ is also a possible execution sequence for the above program. In fact, for any program, the set Σ has the following *tail closure* property:

if $\sigma = s_0, s_1, \ldots$ is in Σ, then for all $i > 0$, $\sigma^{(i)}$ is in Σ, where $\sigma^{(i)} = s_i, s_{i+1}, \ldots$

Tail closure implies that the set of possible computations from a given state is completely determined by the state itself and not by the history of the computation in reaching that state. Defining Σ in this way, rather than restricting ourselves to executions that begin in a distinguished starting state, will prove convenient when using temporal logic.

Although we speak of "concurrent" programming, we are actually modeling concurrency by a nondeterministic interleaving of atomic actions from the various processes. With an appropriate choice of atomic actions, almost any concurrent system can be accurately modeled this way, in the sense that any safety or liveness properties proved about the model will be true of the system. For example, a network of processes that communicate by exchanging messages can be modeled by using a process to represent the communication medium. This

process communicates with the transmitting and receiving processes via shared variables, and its local variables represent the state of the medium. With this structure it is easy to model a variety of assumptions about message transmission—for example, that the process delivers all messages safely or that it may nondeterministically lose or modify some of them.

The nondeterministic interleaving in our model of concurrency means that we make no assumption about the relative speeds of the processes. However, fairness implies that no processor is infinitely faster than another. This requirement is met, for example, by an implementation that provides a separate processor to execute each active process and fair scheduling of concurrent accesses to shared variables.

The set of execution sequences in our model includes all those that could occur when the program is executed fairly. In any implementation, the relative speeds of the processors and the scheduling mechanism would further constrain the possible execution sequences. However, as long as all execution sequences are a subset of those in Σ, any results proved by the methods in this paper will be true for that implementation.

3. TEMPORAL LOGIC

Temporal logic provides us with both a language for stating program requirements and a set of rules for reasoning about them. We now give a precise formulation of temporal logic in terms of program execution sequences. The version of temporal logic we use was introduced by Pnueli in [16] and is the "linear time" logic discussed in [10]. Our exposition here is brief, and we refer the reader to the above papers for more details.

3.1 Immediate Assertions

Temporal logic assertions are built up from *immediate assertions*, using the ordinary logical operators $\wedge$, $\vee$, and $\sim$ and the temporal operators $\Box$ and $\Diamond$. An immediate assertion is a boolean-valued function of the program state. It may refer to program variables or to the control component. We write $s \models P$ to denote that the immediate assertion P has the value *true* for state s. In this case we say that P *holds for* s, or that s *satisfies* P. For example, $s \models x = 1$ means that the program state s assigns the value 1 to x.

We use three kinds of immediate assertions to refer to the control component: *at* A, *in* A, and *after* A, where A is an executable program statement. The immediate assertion *at* A holds for all states where control is at the beginning of A. Since the executable statements in our programming language are formed using assignment, **while**, and **cobegin** statements and concatenation, we can define the immediate assertion *at* A as follows:

$s \models at\ A$ if and only if

IF A is a: $\langle x := e \rangle$ THEN $a \in ready(s)$;

IF A is a: **while** $\langle B \rangle$ **do** C **od** THEN $a \in ready(s)$;

IF A is **cobegin** $B_1 \blacksquare \cdots \blacksquare B_n$ **coend**

THEN $(s \models at\ B_1)$ and ... and $(s \models at\ B_n)$;

IF A is $B;C$ THEN $s \models at\ B$.

Thus, for the program of Figure 1 we see that $s \models at\ b$ if and only if $ready(s) = \{c, g\}$, and $s \models at\ d$ if and only if e and f are both in $ready(s)$.

Note that the immediate assertion *at A* refers to a specific instance of a statement. For example, if there were two "$\langle x := x + 1\rangle$" statements in the program, then we could not write "*at* $\langle x := x + 1\rangle$" because there would be no way of knowing which "$\langle x := x + 1\rangle$" statement it referred to. We use statement labels to refer unambiguously to individual statements.

The immediate assertion *in A* holds for states where control is at the beginning of A or somewhere inside A. In other words, $s \models in\ A$ if and only if either $s \models at\ A$ or there is some component B of A such that $s \models at\ B$. For example, in the program of Figure 1, the following relations hold:

$$in\ a \equiv at\ a;$$

$$in\ g \equiv at\ g \vee at\ h;$$

$$in\ d \equiv at\ e \vee at\ f;$$

$$in\ b \equiv at\ c \vee in\ d \vee in\ g.$$

The immediate assertion *after A* holds for states where control is immediately after statement A. The following definition uses an "outward recursion" to define *after A* in terms of the program statement B that immediately contains A.

$s \models after\ A$ if and only if

IF A is the entire program THEN $(ready(s) = \varphi)$;

IF B is **while** $\langle C\rangle$ **do** A **od** THEN $s \models at\ B$;

IF B is **cobegin** ... ▮ A ▮ ... **coend**

THEN $(s \models after\ B)$ or $[(s \models in\ B)$ and not $(s \models in\ A)]$;

IF B is $A;C$ THEN $s \models at\ C$;

IF B is $C;A$ THEN $s \models after\ B$.

Note that being *after* the body of a **while** loop is the same as being *at* the loop test. Also, being *after* a process A in a **cobegin** statement means either that the entire **cobegin** has finished or that some of its processes (but not A) are still being executed.

3.2 Temporal Assertions

Where an immediate assertion is a function on program states, a temporal assertion is a boolean-valued function on execution sequences. We write $\sigma \models P$ to denote that temporal assertion P is true for the execution sequence σ. For the remainder of this discussion, we let σ denote an arbitrary execution sequence $s_0, s_1, \ldots$. We think of time as being composed of an infinite sequence of discrete instants, where s_i represents the state of the computation at time i. We refer to time 0 as the present and any time greater than 0 as in the future.

An immediate assertion is interpreted as a temporal assertion that refers to the present. More precisely, this means that an immediate assertion P (a statement about program states) is interpreted as a temporal assertion (a statement about

execution sequences) by the convention

$$\sigma \models P \qquad \text{if and only if} \qquad s_0 \models P.$$

Temporal assertions that refer to the future as well as the present are obtained with the temporal operators $\Box$ and $\Diamond$. The assertions formed with these operators always refer to both the present and the future. The unary operator $\Box$ means "for all present and future times it will be true that", and $\Diamond$ means "at some present or future time it will be true that". Recalling that $\sigma^{(i)}$ is the execution sequence $s_i, s_{i+1}, \ldots$, we can define these temporal operators formally as follows, where P denotes any temporal assertion:

$$\sigma \models \Box P \qquad \text{if and only if} \qquad \forall i \geq 0\colon \sigma^{(i)} \models P;$$

$$\sigma \models \Diamond P \qquad \text{if and only if} \qquad \exists i \geq 0\colon \sigma^{(i)} \models P.$$

Note that $\Diamond$ is the dual of $\Box$—that is, $\Diamond P \equiv \sim\Box\sim P$.

Since temporal assertions are formed from immediate assertions using the temporal operators $\Box$ and $\Diamond$ and the ordinary logical operations $\wedge$, $\vee$, and $\sim$, the definition of $\sigma \models P$ for any temporal assertion P is completed as follows:

$$\sigma \models (P \wedge Q) \qquad \text{if and only if} \qquad (\sigma \models P) \text{ and } (\sigma \models Q);$$

$$\sigma \models (P \vee Q) \qquad \text{if and only if} \qquad (\sigma \models P) \text{ or } (\sigma \models Q);$$

$$\sigma \models (\sim P) \qquad \text{if and only if} \qquad \text{it is not the case that } \sigma \models P.$$

In discussing temporal formulas, we often use English phrases like "P holds at time i" instead of the formula $\sigma^{(i)} \models P$. Unfortunately, there is no English tense that combines the present and the future in the way that the temporal operators do. To smooth our syntax, we take a liberty with the English language by using the future tense in such cases, as in the statement "P will be true now or in the future".

We now consider some examples of temporal logic formulas. As usual, we define $\supset$ (logical implication) in terms of $\vee$ and $\sim$. If P and Q are immediate assertions, then the temporal assertion $P \supset \Box Q$ means "if P is true now, then Q will always be true". More precisely, $P \supset \Box Q$ is true for an execution sequence if P is false in the first state or Q is true in all states. This type of assertion expresses a basic safety property and is discussed further in Section 4.

As a second example, consider the formula $\Box(I \supset \Box I)$. It means that if I ever becomes true, then it will remain true forever. An immediate assertion I for which this is true is said to be *invariant*. Invariants play a major role in the proof of safety properties.

As a final example, consider the assertion $\Box(P \supset \Diamond Q)$. It states that if P ever becomes true, then Q will be true at the same time or later. Such an assertion expresses a liveness property and is discussed in Section 5. This particular formula is very useful, and we abbreviate it as $P \rightsquigarrow Q$ (pronounced "P leads to Q"):

$$(P \rightsquigarrow Q) \equiv \Box(P \supset \Diamond Q).$$

(Manna and Waldinger [13] use a similar notation with the following meaning: if P is true at some time, then Q is true at some time, not necessarily later.)

3.3 Theorems

Our definition of $\sigma \models P$ says what it means for the assertion P to be true for the single execution sequence σ. However, we are not interested in properties that hold for some individual execution sequence, but in properties that hold for all of a program's execution sequences. We say that an assertion P holds for a program if it holds for all execution sequences of that program—that is, for all elements of Σ. For example, $P \rightsquigarrow Q$ is true for a program if and only if any execution of the program that reaches a state where P is true must subsequently reach a state where Q is true. To prove that an assertion holds for a program, we use two kinds of reasoning:

—Reasoning based upon the semantics of the individual program under consideration.
—Reasoning that is valid for all programs.

The second kind of reasoning is what temporal logic is all about, and it is the subject of this section. We return to the semantics of programs in subsequent sections.

Temporal logic incorporates all the laws of reasoning of ordinary logic—that is, the axioms and rules of inference of the propositional calculus. For example, if we can prove that the temporal assertions P and $P \supset Q$ are true for a program, then we can conclude that the assertion Q is true for that program.

We also assume some method of reasoning about immediate assertions. For example, the rules of integer arithmetic allow us to prove that $(x > 1) \supset (x > 0)$ for any integer x. It is often possible to prove theorems about program variables that depend only on the types of values they may take on. Such theorems must be true of any program state in which the variables have the appropriate type. Thus $(x > 1) \supset (x > 0)$ must be true for all states of a program in which x is an integer variable. Our first law allows us to use these theorems in our temporal logic reasoning.

TL1. *If the immediate assertion P is true for every program state, then P is true for the program.*

PROOF. This follows immediately from the fact that an immediate assertion is true for an execution sequence if and only if it is true for the first state of that sequence. □

Logicians will note that the proofs of this and the remaining laws are actually proofs of their validity, based upon the semantic definitions given above. Since our goal is to familiarize the reader with temporal logic as an intuitively meaningful way of reasoning, not to overwhelm him with rigor, our proofs will be quite informal. We hope the reader will come to feel, as we do, that this kind of temporal logic reasoning is simple and natural. All our laws can also be proved using the formal temporal logic system given in [16].

The following law states that a true assertion must always be true—truth is eternal.

TL2. *If the temporal assertion P is true for a program S, then $\Box P$ is true for S.*

PROOF. Let σ be any execution sequence of S. The tail closure property implies that for any i, $\sigma^{(i)}$ is also an execution sequence of S. By the hypothesis, this means that $\sigma^{(i)} \models P$ for all i. It then follows immediately from the definition of $\sigma \models \Box P$ that $\Box P$ is true for any execution sequence of S. □

Note that TL2 does *not* imply that the assertion $P \supset \Box P$ is true for all programs. This assertion states that if P is true at the beginning of an execution sequence, then it is true at all points in the execution sequence. TL2 states that if an assertion is true at the beginning of *all* execution sequences, then that assertion must be true throughout all execution sequences.

Connoisseurs of logic will observe that TL1 and TL2 are inference rules. The rest of our laws are theorems: temporal assertions that are true for every program. We have not tried to give a complete set of theorems for proving properties of programs, merely ones we use in our examples. With a little experience, the reader will be able to decide easily for himself whether a temporal logic formula he would like to use as a theorem is really true.

TL3. $\Box(P \supset Q) \supset (P \rightsquigarrow Q)$.

PROOF. Consider an execution sequence σ which satisfies the left-hand side of TL3. For this execution sequence, whenever P is true, Q is true too. Thus, whenever P is true, Q will be true "now or in the future", which means that $P \rightsquigarrow Q$. □

TL4. (*a*) $\Box(P \wedge Q) \equiv (\Box P \wedge \Box Q)$.
(*b*) $\Diamond(P \vee Q) \equiv (\Diamond P \vee \Diamond Q)$.

PROOF. To establish TL4(a), we must show that, for any execution sequence σ,

$$\sigma \models \Box(P \wedge Q) \quad \text{if and only if} \quad \sigma \models (\Box P \wedge \Box Q).$$

This is easily verified by expanding the definition of $\Box$:

$$\begin{aligned}
\sigma \models \Box(P \wedge Q) &\equiv \forall i \geq 0\colon \sigma^{(i)} \models (P \wedge Q) \\
&\equiv \forall i \geq 0\colon (\sigma^{(i)} \models P) \wedge (\sigma^{(i)} \models Q) \\
&\equiv (\forall i \geq 0\colon \sigma^{(i)} \models P) \wedge (\forall i \geq 0\colon \sigma^{(i)} \models Q) \\
&\equiv (\sigma \models \Box P) \wedge (\sigma \models \Box Q) \\
&\equiv \sigma \models (\Box P \wedge \Box Q).
\end{aligned}$$

TL4(b) follows from TL4(a) using the duality of $\Box$ and $\Diamond$. □

Note that the formula

$$\Box(P \vee Q) \equiv (\Box P \vee \Box Q)$$

is not a theorem. The left-hand side is true for an execution sequence if either P or Q is true at all times, while the right-hand side is true if either P is true at all times or Q is true at all times. The implication

$$(\Box P \vee \Box Q) \supset \Box(P \vee Q)$$

is a theorem of temporal logic, but we will not use it.

TL5. $(\Box P \wedge \Box(P \supset Q)) \supset \Box Q$.

PROOF. If P and $P \supset Q$ are true at all times, then Q must be true at all times. □

TL6. $\Diamond P \vee \Box \sim P$.

PROOF. This is an obvious consequence of the duality of $\Box$ and $\Diamond$—that is, $\Diamond P \equiv \sim\Box\sim P$. □

Theorem TL6 states that in every execution sequence, either P is always false or there is some time at which it is true. This fact is frequently used in proofs of liveness properties: to prove $\Diamond P$, one first shows that $\Box\sim P$ leads to a contradiction and then applies TL6.

The next theorem states that $\rightsquigarrow$ is a transitive relation.

TL7. $((P \rightsquigarrow Q) \wedge (Q \rightsquigarrow R)) \supset (P \rightsquigarrow R)$.

PROOF. Recall that $U \rightsquigarrow V$ holds for an execution sequence σ if and only if whenever U holds at time i, there is some time $j \geq i$ at which V holds. Hence, $P \rightsquigarrow Q$ implies that if P is true at time i, Q is true at some time $j \geq i$. Likewise, $Q \rightsquigarrow R$ implies that if Q is true at time j, R is true at some time $k \geq j$. Together, they imply that if P is true at time i, R is true at some time $k \geq i$; so $P \rightsquigarrow R$ is true. □

TL8. $((P \rightsquigarrow R) \wedge (Q \rightsquigarrow R)) \supset ((P \vee Q) \rightsquigarrow R)$.

PROOF. This follows immediately from the fact that if $(P \vee Q)$ holds at some time, either P or Q holds at that time. □

TL9. $\Box(P \vee Q) \supset (\Box P \vee \Diamond Q)$.

PROOF. Consider an execution sequence in which $P \vee Q$ is always true. If there is any point at which P is false, Q must be true at that point. Therefore, either P is always true or there is a point at which Q is true. □

TL10. $[(P \wedge \Box Q) \rightsquigarrow R] \supset [(P \wedge \Box Q) \rightsquigarrow (R \wedge \Box Q)]$.

PROOF. Consider an execution sequence that satisfies the left-hand side of the implication. If P is true at time i and Q is true from time i on, then R must be true at some time $j \geq i$. Since Q will still be true at time j and from then on, this implies that the execution sequence satisfies the right-hand side of the implication as well. □

3.4 Proof Lattices

Suppose that the following three assertions hold for a program:

(1) $P \rightsquigarrow (R1 \vee R2)$;
(2) $R1 \rightsquigarrow Q$;
(3) $R2 \rightsquigarrow Q$.

Writing the meaning of each of these assertions as follows:

(1) if P is true at any time i, then $R1$ or $R2$ will be true at some time $j \geq i$;

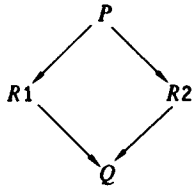

Fig. 2. Lattice proof outline for $P \rightsquigarrow R$.

(2) if $R1$ is true at any time j, then Q will be true at some time $k \geq j$;
(3) if $R2$ is true at any time j, then Q will be true at some time $k \geq j$,

we easily see that they imply the truth of $P \rightsquigarrow Q$. Formally, this is proved by a simple application of TL8 and TL7.

This reasoning is conveniently described by the *proof lattice* of Figure 2. The two arrows leading from P to $R1$ and $R2$ denote the assertion $P \rightsquigarrow (R1 \vee R2)$; the arrow from $R1$ to Q denotes the assertion $R1 \rightsquigarrow Q$; and the arrow from $R2$ to Q denotes the assertion $R2 \rightsquigarrow Q$.

In general, we make the following definition:

Definition. A *proof lattice* for a program is a finite directed acyclic graph in which each node is labeled with an assertion, such that

(1) There is a single *entry node* having no incoming edges.
(2) There is a single *exit node* having no outgoing edges.
(3) If a node labeled R has outgoing edges to nodes labeled $R_1, R_2, \ldots, R_k$, then $R \rightsquigarrow (R_1 \vee R_2 \vee \cdots \vee R_k)$ holds for the program.

The third condition means that, if R is true at some time, then one of the R_i must be true at some later time. By a generalization of the informal reasoning used for the lattice of Figure 2, it is easy to see that if the entry node assertion is true at some time, then the exit node assertion must be true at some later time. This is stated and proved formally by the following theorem:

THEOREM. *If there is a proof lattice for a program with entry node labeled P and exit node labeled Q, then $P \rightsquigarrow Q$ is true for that program.*

PROOF. We prove the following hypothesis, which clearly implies that $P \rightsquigarrow Q$.

Induction Hypothesis. If r is a node in the lattice with label R, then $R \rightsquigarrow Q$.

The proof is by induction on the length of the longest path from node r to the exit node. (Since the lattice has only one exit, either r is the exit or there is a path from r to the exit.) If the longest path has length 0, the hypothesis clearly holds, since then r is the exit node and $R = Q$.

Now assume that the hypothesis holds for nodes whose longest path to the exit has length $n \geq 0$, and consider a node r whose longest path to the exit has length $n + 1$. Let the nodes reached by outgoing edges from r be labeled $R_1, R_2, \ldots, R_k$. By definition of a proof lattice, we have

$$R \rightsquigarrow (R_1 \vee R_2 \vee \cdots \vee R_k).$$

By the induction hypothesis, $R_i \rightsquigarrow Q$ for $i = 1, \ldots, k$. Applying TL8 $k - 1$ times yields $R_1 \vee \cdots \vee R_k \rightsquigarrow Q$. It now follows from TL7 that $R \rightsquigarrow Q$. □

Proof lattices in which every node is labeled by an immediate assertion were introduced in [11]. The significant change introduced in this paper is the use of more general temporal assertions in the lattices—in particular, assertions involving the $\Box$ operator.

Consider a lattice containing a node labeled R with arcs pointing to nodes labeled $R_1, \ldots, R_k$. This construction implies that if R ever becomes true during execution of the program, one of the R_i must subsequently become true. Now suppose that R has the form $P \wedge \Box Q$. Saying that R is true at some time means that P and Q are true then, and that Q will be true at all future times. In particular, Q must be true when any of the R_i subsequently become true. Hence, we could replace each of the R_i by $R_i \wedge \Box Q$. More formally, it follows from TL10 that condition (3) in the definition of a proof lattice still holds if each R_i is replaced by $R_i \wedge \Box Q$.

We see from this that if $\Box Q$ appears as a conjunct ("and" term) of an assertion in a proof lattice, then $\Box Q$ is likely to appear as a conjunct of the assertions "lower down" in the lattice. It is therefore convenient to introduce the following notation, which makes our proof lattices clearer. For any assertion Q, drawing a box labeled $\Box Q$ around some of the nodes in the lattice denotes that $\Box Q$ is to be conjoined to the assertion attached to every node in the box. This notation is illustrated by the proof lattice of Figure 3, which is expanded in Figure 4 into the same proof lattice written without the box notation.

The lattice in Figure 3 also illustrates the typical structure of a proof by contradiction for $P \rightsquigarrow Q$. In the first step, the proof is split into two cases based on the temporal logic theorem

$$P \rightsquigarrow [Q \vee (P \wedge \Box \sim Q)].$$

This theorem can be proved using TL3, TL6, and TL10. Intuitively, it is true because starting from a time when P is true, either

- Q will be true at some subsequent time, or
- $\sim Q$ will be true from then on.

The former possibility is represented by the right-hand branch, the latter by the left-hand branch. Within the box labeled $\Box \sim Q$ is some argument that leads to a contradiction, which appears at the node labeled *false*. Note that *false* $\rightsquigarrow Q$ follows from TL1–TL3, since *false* $\supset Q$ is a tautology. Thus, the general pattern of these proofs by contradiction is to assume that the desired predicate never becomes true, and then show that this assumption leads to a contradiction. We will see a number of examples of this type of reasoning in the next two sections.

4. SAFETY

In order to prove that "something good eventually happens", one usually has to show that "nothing bad happens" along the way. In other words, in order to prove a liveness property, one must usually prove one or more safety properties. A number of formal methods have been proposed for proving safety properties [7, 9, 11, 14]. They all permit one to prove the same kind of properties, the differences

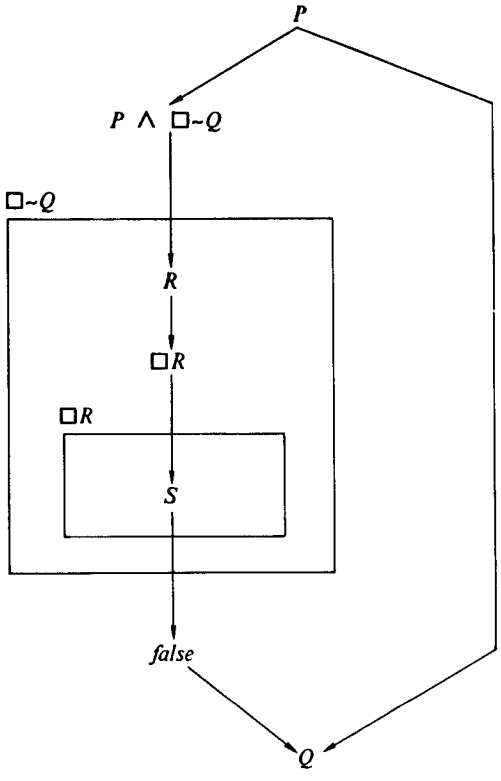

Fig. 3. Abbreviated lattice.

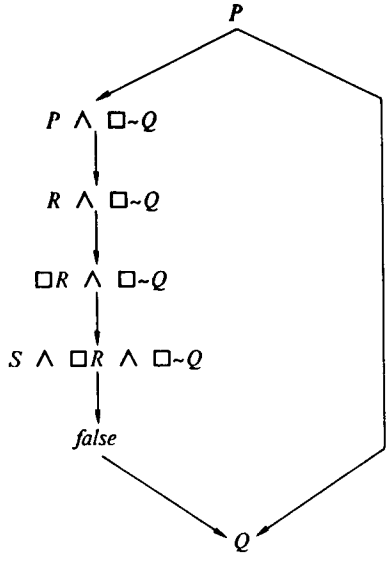

Fig. 4. Expanded lattice for Figure 3.

among the methods being largely syntactic. We now describe their basic approach in terms of temporal logic.

A safety property has the form $P \supset \Box Q$, where P and Q are immediate assertions. This assertion means that if the program starts with P true, then Q is always true throughout its execution. (A more general type of safety property is discussed in the appendix.) The most familiar safety property is *partial correctness*, which states that if the program S begins execution with some precondition P true, and if the execution of S terminates, then it terminates with some postcondition Q true. This is expressed by the temporal logic formula

$$(at\ S \wedge P) \supset \Box(after\ S \supset Q).$$

For most sequential programs, partial correctness is the only safety property required. For concurrent programs, a number of interesting safety properties have been considered. As an example, we consider everyone's favorite multiprocess synchronization property: mutual exclusion. Let S be a program with two processes, each of which has a *critical section*. The mutual exclusion property for S states that the two processes never execute their critical sections concurrently. Letting CS_1 and CS_2 be the two critical sections, this property is expressed as follows:

$$at\ S \supset \Box\sim(in\ CS_1 \wedge in\ CS_2).$$

The hypothesis *at* S means that the mutual exclusion property has to hold only for those program execution sequences in which S is started at the beginning. (Remember that we allow execution sequences beginning in any state, including one in which both processes are in their critical section.[2])

To prove an assertion of the form $P \supset \Box Q$, one must find an invariant assertion I—that is, an assertion for which $I \supset \Box I$ is true—such that (i) $P \supset I$ and (ii) $I \supset Q$. To see that this implies $P \supset \Box Q$, we simply observe that if the program is started with P initially true, then

—(i) implies that I is true initially;
—the invariance of I then implies that I is always true;
—(ii) then implies that Q is always true.

The implications (i) and (ii) are proved using ordinary logic. Proving that I is an invariant requires reasoning about the program and is discussed next.

We illustrate this method by showing that the program of Figure 5 satisfies the mutual exclusion property, assuming that the critical sections do not modify p_1 or p_2. Although it satisfies the mutual exclusion property, this program does not satisfy any of the other properties one generally requires of a mutual exclusion algorithm. (For example, rather few of the possible execution sequences actually let any process enter its critical section.) However, it does serve as an example.

To prove the mutual exclusion property

$$at\ c \supset \Box\sim(in\ CS_1 \wedge in\ CS_2)$$

[2] At this point, the reader may feel that things would be made much simpler by letting execution sequences always begin in a designated starting state. Both authors did just that in earlier work and now feel that the current approach is better. We refer the reader to [10] for a discussion of the reasons.

```
boolean  p1, p2 ;
c: cobegin
      a1 :  <p1 := true> ;
      b1 :  if  <~p2>  then  CS1 : critical section 1  fi
   ||
      a2 :  <p2 := true> ;
      b2 :  if  <~p1>  then  CS2 : critical section 2  fi
   coend
```

Fig. 5. Oversimplified mutual exclusion program.

we use the following invariant I:

$$(in\ b_1 \supset p_1) \wedge (in\ b_2 \supset p_2) \wedge \sim(in\ CS_1 \wedge in\ CS_2).$$

The reader can check that $at\ c \supset I$, and I obviously implies the mutual exclusion property. Thus the mutual exclusion property is proved once we show that I is invariant.

To prove the invariance of I, one must show that if execution is begun in a program state s in which I is true, then executing any single atomic action in $ready(s)$ leaves I true. This is the basic idea underlying all the aforementioned methods for proving safety properties. We leave it to the reader to verify the invariance of I and of all the invariants we use in our proofs of liveness properties.

We use the notation introduced by Lamport in [9] for expressing safety properties. If S is a program statement and P and Q are immediate assertions, then the formula $\{P\}\ S\ \{Q\}$ has the following meaning:

> If execution begins anywhere in S with P true, then executing the next atomic action of S yields a new state in which either
>
> — control is still within S and P is true, or
> — control is after S and Q is true.

Note that $\{P\}\ S\ \{Q\}$ says nothing about what can happen if an atomic action not in S—perhaps from another process—is executed.

The method of proving safety properties described in [9] is based upon a logic for deriving formulas of the form $\{P\}\ S\ \{Q\}$. This logic does not concern us here. We merely point out that safety properties are deduced from the fact that if S is the entire program, then the formula $\{I\}\ S\ \{I\}$ means that I is invariant. Thus the logic can be used in proving invariants, as required for our method of proving simple safety properties. A method for proving more general safety properties, described briefly in [9], is explained in the appendix.

5. LIVENESS

5.1 The Axioms

In the preceding section, we discussed how one proves safety properties. To prove liveness properties of programs written in our programming language, we need only introduce fairness into our formalism. This can be done with a single rule:

atomic actions always terminate. This means that if the program reaches a state s, then any atomic action in $ready(s)$ will eventually be executed. Since there are two kinds of atomic actions (assignment statements and **while** tests), fairness is expressed formally by the following two axioms:

ATOMIC ASSIGNMENT AXIOM. *For any atomic assignment statement* S:

$$at\ S \rightsquigarrow after\ S.$$

while CONTROL FLOW AXIOM. *For the statement* w: **while** $\langle b \rangle$ **do** s: S **od**,

$$at\ w \rightsquigarrow (at\ s \vee after\ w).$$

Given a method for proving safety properties, these two axioms, together with the laws of temporal logic, enable us to derive all the liveness properties we wish to prove about programs. In the next two sections we give a number of additional proof rules for liveness properties, all of which are derived in the appendix from the liveness axioms and various safety properties. These derivations essentially combine a safety property "nothing not good ever happens" with the two axioms which say "something eventually happens" to conclude that "something good eventually happens".

5.2 Control Flow Rules

The simplest liveness properties are statements about program control flow: if control is at one point, then it must eventually reach some other point. The two liveness axioms are of that form, stating that if control is at the beginning of an atomic operation, then it will eventually be after that operation. We can derive from them the following additional control flow rules. The validity of these rules should be obvious, and they are presented here without proof. Formal derivations of these and our other liveness rules are given in the appendix.

CONCATENATION CONTROL FLOW. *For the statement* S ; T,

$$\frac{at\ S \rightsquigarrow after\ S, \qquad at\ T \rightsquigarrow after\ T}{at\ S \rightsquigarrow after\ T.}$$

cobegin CONTROL FLOW. *For the statement* c: **cobegin** $S \parallel T$ **coend**,

$$\frac{at\ S \rightsquigarrow after\ S, \qquad at\ T \rightsquigarrow after\ T}{at\ c \rightsquigarrow after\ c.}$$

SINGLE EXIT RULE. *For any statement* S:

$$in\ S \supset (\Box\ in\ S \vee \Diamond\ after\ S).$$

Note that the Single Exit Rule is true only because our programming language does not have a **goto** statement. Thus, if control is *in* S, it can only leave S by passing through the control point *after* S.

5.3 More Complex Rules

The above axioms and rules refer only to the control component of the program state and not to the values of variables. We also need rules that describe the

```
boolean  p ;  integer  x ;
a: cobegin
     b: ⟨p := false⟩
   ‖
     c: while ⟨p⟩ do d: ⟨x := x+1⟩ od
   coend
```

Fig. 6. A terminating concurrent program.

interaction between control flow and the values of program variables.[3] For example, consider the program in Figure 6. It consists of two processes: one that sets the variable p false and another that loops as long as p is true. Since the first process eventually sets p false and terminates, the second process eventually terminates. Hence, the entire **cobegin** terminates. However, its termination cannot be inferred directly from the above rules, because it depends upon the interaction between the control flow in the second process and the value of the variable p set by the first process.

We now state and informally justify several rules for reasoning about the interaction between control flow and variable values. More formal proofs are given in the appendix.

First, suppose that the safety property $\{P\}\ \langle S\rangle\ \{Q\}$ holds for the atomic statement $\langle S\rangle$. This tells us that if $\langle S\rangle$ is executed when P is true, then Q will be true immediately after its execution, when control is right after $\langle S\rangle$. The Atomic Liveness Axiom tells us that if control is at $\langle S\rangle$, then $\langle S\rangle$ will eventually be executed. We therefore deduce the following rule:

ATOMIC STATEMENT RULE. *For any atomic statement* $\langle S\rangle$:

$$\frac{\{P\}\ \langle S\rangle\ \{Q\},\qquad \Box(at\ \langle S\rangle \supset P)}{at\ \langle S\rangle \rightsquigarrow (after\ \langle S\rangle \wedge Q).}$$

In the application of this rule, the hypothesis $\Box(at\ \langle S\rangle \supset P)$ must first be proved as a safety property, using the techniques of Section 4.

One might be tempted to write a rule stating that, if $\{P\}\ \langle S\rangle\ \{Q\}$ holds, then $(at\ \langle S\rangle \wedge P) \rightsquigarrow (after\ \langle S\rangle \wedge Q)$. However, this would not be valid. Even if $at\ \langle S\rangle \wedge P$ is true at some point in an execution sequence, P may not be true when $\langle S\rangle$ is actually executed—another process could execute a statement making P false before $\langle S\rangle$ is executed. If this happens, there is no reason why Q should be true upon completion of $\langle S\rangle$. Thus the stronger assumption in the Atomic Statement Rule is necessary.

We next extend the Atomic Statement Rule to nonatomic statements. If $\{P\}\ S\ \{Q\}$ is true for some statement S that will eventually terminate, what will guarantee that Q is true when S terminates? From the meaning of $\{P\}\ S\ \{Q\}$, it is clear that Q will be true upon termination of S if P is true just before the last atomic step of S is executed. This in turn will be true if P is true throughout the execution of S. This gives us the following rule:

[3] In the formal proof of these rules from the liveness axioms, the relationship between the variable values and control flow is derived from safety properties, as discussed in the appendix.

GENERAL STATEMENT RULE.

$$\frac{\{P\}\ S\ \{Q\},\qquad \Box(in\ S \supset P),\qquad in\ S \rightsquigarrow after\ S}{in\ S \rightsquigarrow (after\ S \wedge Q).}$$

Our final two rules involve the atomic test in a **while** statement. Consider the statement w: **while** $\langle B \rangle$ **do** S **od**. The **while** Control Flow Axiom tells us that if control is at w, then it will eventually be at S or after w. We also know that control will go to S only if B is true, and will leave w only if B is false. (This is a safety property of the **while** statement.) Combining these observations, we deduce that if control is at w, then eventually it will be at S with B true, or will be after S with B false, giving us the following rule:

while TEST RULE. *For the statement* w: **while** $\langle B \rangle$ **do** S **od**:

$$at\ w \rightsquigarrow ((at\ S \wedge B) \vee (after\ w \wedge \sim B)).$$

The **while** Test Rule tells us that control must go one way or the other at a **while** statement test. If we know that the value of the test expression is fixed for the rest of an execution sequence, then we can predict which way the test will go. In particular, we can deduce the following:

while EXIT RULE. *For the statement* w: **while** $\langle B \rangle$ **do** S **od**:

$$(at\ w \wedge \Box(at\ w \supset B)) \rightsquigarrow at\ S;$$
$$(at\ w \wedge \Box(at\ w \supset \sim B)) \rightsquigarrow after\ w.$$

5.4 A Trivial Example

We now prove that the example program of Figure 6 terminates—that is, we prove $at\ a \rightsquigarrow after\ a$. The proof is described by the lattice of Figure 7. The numbers attached to the lattice refer to the comments in the text.

1. This step follows from the Atomic Statement Rule applied to statement b, using the formula $\{true\}\ b\colon \langle p := false \rangle\ \{\sim p\}$. This is obviously a valid formula, since no matter what state b is started in, it ends with p having the value *false*.

2. This is a consequence of the safety property $(after\ b \wedge \sim p) \supset \Box(after\ b \wedge \sim p)$, which states that once control reaches *after* b with p false, it must remain there (it has no place else to go) and p must stay false (no assignment in the program can change its value).

3. For this program, control must be either in c or after c. This step separates the two cases. Formally, it follows from the fact that the predicate $in\ c \vee after\ c$ is true in any program state. At this point we use the box abbreviation to indicate that $\Box(after\ b \wedge \sim p)$ is true at all descendants of this node.

4. This follows from the fact that $in\ c$ and $at\ c \vee at\ d$ are equivalent. Again, the branch in the lattice separates the cases.

5. This follows from the Atomic Liveness Axiom, applied to statement d, plus the fact that *after* d is equivalent to *at* c.

6. The enclosing box tells us that $\Box\sim p$ is true at this node. Thus, we can apply the **while** Exit Rule to infer that control eventually leaves the **while** loop, making *after* c true.

7. This is a trivial implication. The enclosing box tells us that $\Box$ *after* b is true, and $\Box$ *after* b implies *after* b.

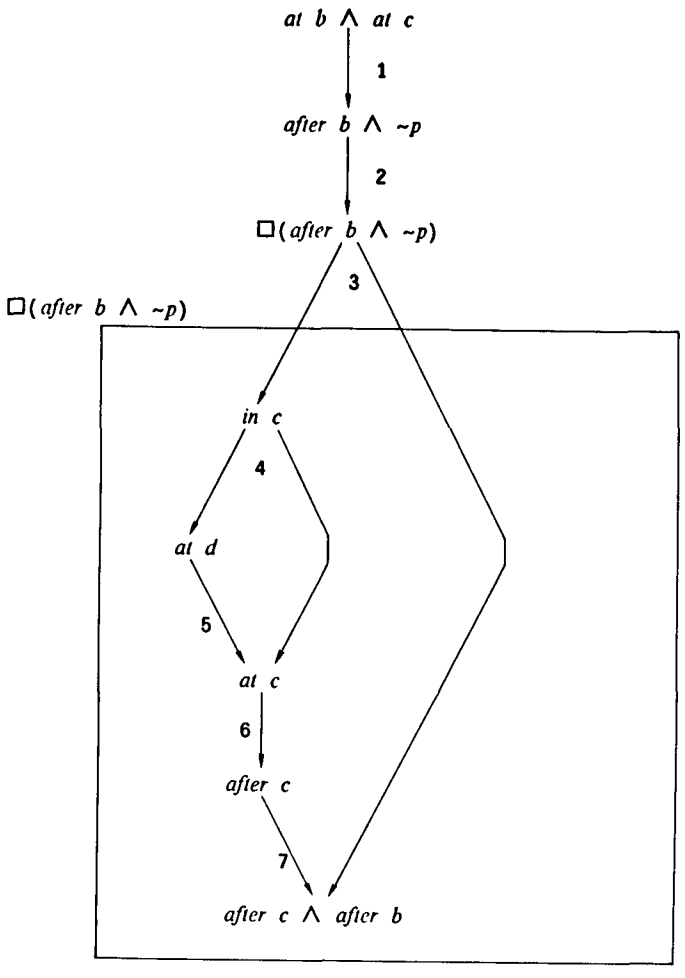

Fig. 7. Proof lattice for program of Figure 6.

In this program, the termination of the **while** loop was proved by showing that $\Box\sim p$ must eventually become true. This enabled us to use the **while** Exit Rule to show that control must eventually leave the loop. But suppose that we wanted to verify termination for the similar program in Figure 8. In this program $\Box\sim p$ does not eventually hold, because after control leaves the loop, p is reset to true. How can we hope to verify such a program, since the **while** Exit Rule requires us to know $\Box\sim p$ in order to prove termination? The answer is given by the proof lattice in Figure 9. It illustrates a type of proof by contradiction that we use quite often. We start by using the Single Exit Rule to break the proof into two cases (this is the first branch in the lattice). In one of those cases, control remains forever inside the loop. In this case, we can establish that eventually $\Box\sim p$ must be true, and then our reasoning is essentially the same as before.

The proofs above were quite detailed, with each application of a proof rule cited explicitly. This is the sort of proof that mechanical verifiers do well but people find unbearably tedious. If people as well as machines are to be able to use

```
boolean  p ; integer  x ;
a: cobegin
      b: ⟨p := false⟩
   ‖
      c: while ⟨p⟩ do d: ⟨x := x+1⟩ od ;
      e: ⟨p := true⟩
   coend
```

Fig. 8. Another terminating program.

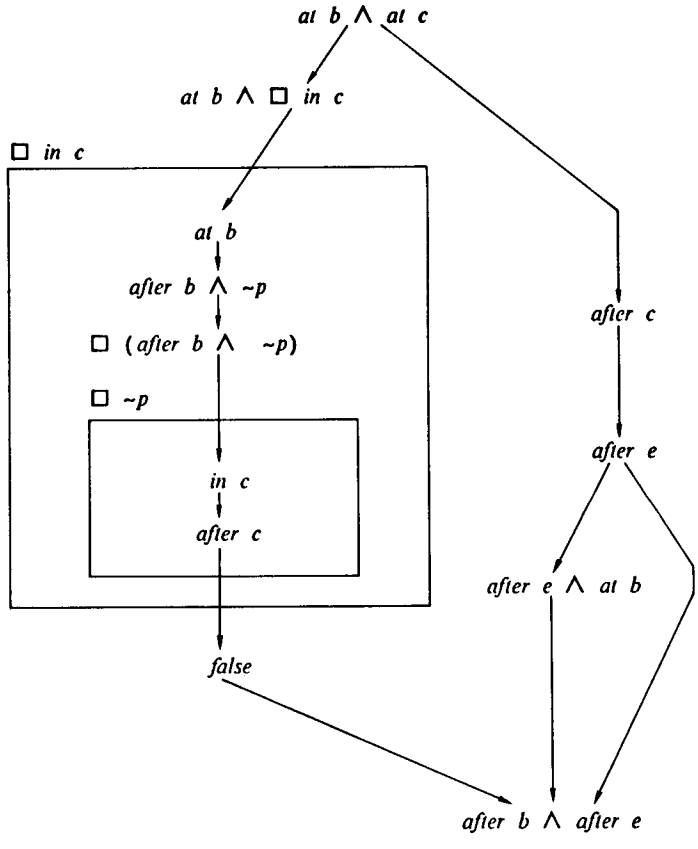

Fig. 9. Proof lattice for program of Figure 8.

a proof method, they must be able to omit obvious details. For example, the reasoning in step 7 of Figure 7 is so trivial that it does not really need to be explained—so steps 6 and 7 can be combined and the "*after c*" node of Figure 7 eliminated. Also, steps 4–6 of the proof simply show that if control is in c and $\Box\sim p$ holds, then control must eventually be after c. This is such an obvious conclusion that it could be reached in a single step. One often combines a number of proof rules when they describe simple progress of control in a single process.

However, informal reasoning about concurrent programs often leads to errors, so we must be careful when we leave out steps in a proof. Fortunately, some

kinds of informal reasoning are relatively safe. We recommend the following guidelines for constructing informal proofs:

(1) Each step in the proof should combine actions from just one process.

(2) To conclude that evaluating an expression E yields a value v, one must prove $\Box(E = v)$. For example, in the above proof we were able to conclude that the **while** test evaluated to *false*, and the loop therefore ended, because we had proved $\Box\sim p$. (The $\Box$ is needed here for the same reason as in the Atomic Statement Rule.)

In the rest of the paper, we often omit proof details that we feel are obvious, following these guidelines. All the missing steps can be proved directly from the axioms and rules we have given, and we urge the reader to do so if he is uncomfortable with the proof. It has been our experience that proof lattices help avoid mistakes in informal proofs by imposing a structure that makes it easy to see where care is needed.

6. AN EXAMPLE: MUTUAL EXCLUSION

6.1 The Problem

We illustrate the use of these rules by proving a liveness property for a solution to one of the standard problems in concurrent programming: providing mutually exclusive access to critical sections. We must construct a program with two processes, each repeatedly executing a noncritical and a critical section. The content of these sections may be arbitrary, except that the critical sections are guaranteed to terminate. Both processes are to be started in their noncritical sections.

The solution must, of course, satisfy the mutual exclusion property described in Section 4. However, a solution is useless unless it also satisfies some liveness property. (For example, mutual exclusion can be achieved by merely halting both processes.) Typically, one requires that under certain conditions, a process that is trying to enter its critical section will eventually succeed. We construct a solution that gives priority to Process 1—meaning that Process 1 is always guaranteed eventual entry to its critical section, but Process 2 could be forever locked out of its critical section if Process 1 keeps executing its critical section often enough.

The requirement that Process 1 is always guaranteed eventual entry to its critical section can be stated more precisely as follows:

> It is always the case that after Process 1 finishes executing its noncritical section, it will eventually enter its critical section.

This condition is a simple liveness property having the form

"after executing noncritical section $\rightsquigarrow$ executing critical section".

To write this requirement as a formal temporal logic assertion to be proved, we must remember to include the initial condition, specifying that the requirement need only hold for execution sequences starting from the beginning of the program. We also include as an explicit hypothesis the requirement that Process 2's critical section is always guaranteed to terminate. (This requirement does not

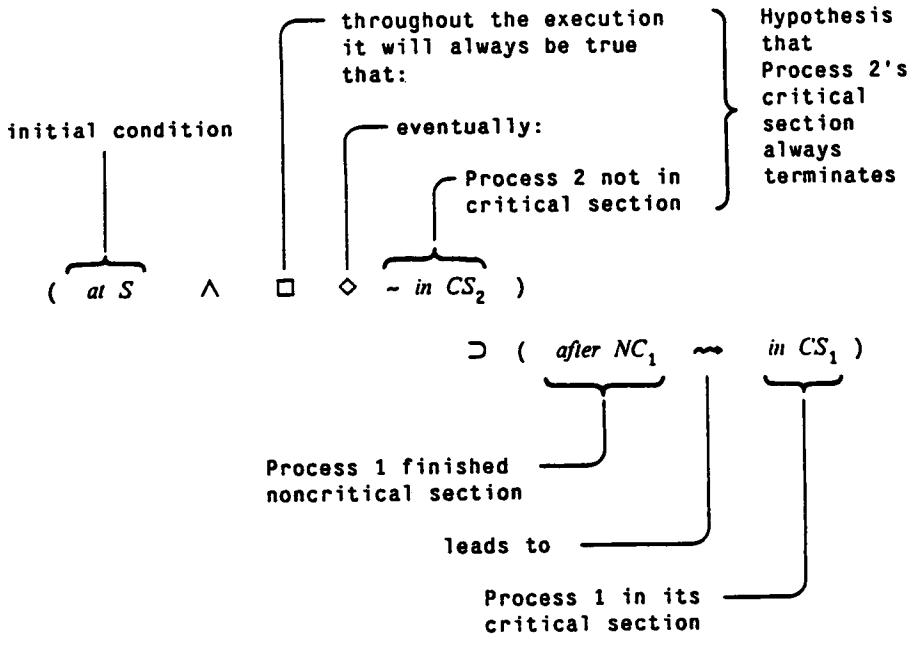

Fig. 10. Process 1 liveness property.

depend upon the termination of Process 1's critical section.) Letting NC_i and CS_i be the noncritical and critical sections of Process i, and S the entire program, we get the assertion shown in Figure 10, annotated to explain the meaning of each of its clauses.

The Process 1 liveness property could be achieved rather simply by permanently barring Process 2 from its critical section. Such a solution is clearly not what is intended, so some other requirement is needed. A little thought will reveal that giving Process 1 priority tacitly implies that Process 2 should be able to enter its critical section whenever Process 1 stays in its noncritical section. This condition can be expressed as follows:

> It is always the case that, if Process 2 has finished its noncritical section and Process 1 remains forever in its noncritical section, then Process 2 will eventually enter its critical section.

Remembering to add the initial condition as an hypothesis, this gives us the formal property shown in Figure 11.

The Process 2 liveness property only guarantees Process 2 entry to its critical section if Process 1 remains forever in its noncritical section. One might reasonably want a stronger condition that guarantees Process 2 entry to its critical section if Process 1 remains in its noncritical section long enough. This condition cannot be expressed in our temporal logic, since there is no way to express "long enough". It is actually the case that any program satisfying the Process 2 liveness property must also satisfy such a "long enough" property. This is because programs cannot test the future; program statements like "if p will never become true then . . ." lead to logical contradictions. However, further discussion of this would lead us too far from our main subject.

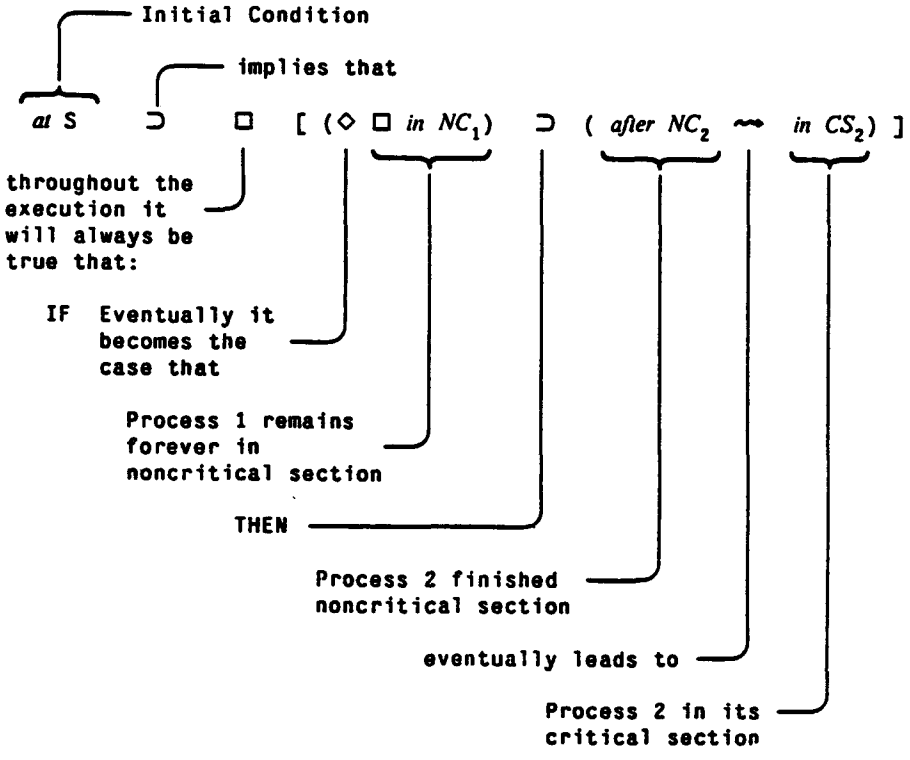

Fig. 11. Process 2 liveness property.

6.2 The Solution

Our solution uses the same method for achieving mutual exclusion as does the simple program of Figure 5. Each process i has a flag p_i, and executes the protocol "set my flag true and check that the other process's flag is false" before entering its critical section. The simplest way to use this protocol is to precede each process's critical section by the following:

Procedure A:
Set my flag true and wait until the other process's flag is false.

However, this is unsatisfactory because if both processes concurrently tried to enter their critical sections, then they could both wait forever—a situation known as *deadlock.*

Deadlock can be prevented by having each process do the following before entering its critical section:

Procedure B:
Set my flag to true;
If other process's flag is true:
then set my flag false and try again
else proceed.

Although this avoids deadlock, it is unsatisfactory because each process might be unlucky enough to examine the other process's flag only when it is true, in which

```
boolean p1, p2
S: cobegin
     S1: begin
           p1 := false;
         w1: while < true >
             do
               NC1: noncritical section 1 ;
               a1 : < p1 := true > ;
               b1 : while < p2 > do c1: <skip> od ;
               CS1: critical section 1 ;
               d1 : < p1 := false >
             od
         end ∎
     S2: begin
           p2 := false;
         w2: while < true >
             do
               NC2: noncritical section 2 ;
               a2 : < p2 := true > ;
               b2 : while < p1 >
                    do c2: < p2 := false > ;
                       e2: while < p1 > do <skip> od ;
                       f2: <p2 := true >
                    od
               CS2: critical section 2 ;
               d2 : < p2 := false >
             od
         end
coend
```

Fig. 12. Mutual exclusion algorithm.

case no process ever enters its critical section. This type of behavior is called "livelock" or "tempo blocking".

Note that the absence of deadlock is a safety property, since it implies that a bad state (one in which both processes are waiting) cannot occur. Hence, it can be proved with the method described in Section 4. However, the absence of livelock is a liveness property, and requires a different proof method.

Our solution, given in Figure 12, uses the following approach. To enter its critical section, Process 1 uses Procedure A—not resetting its flag until it leaves its critical section. When Process 2 wants to enter its critical section, it executes a modified version of Procedure B: if it finds Process 1's flag true, then it waits until that flag becomes false before trying again. We assume that the noncritical and critical sections do not modify the values of the variables p_1 and p_2.

6.3 The Correctness Proof

The proof that the program of Figure 12 satisfies the mutual exclusion property is similar to the proof for the program of Figure 5 and is left to the reader. We prove the two liveness properties.

We begin with an informal proof of the Process 1 liveness property, which is illustrated by the diagram of Figure 13. If Process 1 is after its noncritical section, then either it will eventually enter its critical section or else it will remain forever in b_1 with p_1 true. We show by contradiction that the latter is impossible. Suppose Process 1 remains in b_1 with p_1 true. Then, since Process 2's critical section always terminates, it either will reach e_2 or will remain forever in its noncritical section with p_2 false. However, if it reaches e_2, it will remain there forever with p_2 false because p_1 is true. In either case, p_2 remains false forever. But this is impossible, because in that case Process 1 must eventually leave loop b_1, and this is the required contradiction.

The rigorous proof of the Process 1 liveness property is obtained by formalizing this argument. The informal reasoning relied upon certain simple safety properties of the program—for example, that p_2 is false while Process 2 is in its noncritical section. Our first step is to prove these safety properties. This is done by demonstrating the invariance of the following predicate I, which relates the values of p_i to control points in process i, and also asserts that S never terminates:

$$\begin{aligned} I\colon\ & (in\ b_1 \supset p_1) \wedge (in\ NC_1 \supset \sim p_1) \\ & \wedge (in\ e_2 \supset \sim p_2) \wedge (in\ NC_2 \supset \sim p_2) \\ & \wedge in\ S_1 \wedge in\ S_2. \end{aligned}$$

The proof of invariance involves simple local reasoning about each process and is left to the reader. The invariance of I, plus the fact that I is true initially, imply that $at\ S \supset \Box I$. Thus we have shown that

$$(at\ S \wedge \Box\Diamond \sim in\ CS_2) \rightsquigarrow (\Box I \wedge \Box\Diamond \sim in\ CS_2). \tag{1}$$

We also need the fact that control in both processes eventually reaches and remains inside the **while** loops—that is,

$$in\ S_1 \rightsquigarrow \Box\ in\ w_1, \qquad \text{and} \qquad in\ S_2 \rightsquigarrow \Box\ in\ w_2. \tag{2}$$

This can be proved by combining simple liveness arguments that $at\ S \rightsquigarrow in\ w_i$ with the fact that $in\ w_i$ is invariant.

Our next step is to prove that

$$(\Box I \wedge \Box\Diamond \sim in\ CS_2) \supset (at\ a_1 \rightsquigarrow in\ CS_1). \tag{3}$$

Combining eqs. (1) and (3) gives the Process 1 liveness property. Figure 14 contains a proof lattice for eq. (3); the steps are explained below.

1. By TL4(a), the assertion

$$\Box(I \wedge \Diamond \sim in\ CS_2)$$

is equivalent to the hypothesis of eq. (3). Since the hypothesis is a "henceforth" property that can be used throughout the proof, we extend our notation slightly and attach it to a box containing the entire lattice.

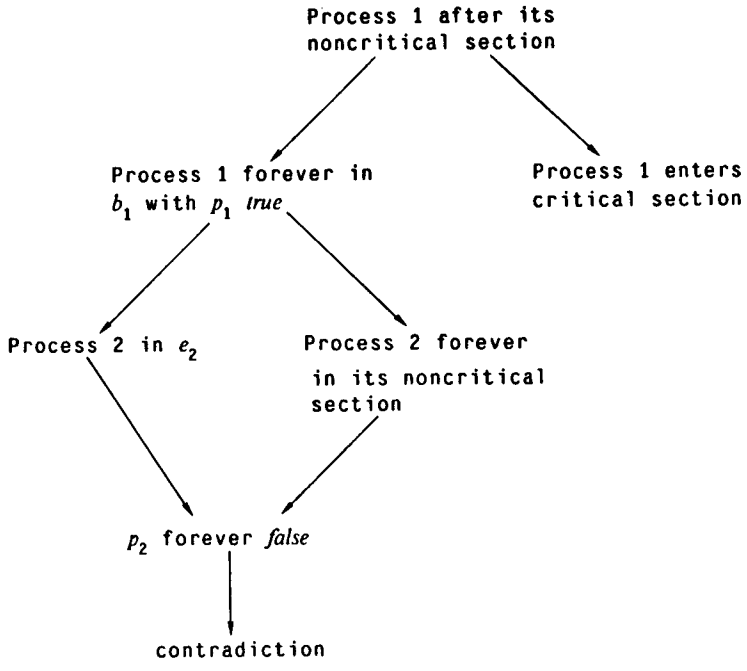

Fig. 13. Informal proof of Process 1 liveness property.

2. This is a simple application of the Atomic Liveness Axiom.

3. This follows from the Single Exit Rule. It sets up two cases: either control eventually enters the critical section and we are immediately finished, or it remains forever in b_1.

4. We use eq. (2) and the *in* S_2 clause from I on the outer box.

5. Since we can assume $\Box I$ (by the outer box), we have $\Box(in\ b_1 \supset p_1)$. By TL5, $\Box$ *in* b_1 then implies $\Box p_1$. Hence, we can create a new box, in which we assume $\Box p_1$.

6. Since *in* w_2 is equivalent to *at* $a_2 \vee$ *in* $b_2 \vee$ *in* $CS_2 \vee$ *at* $d_2 \vee$ *in* NC_2, this step follows from TL9, with *at* $a_2 \vee$ *in* b_2 substituted for Q and *in* $CS_2 \vee$ *at* $d_2 \vee$ *in* NC_2 substituted for P.

7. Here we are using local reasoning about control flow in Process 2, under the assumption $\Box p_1$. Note that this assumption completely determines the direction that will be taken at each **while** test. Thus it is easy to see that if control is anywhere in a_2 or b_2, then it will eventually reach e_2 and remain there.

8. We can assume $\Box I$, so we have $\Box(in\ e_2 \supset \sim p_2)$. By TL5, $\Box$ *in* e_2 then implies $\Box\sim p_2$.

9. It follows from the **while** Exit Rule that $\Box\sim p_2$ implies that Process 1 must eventually leave statement b_1. Since this node is inside a box labeled $\Box$ *in* b_1, we have a contradiction.

10. Here we use the trivial implication *false* $\supset P$ for any P.

11. Since we are assuming $\Diamond \sim in\ CS_2$, the Single Exit Rule implies that *in* $CS_2 \rightsquigarrow$ *at* d_2, and the Atomic Liveness Axiom implies that *at* $d_2 \rightsquigarrow$ *at* NC_2.

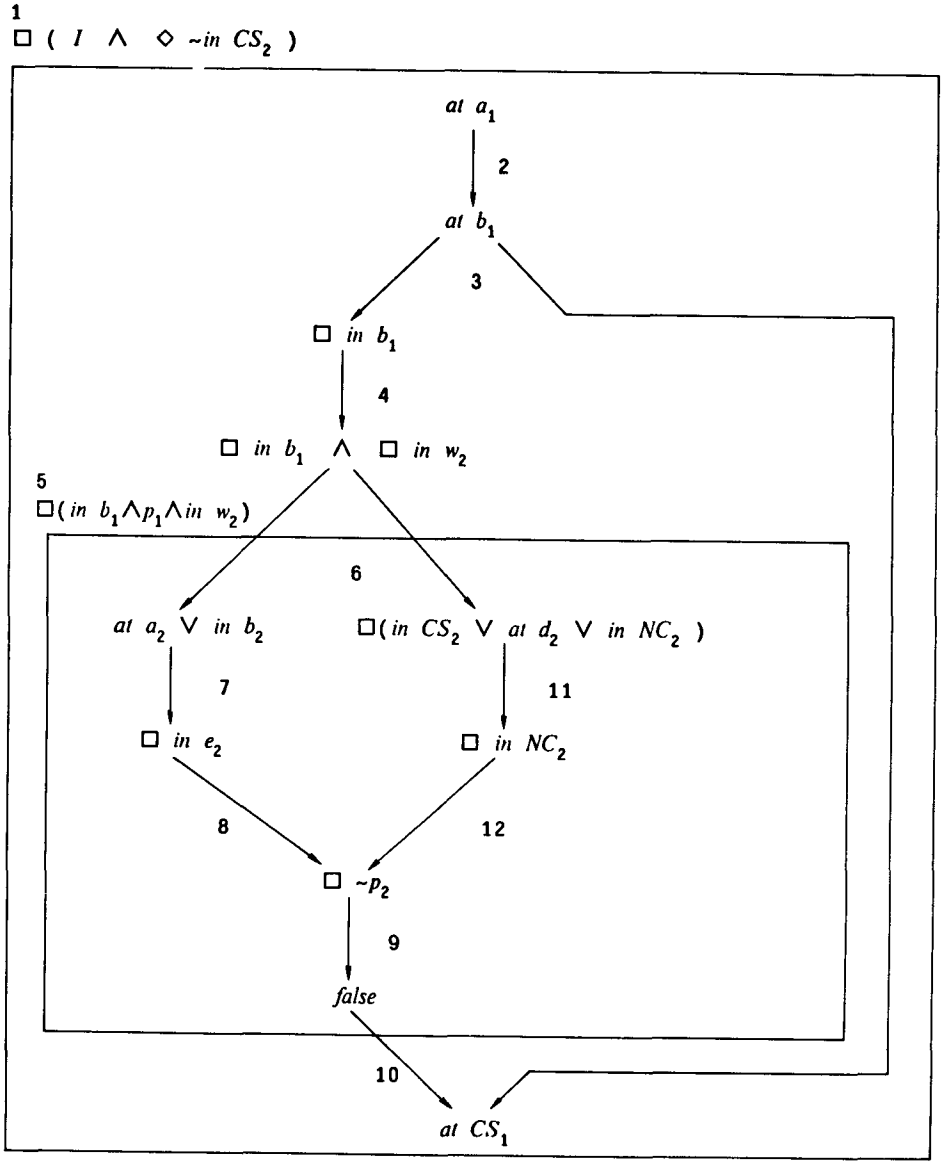

Fig. 14. Lattice proof of Process 1 liveness property.

12. It follows from I that $in\ NC_2 \supset \sim p_2$.

This completes the proof of the Process 1 liveness property. To prove the Process 2 liveness property, we use the proof lattice of Figure 15 to show that $\Box(I \wedge \Diamond\Box \sim in\ CS_1)$ implies $at\ a_2 \rightsquigarrow in\ CS_2$. Using this result, we can prove the Process 2 liveness property with the same kind of reasoning used for the Process 1 liveness property. The steps in the lattice are explained below.

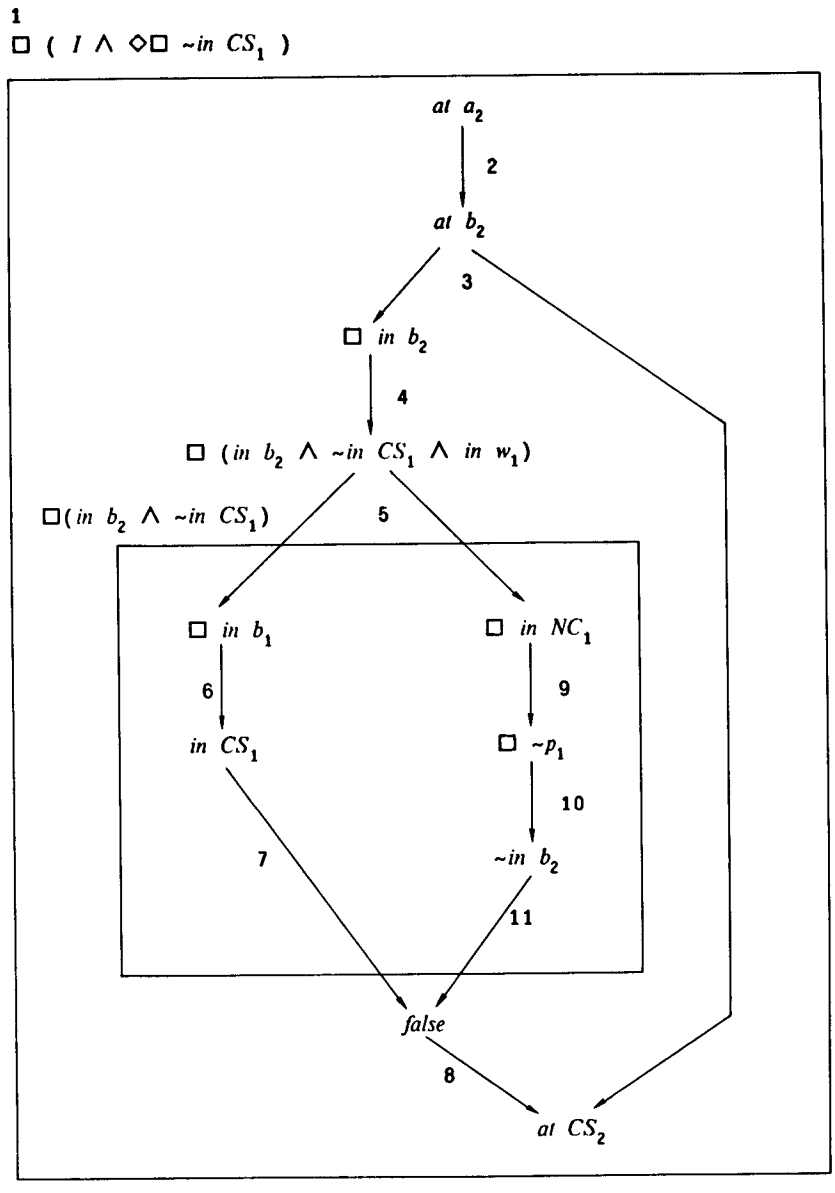

Fig. 15. Lattice proof of Process 2 liveness property.

1. As in the preceding lattice, we attach the assumption to a box that contains the entire proof.

2. This step is based on control flow reasoning for Process 2.

3. As before, we use the Single Exit Rule to break our proof into two cases, and prove by contradiction that control cannot remain forever in the loop.

4. We use eq. (2) and the *in* S_1 clause from I on the outer box, together with $\Diamond\Box \sim in\ CS_1$ also on the outer box.

5. Simple control flow reasoning about Process 1 shows that if it stays in w_1 and never enters its critical section, then eventually it must stay in b_1 or in its noncritical section.

6. The Process 1 liveness property, which we have already proved, guarantees that Process 1 will be able to enter its critical section if Process 2 does not stay forever inside its critical section. But here we may assume that $\Box$ *in* b_2, so Process 1 must eventually enter its critical section.

7. Having Process 1 in its critical section gives a contradiction, since within the inner box we can assume $\Box \sim in\ CS_1$.

8. Here we are again using the trivial fact that *false* implies anything.

9. This follows from the assumption that $\Box I$ holds.

10. Local reasoning about Process 2 shows that control cannot remain forever in b_2 if p_1 remains false.

11. Process 2's leaving b_2 contradicts the assumption from the inner box.

This completes the proof of the two liveness properties for the program of Figure 12.

7. SYNCHRONIZATION PRIMITIVES

For programs written in our simple language, a nonterminated process can never stop. A process can wait for something to happen only by executing a "busy waiting" loop. Thus, the processes in our mutual exclusion algorithm had to wait in **while** $\langle p_i \rangle$ **do** ... **od** loops. This is undesirable in a multiprogramming environment, since a waiting process could tie up the processor in a pointless loop. It is common in such environments to provide synchronization primitives that release the processor until the desired condition is true. Our basic method for proving liveness properties can be applied to programs using such synchronization primitives. We illustrate this by considering the well-known *semaphore* primitives introduced by Dijkstra [2].

7.1 The Fair Semaphore

A semaphore is a nonnegative integer variable that can be accessed by two primitive operations: P and V. A V operation increments the value of the semaphore by one, while a P operation decrements it by one. However, since the semaphore's value must be nonnegative, a P operation can only be performed when the value is positive. This means that if a process's control is at a $P(s)$ operation when the value of the semaphore s is zero, then the process must wait until another process has performed a $V(s)$ operation before it can proceed. A $V(s)$ operation can always be performed.

In order to prove properties of programs that use the semaphore primitives, we need a precise definition of the semantics of these primitives. It is not hard to define their safety properties, and various axiomatizations have been given—for example, in [5] and [9]. Specifying their liveness properties presents a more interesting problem. In fact, the liveness properties of the semaphore operations were not fully specified in their original definition, and several different versions have been implemented. The differences result from different methods of choosing which process to activate when a $V(s)$ operation is executed and several processes

are waiting to execute $P(s)$ operations. Lipton [12] distinguishes several different ways of making that choice, each leading to different liveness properties for the semaphore operations. For our example, we assume that the choice is made "fairly".

We define the P and V operations to be atomic. The axioms defining the V operation are quite straightforward:

V OPERATION AXIOMS. *For the statement l*: $\langle V(s) \rangle$:

Safety: $\{Q[s+1 \,/\, s]\}\ V(s)\ \{Q\}$
Liveness: $at\ l \rightsquigarrow after\ l$

where $Q[s+1 \,/\, s]$ *is the formula obtained by replacing every free occurrence of* s *in* Q *by* $s+1$.

The safety axiom states that the V operation adds 1 to the semaphore; the liveness axiom that the V operation always terminates.

The subtle part of axiomatizing the semaphore operations lies in specifying under what conditions a P operation must eventually terminate. It is obviously not enough that the semaphore be positive when control reaches the P operation, because it could be decremented by another process before that operation is executed. The following axiom states that a process trying to perform a P operation will not have to wait forever while other processes keep executing P operations on that semaphore:

P OPERATION AXIOMS. *For the statement l*: $\langle P(s) \rangle$:

Safety: $\{Q[s-1 \,/\, s]\}\ P(s)\ \{Q \wedge s \geq 0\}$.
Liveness: $(at\ l \wedge \Box\Diamond(s > 0)) \rightsquigarrow after\ l$.

The safet axiom states that P decrements the semaphore, and that the semaphore's value will be nonnegative after the operation is executed. This prevents the operation from being executed when the semaphore is 0. The liveness axiom states that a P operation will be executed if the semaphore repeatedly assumes a positive value. (The formula $\Box\Diamond(s > 0)$ states that s is positive infinitely often.)

The Atomic Liveness Axiom and the Atomic Statement Rule of Section 5 hold for all the atomic statements of our original language. They also hold for a $\langle V(s) \rangle$ statement, but do not hold for a $\langle P(s) \rangle$ statement. However, the remaining rules from Section 5—in particular, the Single Exit and General Statement Rules—do hold for these new statements. (Their proofs, given in the appendix, are not affected by the introduction of the new statements into the language.)

7.2 A Simple Example

We illustrate the use of these semaphore axioms with the simple mutual exclusion algorithm of Figure 16. To prove mutual exclusion, we first show the invariance of the following immediate assertion:

I: $0 \leq s \leq 1 \wedge 1 - s =$ number of processes i such that $near\ CS_i$ is true

where $near\ CS_i \equiv in\ CS_i \vee after\ CS_i$.

```
semaphore s ;
a0: < s := 1 > ;
cobegin
        w1: while < true >
            do
              NC1: noncritical section 1 ;
              a1 : < P(s) > ;
              CS1: critical section 1 ;
              d1 : < V(s) >
            od
 ∥
        while < true >
            do
              NC2: noncritical section 2 ;
              a2 : < P(s) > ;
              CS2: critical section 2 ;
              d2 : < V(s) >
            od
coend
```

Fig. 16. Mutual exclusion algorithm using semaphores.

(We could, of course, write a more formal statement of the clause "$1 - s =$ number of . . .".) This allows us to conclude that

$$at\ a_0 \supset \Box I, \tag{4}$$

from which mutual exclusion follows. The details of this safety proof are straightforward and are omitted.

Our liveness axiom for the P operation guarantees that any process that wants to enter its critical section will eventually do so, unless the other process remains forever inside its critical section. Thus, we have the following liveness property for Process 1:

$$(at\ a_0 \wedge \Box\Diamond \sim in\ CS_2) \supset (at\ a_1 \rightsquigarrow in\ CS_1). \tag{5}$$

Note that this is essentially the same as the Process 1 liveness property of the previous example. A similar liveness property holds for Process 2.

To prove eq. (5), we first prove the following formula, which states that the semaphore repeatedly assumes a positive value unless some process stays inside its critical section forever:

$$(\Box I \wedge \Box\Diamond \sim near\ CS_1 \wedge \Box\Diamond \sim near\ CS_2) \supset (s = 0 \rightsquigarrow s = 1). \tag{6}$$

Its proof is given by the lattice of Figure 17, with the steps explained below.

1. This is the usual introduction of an assumption to be used in the remainder of the proof.

2. The safety invariant implies that when the semaphore has the value 0, one of the processes is near its critical section.

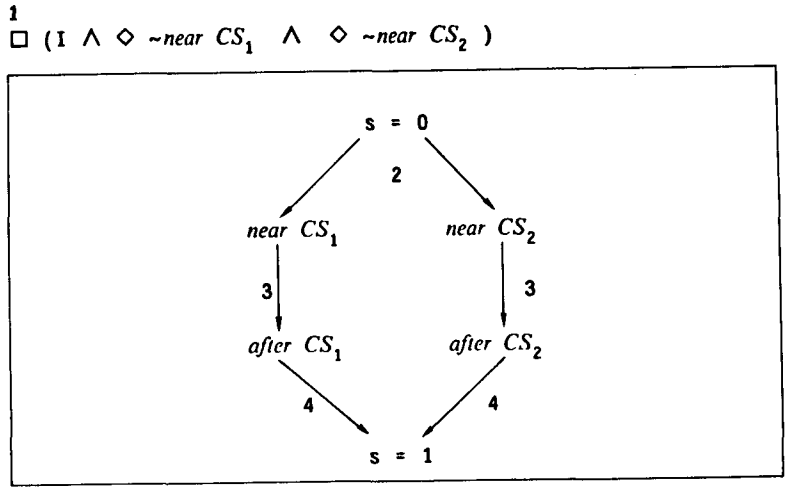

Fig. 17. Proof lattice for eq. (6).

3. The assumption that a process does not remain in its critical section, together with the Single Exit Rule, guarantees that if a process is near its critical section, then it is eventually after it.

4. The liveness axiom for the V operation implies that if control is after CS_i, then the subsequent V operation will terminate. We can then apply the General Liveness Rule to conclude that s will equal 1 when the $V(s)$ operation terminates. This is based on the truth of $\{s \geq 0\}\ V(s)\ \{s > 0\}$ and on the fact that the invariant implies that s is always 0 or 1.

Using eq. (6), we now prove eq. (5) with the lattice of Figure 18, whose steps are explained below.

1. As usual, we put a box around the whole lattice labeled with the hypothesis.

2. This is an application of the Single Exit Rule, noting that $at\ a_1 = in\ a_1$ since a_1 is atomic.

3. Since $at\ a_1$ implies $\sim near\ CS_1$, we can conclude from eq. (6) that $s = 0 \leadsto s = 1$. Since s is always nonnegative, this implies $\Box\Diamond(s > 0)$.

4. The liveness axiom for the $P(s)$ operation implies that if control is at a_1 and $\Box\Diamond(s > 0)$ is true, then eventually the $P(s)$ operation will be executed and control will be after a_1, contradicting the assumption of the inner box that control is forever at a_1.

This completes the proof of eq. (5), the liveness property for Process 1. The corresponding property for Process 2 is proved in exactly the same way.

7.3 Other Semaphores

The above proof may have seemed rather long for the simple program of Figure 16, whose correctness seems obvious. However, liveness properties for programs using semaphores tend to be rather subtle, and there are quite reasonable ways of defining the semaphore operations for which the algorithm would not guarantee the liveness property (5).

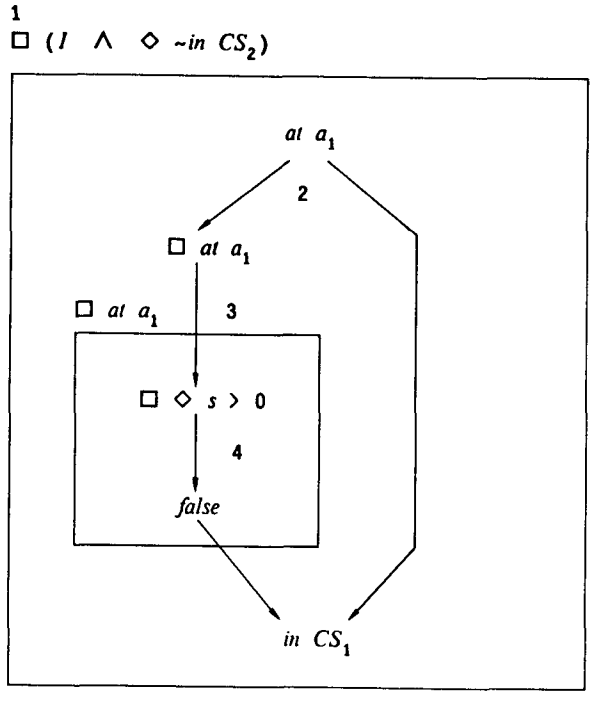

Fig. 18. Proof lattice for semaphore program.

The P operation liveness axiom we gave above is called the *fairness* axiom. Some semaphore implementations are not fair; these implementations are described by a weaker liveness axiom.

WEAK LIVENESS AXIOM. *For the statement l*: $\langle P(s) \rangle$:

$$(at\ l \wedge \Box(s > 0)) \rightsquigarrow after\ l.$$

This axiom states that a process cannot wait forever at a P operation if the semaphore remains positive. However, it does not prevent the process from waiting forever if other processes keep performing P operations that reset the value to 0. This axiom is not enough to guarantee property (5) for our mutual exclusion algorithm, because it does not rule out the possibility that Process 1 waits forever while Process 2 repeatedly enters and leaves its critical section.

Two other kinds of semaphores have also been proposed, defined by the following properties:

First-Come-First-Served. Waiting processes must complete their P operations in the order in which they began them.

Latching Property. If a process executes a V operation when there are other processes waiting on the semaphore, then it cannot complete a subsequent P operation before one of the waiting processes completes its P operation.

These are safety properties, since they state that "something bad cannot happen". (This becomes more apparent if the first property is expressed as

"processes cannot complete their P operations out of order".) They are usually combined with the weak liveness property. For a bounded number of processes, first-come-first-served plus weak liveness implies fairness.

In order to state these properties precisely, some additional structure must be given to the semaphore operations. In particular, the P operation cannot be atomic, because a process must do something after it reaches the P operation to announce that it is waiting. The formalization of these properties is beyond the scope of this paper.

8. CONCLUSION

We have considered two kinds of correctness properties for concurrent programs:

Safety properties—stating that some assertion always holds.
Liveness properties—stating that some assertion will eventually hold.

We have used temporal logic to provide a simple, uniform logical formalism for expressing and reasoning about these properties.

Several methods for proving safety properties of concurrent programs have appeared. In this paper, we have presented a method for proving liveness properties. The method seems powerful enough to prove a large class of such properties, although we have not characterized precisely what can and cannot be proved by it. The method provides a straightforward way of turning an intuitively clear informal proof—one that captures our understanding of why the program works—into a logically rigorous proof. We have found it better to separate proofs of safety and liveness properties, rather than to combine them as in [16].

Our method is independent of any particular programming language. Each language has its own axioms, but the basic proof method does not change. For example, synchronization mechanisms other than semaphores have liveness axioms that describe their conditions for termination. If programs based upon the same algorithm were written in two different languages, we would expect the proof lattices used in their correctness proofs to have the same structure.

Safety and liveness are not the only kinds of program properties. There are others that we have not considered, such as the following:

(1) Some assertion will become true within a certain number of process execution steps.
(2) Some assertion might possibly become true.
(3) The program is equivalent to some other program.

In fact, these properties cannot be expressed in our temporal logic. Their omission was deliberate, since we could have chosen other modal logics in which such properties are expressible. The simplicity of our approach is due largely to a very careful limitation of its scope. We have deliberately eschewed unnecessary generality, devising a method for proving exactly what we want to prove and no more. We have been guided by the desire to provide a practical method for proving properties of concurrent programs, and have rejected irrelevant generality.

We do not wish to imply that we have said the last word on proving properties of concurrent programs. We feel that we have provided the foundation for a useful system for proving properties of real concurrent programs. However,

constructing a useful edifice upon this foundation will require the ability to handle more sophisticated language features, and further work in this direction is needed.

The logical system described in this paper is an *endogenous* one, meaning that each program defines its own separate formal system. In an endogenous system, one is always reasoning about the entire program. An *exogenous* system is one in which there is a single formal system within which one can reason about a large class of programs, such as all programs written in a certain language. In such a system, the formulas explicitly mention programs. The logic of [9] for manipulating $\{P\}\ S\ \{Q\}$ formulas is an example of an exogenous system. Such systems are convenient because they provide a formal framework for structured proofs, in which one derives properties of a statement from the properties of its components.

The temporal logic we have used is endogenous, and there is no way to base an exogenous system upon it. For example, the property $\{P\}\ S\ \{Q\}$ cannot be expressed in this temporal logic if S is not the entire program. We are currently developing an exogenous temporal logic for proving both safety and liveness properties of concurrent programs.

Dynamic logic [18] is a well-known exogenous logic for programming language semantics. However, it is less attractive than temporal logic for reasoning about concurrent programs because it is based on a branching time rather than a linear time model of computation, a difference discussed in [10]. *Process logic* [6, 17] is an exogenous logic that can express characteristics of both the linear and branching models, so it is more general than temporal logic. However, this is the sort of generality that we see as inappropriate for a practical verification method.

APPENDIX. DERIVED PROOF RULES

We now prove the validity of the derived proof rules of Section 5. To do this, we have to prove a more general kind of safety property than has been discussed so far: properties of the form $(P \wedge \Box R) \supset \Box Q$. The method of proving these properties is similar to that used for ordinary safety properties and involves finding an assertion I satisfying the following conditions:

- $(P \wedge R) \supset I$,
- $(I \wedge R) \supset Q$, and
- $(I \wedge \Box R) \supset \Box I$.

The last condition is a generalized form of invariance. To verify it, one shows that starting in a state in which $I \wedge R$ is true, executing a single atomic action yields a new state in which I is true or R is false. In the notation of [9], this is expressed by the formula $R \vdash \{I\}\ S\ \{I\}$, where S is the entire program. (This actually proves the stronger assertion $I \supset R \,\Box\, I$, where the binary operator $\Box$ is defined in [10], but we will not need this stronger assertion.) When a proof depends on such a generalized safety property, we give the assertion I and argue informally that it is invariant if R is always true.

CONCATENATION CONTROL FLOW. *For the statement* S ; T,

$$\frac{at\ S \rightsquigarrow after\ S, \qquad at\ T \rightsquigarrow after\ T}{at\ S \rightsquigarrow after\ T.}$$

PROOF. The definition of *at* implies that *after* $S = at\ T$, so the first hypothesis can be rewritten as $at\ S \rightsquigarrow at\ T$. The second hypothesis, together with the transitivity of $\rightsquigarrow$ (TL7), then implies the conclusion. □

SINGLE EXIT RULE. *For any statement* S:

$$in\ S \supset (\Box\ in\ S \vee \Diamond\ after\ S).$$

PROOF. The proof is based on the following safety property:

$$(in\ S \wedge \Box \neg after\ S) \supset \Box\ in\ S. \tag{7}$$

Intuitively, this states that if control starts in S and never reaches the control point *after* S, then it stays in S forever. It is proved by observing that any atomic action in our programming language transforms a state where *in* S is true to a state where either *in* S or *after* S is true.

The safety property (7) can be rewritten as

$$in\ S \supset (\Box\ in\ S \vee \neg\Box \neg after\ S).$$

The duality of $\Box$ and $\Diamond$ implies that this is equivalent to

$$in\ S \supset (\Box\ in\ S \vee \Diamond\ after\ S).$$

Finally, we apply TL2 to obtain the desired result. □

cobegin CONTROL FLOW. *For the statement* c: **cobegin** $S \parallel T$ **coend**,

$$\frac{at\ S \rightsquigarrow after\ S, \qquad at\ T \rightsquigarrow after\ T}{at\ c \rightsquigarrow after\ c.}$$

PROOF. The proof requires the safety property

$$I\colon\ (\Box\ in\ c) \supset ((after\ S \supset \Box\ after\ S) \wedge (after\ T \supset \Box\ after\ T)),$$

which states that if control never leaves the **cobegin** statement, then once it reaches the end of one of the processes it remains there forever. This follows from the definition of *after* and the fact that any atomic action that makes *after* S or *after* T false has to make *after* c false too.

Termination of the **cobegin** is proved by the lattice of Figure 19, whose steps are justified as follows:

1. This is an application of the Single Exit Rule.
2. This step is just a rewriting, since $at\ c = (at\ S \wedge at\ T)$.
3. The hypothesis (recorded on the outermost box) implies that both S and T terminate.
4. The safety property I (from the outermost box) implies that once control reaches the end of S or T, it remains there as long as the **cobegin** does not terminate.
5. This is implied by the temporal logic theorem $(\Diamond\Box P \wedge \Diamond\Box Q) \supset \Diamond\Box(P \wedge Q)$.
6. This is a rewriting, based on the fact that $(after\ S \wedge after\ T) = after\ c$. □

GENERAL STATEMENT RULE.

$$\frac{\{P\}\ S\ \{Q\}, \qquad \Box(in\ S \supset P), \qquad in\ S \rightsquigarrow after\ S}{in\ S \rightsquigarrow (after\ S \wedge Q).}$$

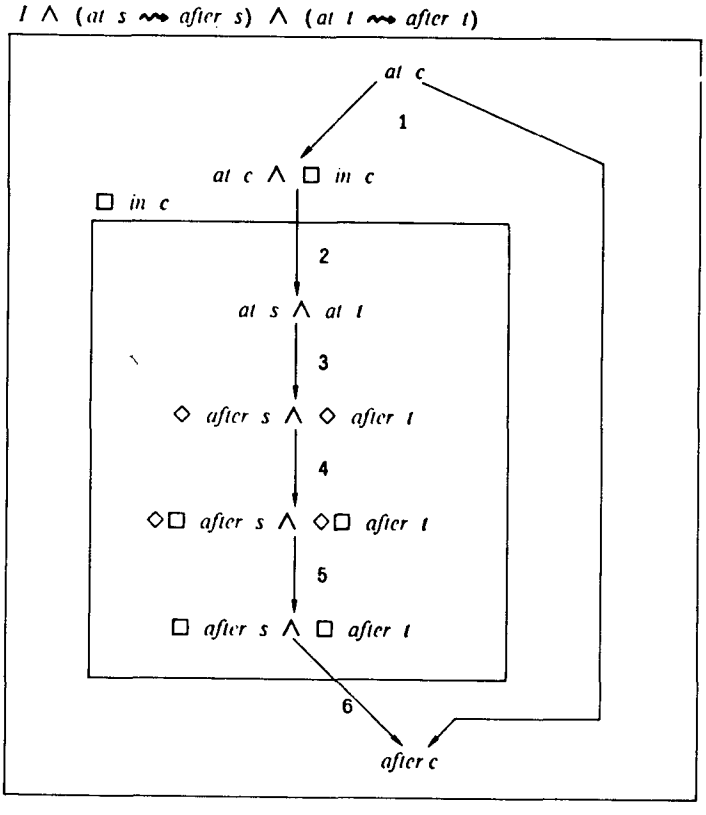

Fig. 19. Proof lattice for **cobegin** termination.

PROOF. The proof is similar to that of the Single Exit Rule. We make use of the safety property:

$$(in\ S \wedge \Box[(in\ S \supset P) \wedge \neg(after\ S \wedge Q)]) \supset \Box\ in\ S,$$

whose validity can be intuitively justified as follows. We assume $\{P\}\ S\ \{Q\}$ and $\Box(in\ S \supset P)$, so for any execution starting in S with P true, if S terminates then it must terminate with $after\ S \wedge Q$ true. Thus, if S terminates, it contradicts the assumption $\Box\neg(after\ S \wedge Q)$, so control can never reach $after\ S$. Hence, by the Single Exit Rule, control must remain forever in S.

Manipulating this safety property as before, we can obtain

$$(in\ S \wedge \Box(in\ S \supset P)) \supset (\Box\ in\ S \vee \Diamond(after\ S \wedge Q)).$$

Since we assume that S terminates, this can be simplified to

$$(in\ S \wedge \Box(in\ S \supset P)) \supset \Diamond(after\ S \wedge Q).$$

Since the second hypothesis of this implication is one of the assumptions of the General Statement Rule, we can conclude that $in\ S \supset \Diamond(after\ S \wedge Q)$. The result now follows from TL2. □

ATOMIC STATEMENT RULE. *For any atomic statement* $\langle S \rangle$:

$$\frac{\{P\}\ \langle S \rangle\ \{Q\}, \qquad \Box(at\ \langle S \rangle \supset P)}{at\ \langle S \rangle \rightsquigarrow (after\ \langle S \rangle \wedge Q).}$$

PROOF. This is an obvious corollary of the Atomic Liveness Axiom and the General Statement Rule. The former guarantees that the atomic statement terminates, while the latter guarantees that if it terminates, it terminates with Q true. □

while TEST RULE. *For the statement* w: **while** $\langle B \rangle$ **do** S **od**:

$$at\ w \rightsquigarrow ((at\ S \wedge B) \vee (after\ w \wedge \neg B)).$$

PROOF. This proof is similar to the proofs of the Single and General Statement Rules. In this case the starting safety property is

$$[at\ w \wedge \Box\sim((at\ S \wedge B) \vee (after\ w \wedge \neg B))] \supset \Box\ at\ w.$$

This is true because the semantics of the **while** test imply that if control leaves the point *at* w, then it reaches a state where either *at* $S \wedge B$ or *after* $w \wedge \neg B$ is true. The above implication can be rewritten as

$$at\ w \supset [\Box\ at\ w \vee \Diamond((at\ S \wedge B) \vee (after\ w \wedge \neg B))].$$

But the **while** Control Flow Axiom implies $\neg\Box\ at\ w$, so this reduces to

$$at\ w \supset \Diamond((at\ S \wedge B) \vee (after\ w \wedge \neg B)).$$

Applying TL2 gives the desired result. □

while EXIT RULE. *For the statement* w: **while** $\langle B \rangle$ **do** S **od**:

$$(at\ w \wedge \Box(at\ w \supset B)) \rightsquigarrow at\ S;$$
$$(at\ w \wedge \Box(at\ w \supset \neg B)) \rightsquigarrow after\ w.$$

PROOF. We give the proof for the first rule; the proof for the second is quite similar. We start with the safety property

$$[at\ w \wedge \Box(at\ w \supset B) \wedge \Box\ \neg at\ S] \supset \Box\ at\ w.$$

Rewriting it in the standard way gives

$$[at\ w \wedge \Box(at\ w \supset B)] \supset [\Box\ at\ w \vee \Diamond\ at\ S].$$

Since the **while** Control Flow Axiom guarantees $\sim\Box\ at\ w$, this reduces to

$$(at\ w \wedge \Box(at\ w \supset B)) \supset \Diamond\ at\ S,$$

which, after application of TL2, yields the desired result. □

ACKNOWLEDGMENTS

Many of the ideas in this paper were first put forward in a seminar on temporal logic at Stanford. We are particularly grateful to participants Sheldon Finkelstein, John Gilbert, Brent Hailpern, Dick Karp, Arthur Keller, Amy Lansky, Larry Paulsen, David Wall, and Nagatsugu Yamanouchi for helping us clarify our thinking. Amy Lansky, Pierre Wolper, and Nagatsugu Yamanouchi provided helpful comments on earlier drafts of the paper.

REFERENCES

1. BURSTALL, R.M. Program proving as hand simulation with a little induction. In Information Processing 74, Stockholm, 1974, pp. 308–312.
2. DIJKSTRA, E.W. The structure of the "THE"-multiprogramming system. *Commun. ACM 11*, 5 (May 1968), 341–346.
3. FLON, L., AND SUZUKI, N. The total correctness of parallel programs. *SIAM J. Comput. 10*, 2 (May 1981), 227–246.
4. FRANCEZ, N., AND PNUELI, A. A proof method for cyclic programs. *Acta Inf. 9*, 2 (1978), 133–158.
5. HABERMANN, A.N. Synchronization of communicating processes. *Commun. ACM 15*, 3 (Mar. 1972), 171–176.
6. HAREL, D., KOZEN, D., AND PARIKH, R. Process logic: Expressiveness, decidability, completeness. In Proceedings of the 21st Symposium on the Foundations of Computer Science, IEEE, Syracuse, N.Y., Oct. 1980, pp. 129–142.
7. KELLER, R.M. Formal verification of parallel programs. *Commun. ACM 19*, 7 (July 1976), 371–384.
8. KWONG, Y.S. On the absence of livelock in parallel programs. In *Lecture Notes in Computer Science*, vol. 70: *Semantics of Concurrent Computation*. Springer-Verlag, New York, 1979, pp. 172–190.
9. LAMPORT, L. The "Hoare logic" of concurrent programs. *Acta Inf. 14*, 1 (1980), 21–37.
10. LAMPORT, L. "Sometime" is sometimes "not never": On the temporal logic of programs. In Conference Record of the 7th Annual ACM Symposium on Principles of Programming Languages, Las Vegas, Nev., Jan. 28–30, 1980, pp. 174–185.
11. LAMPORT, L. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng. SE-3*, 2 (Mar. 1977), 125–143.
12. LIPTON, R. On Synchronization Primitive Systems. Ph.D. dissertation, Dep. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1974.
13. MANNA, Z., AND WALDINGER, R. Is "sometime" sometimes better than "always"? Intermittent assertions in proving program correctness. *Commun. ACM 21*, 2 (Feb. 1978), 159–172.
14. OWICKI, S., AND GRIES, D. An axiomatic proof technique for parallel programs. *Acta Inf. 6*, 4 (1976), 319–340.
15. PNUELI, A. The temporal semantics of concurrent programs. In *Lecture Notes in Computer Science*, vol. 70: *Semantics of Concurrent Computation*. Springer-Verlag, New York, 1979, pp. 1–20.
16. PNUELI, A. The temporal logic of programs. In Proceedings of the 18th Symposium on the Foundations of Computer Science, IEEE, Providence, Nov. 1977, pp. 46–57.
17. PRATT, V.R. Process logic: Preliminary report. In Conference Record of the 6th Annual ACM Symposium on Principles of Programming Languages, San Antonio, Tex., Jan. 29–31, 1979, pp. 93–100.
18. PRATT, V.R. Semantical considerations on Floyd-Hoare logic. In 17th Symposium on Foundations of Computer Science, IEEE, Houston, Tex., Oct. 1976, pp. 109–121.

Received November 1980; revised November 1981; accepted November 1981